The Supplemental Isabelle/HOL Library

April 19, 2009

Contents

1 Abstract-Rat: Abstract rational numbers

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

types $Num = int \times int$

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
  $isnormNum = (\lambda(a,b).\ (if\ a = 0\ then\ b = 0\ else\ b > 0 \wedge zgcd\ a\ b = 1))$

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

declare *zgcd-zdvd1*[*presburger*]
declare *zgcd-zdvd2*[*presburger*]
lemma *normNum-isnormNum* [*simp*]: $isnormNum\ (normNum\ x)$
proof −
  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 $?g = zgcd\ a\ b$
    let $?a' = a\ div\ ?g$
    let $?b' = b\ div\ ?g$
    let $?g' = zgcd\ ?a'\ ?b'$
    from *anz bnz* **have** $?g \neq 0$ **by** *simp* **with** *zgcd-pos*[*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 add*:*zgcd-def*)+

from *anz bnz* **have** *stupid*: $a \neq 0 \vee b \neq 0$ **by** *arith*
from *div-zgcd-relprime*[*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 fst-conv snd-conv*)**}**
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* $g = zgcd\ (a*a')\ (b*b')$
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*: *zgcd-def isnormNum-def Let-def Nmul-def split-def*)**}**
moreover
{assume *a′* = *0*
hence *?thesis* **using** *yn ab ab′*
by (*simp add*: *zgcd-def 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*: *zgcd-commute*)

lemma *isnormNum-int*[*simp*]:
isnormNum 0$_N$ *isnormNum* (*1*::*int*)$_N$ *i* ≠ *0* ⟹ *isnormNum i*$_N$
by (*simp-all add*: *isnormNum-def zgcd-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*::{*ring-char-0*,*field*, *division-by-zero*}) = *INum y*) = (*x* = *y*) (**is** *?lhs* = *?rhs*)
proof
have ∃ *a b a' b'*. *x* = (*a*,*b*) ∧ *y* = (*a'*,*b'*) **by** *auto*
then obtain *a b a' b'* **where** *xy*[*simp*]: *x* = (*a*,*b*) *y*=(*a'*,*b'*) **by** *blast*
assume *H*: *?lhs*
{assume *a* = *0* ∨ *b* = *0* ∨ *a'* = *0* ∨ *b'* = *0* **hence** *?rhs*
using *na nb H*
apply (*simp add*: *INum-def split-def isnormNum-def*)
apply (*cases a* = *0*, *simp-all*)
apply (*cases b* = *0*, *simp-all*)
apply (*cases a'* = *0*, *simp-all*)
apply (*cases a'* = *0*, *simp-all add*: *of-int-eq-0-iff*)
done}

```
  moreover
  { assume az: a ≠ 0 and bz: b ≠ 0 and a'z: a'≠0 and b'z: b'≠0
   from az bz a'z b'z na nb have pos: b > 0 b' > 0 by (simp-all add: isnormNum-def)
    from prems have eq:a * b' = a'*b
     by (simp add: INum-def  eq-divide-eq divide-eq-eq of-int-mult[symmetric] del:
of-int-mult)
    from prems have gcd1: zgcd a b = 1 zgcd b a = 1 zgcd a' b' = 1 zgcd b' a' =
1
     by (simp-all add: isnormNum-def add: zgcd-commute)
    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-dvd-eq[OF bz zrelprime-dvd-mult[OF gcd1(2) raw-dvd(2)]
      zrelprime-dvd-mult[OF gcd1(4) raw-dvd(4)]]
     have eq1: b = b' using pos by 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::{ring-char-0,
field,division-by-zero})) = (x = 0_N)
  unfolding INum-int(2)[symmetric]
  by (rule isnormNum-unique, simp-all)

lemma of-int-div-aux: d ~= 0 ==> ((of-int x)::'a::{field, ring-char-0}) / (of-int
d) =
    of-int (x div d) + (of-int (x mod d)) / ((of-int d)::'a)
proof -
  assume d ~= 0
  hence dz: of-int d ≠ (0::'a) by (simp add: of-int-eq-0-iff)
  let ?t = of-int (x div d) * ((of-int d)::'a) + of-int(x mod d)
  let ?f = 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*, *ring-char-0*}) = *of-int n* / *of-int d*
apply (*frule of-int-div-aux* [*of d n*, **where** *?'a* = '*a*])
apply *simp*
apply (*simp add*: *dvd-eq-mod-eq-0*)
done

lemma *normNum*[*simp*]: *INum* (*normNum x*) = (*INum x* :: '*a*::{*ring-char-0*,*field*, *division-by-zero*})
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *auto*
then obtain *a b* **where** *x*[*simp*]: *x* = (*a*,*b*) **by** *blast*
{assume *a*=*0* ∨ *b* = *0* **hence** *?thesis*
by (*simp add*: *INum-def normNum-def split-def Let-def*)**}**
moreover
{assume *a*: *a*≠*0* **and** *b*: *b*≠*0*
let *?g* = *zgcd 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*, *division-by-zero*, *ring-char-0*})
= *INum y* ⟷ *normNum x* = *normNum y* (**is** *?lhs* = *?rhs*)
proof −
have *normNum x* = *normNum y* ⟷ (*INum* (*normNum x*) :: '*a*) = *INum* (*normNum y*)
by (*simp del*: *normNum*)
also have ... = *?lhs* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *Nadd*[*simp*]: *INum* (*x* +$_N$ *y*) = *INum x* + (*INum y* :: '*a* :: {*ring-char-0*,*division-by-zero*,*field*})
proof−
let *?z* = *0*:: '*a*
have ∃ *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 = zgcd (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 zgcd-zdvd1*[**where** *i=a ∗ b′ + b ∗ a′* **and** *j=b∗b′*]] *of-int-div*[**where** *?′a = ′a*,
OF gz zgcd-zdvd2[**where** *i=a ∗ b′ + b ∗ a′* **and** *j=b∗b′*]]
by (*simp add*: *x y Nadd-def INum-def normNum-def Let-def add-divide-distrib*)**}**
ultimately have *?thesis* **using** *aa′ bb′*
by (*simp add*: *Nadd-def INum-def normNum-def x y Let-def*) **}**
ultimately show *?thesis* **by** *blast*
qed

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

proof−
let *?z = 0*::*′a*
have ∃ *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=zgcd (a∗a′) (b∗b′)*
have *gz*: *?g ≠ 0* **using** *z* **by** *simp*
from *z of-int-div*[**where** *?′a = ′a*, *OF gz zgcd-zdvd1*[**where** *i=a∗a′* **and** *j=b∗b′*]]
of-int-div[**where** *?′a = ′a* , *OF gz zgcd-zdvd2*[**where** *i=a∗a′* **and** *j=b∗b′*]]
have *?thesis* **by** (*simp add*: *Nmul-def x y Let-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *Nneg*[*simp*]: *INum (~N x) = − (INum x ::′a:: field)*
by (*simp add*: *Nneg-def split-def INum-def*)

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

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

x)
by (*simp add*: *Ninv-def INum-def split-def*)

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

lemma *Nlt0-iff*[*simp*]: **assumes** *nx*: *isnormNum x*
shows ((*INum x* :: '*a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*})< *0*) = *0*>$_N$ *x*
proof−
have ∃ *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* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) ≤ *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* :: {*ring-char-0*,*division-by-zero*,*ordere*
0) = *0*<$_N$ *x*
proof−
have ∃ *a b*. *x* = (*a*,*b*) **by** *simp*
then obtain *a b* **where** *x*[*simp*]:*x* = (*a*,*b*) **by** *blast*
{assume *a* = *0* **hence** *?thesis* **by** (*simp add*: *Ngt0-def INum-def*) **}**
moreover
{assume *a*: *a*≠*0* **hence** *b*: (*of-int b*::'*a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-less-divide-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::'*a*]
have *?thesis* **by** (*simp add*: *Ngt0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed
lemma *Nge0-iff*[*simp*]:**assumes** *nx*: *isnormNum x*
shows ((*INum x* :: '*a* :: {*ring-char-0*,*division-by-zero*,*ordered-field*}) ≥ *0*) = *0*≤$_N$ *x*

proof−
have $\exists\ a\ b.\ x = (a,b)$ **by** *simp*
then obtain *a b* **where** *x*[*simp*]:$x = (a,b)$ **by** *blast*
{assume $a = 0$ **hence** *?thesis* **by** (*simp add*: *Nge0-def INum-def*) **}**
moreover
{assume *a*: $a \neq 0$ **hence** *b*: (*of-int b*::'*a*) > *0* **using** *nx* **by** (*simp add*: *isnormNum-def*)
from *pos-le-divide-eq*[*OF b*, **where** *b*=*of-int a* **and** *a*=*0*::'*a*]
have *?thesis* **by** (*simp add*: *Nge0-def INum-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

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

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

lemma *Nadd-commute*:
assumes *SORT-CONSTRAINT*('*a*::{*ring-char-0*,*division-by-zero*,*field*})
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*::{*ring-char-0*,*division-by-zero*,*field*})
shows (*0*, *b*) $+_N\ y$ = *normNum y*
and (*a*, *0*) $+_N\ y$ = *normNum y*
and $x +_N$ (*0*, *b*) = *normNum x*
and $x +_N$ (*a*, *0*) = *normNum x*
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*::{*ring-char-0*,*division-by-zero*,*field*})
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*::{*ring-char-0*,*division-by-zero*,*field*})
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*::{*ring-char-0*,*division-by-zero*,*field*})
shows *normNum* (*x* +$_N$ *y*) = *x* +$_N$ *y* **by** *simp*

lemma *Nadd-normNum1*[*simp*]:
assumes *SORT-CONSTRAINT*('*a*::{*ring-char-0*,*division-by-zero*,*field*})
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*::{*ring-char-0*,*division-by-zero*,*field*})
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*::{*ring-char-0*,*division-by-zero*,*field*})
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 zgcd-commute mult-commute*)

lemma *Nmul-assoc*:
assumes *SORT-CONSTRAINT*('*a*::{*ring-char-0*,*division-by-zero*,*field*})
assumes *nx*: *isnormNum x* **and** *ny*:*isnormNum y* **and** *nz*:*isnormNum z*
shows $x *_N y *_N z = x *_N (y *_N z)$
proof−
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*::{*ring-char-0*,*division-by-zero*,*field*})
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 ... = $(INum\ x = (INum\ y :: 'a))$ **by** *simp*
also have ... = $(x = y)$ **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*::{*ring-char-0*,*division-by-zero*,*field*})
assumes *nx*:*isnormNum x* **and** *ny*: *isnormNum y*
shows $(x *_N y = 0_N) = (x = 0_N \vee y = 0_N)$
proof−
{ 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*])
apply (*simp add*: *INum-def split-def isnormNum-def fst-conv snd-conv*)
apply (*cases a=0*,*simp-all*)
apply (*cases a'=0*,*simp-all*)
done
}
qed
lemma *Nneg-Nneg*[*simp*]: $\sim_N$ ($\sim_N$ *c*) = *c*
by (*simp add*: *Nneg-def split-def*)

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

end

2 AssocList: Map operations implemented on association lists

theory *AssocList*
imports *Map Main*
begin

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

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

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

primrec
updates :: *'key list* ⇒ *'val list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where

```
    updates [] vs ps = ps
  | updates (k#ks) vs ps = (case vs
      of [] ⇒ ps
       | (v#vs') ⇒ updates ks vs' (update k v ps))

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

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

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

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

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

primrec
  map-ran :: ('key ⇒ 'val ⇒ 'val) ⇒ ('key × 'val) list ⇒ ('key × 'val) list
```

where
  map-ran f [] = []
| *map-ran f* (*p*#*ps*) = (*fst p*, *f* (*fst p*) (*snd p*)) # *map-ran f ps*

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

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

2.1 *delete*

lemma *delete-eq*:
  delete k xs = *filter* (λ*p*. *fst p* ≠ *k*) *xs*
  by (*induct xs*) *auto*

lemma *delete-id* [*simp*]: *k* ∉ *fst* ‘ *set al* ⟹ *delete k al* = *al*
  by (*induct al*) *auto*

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

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

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

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

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

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

lemma *distinct-delete*:
  assumes *distinct* (*map fst al*)
  shows *distinct* (*map fst* (*delete k al*))
using *assms*
proof (*induct al*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a al*)
  from *Cons.prems* **obtain**

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

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

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

2.2 *clearjunk*

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

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

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

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

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

lemma map-of-clearjunk: map-of (clearjunk al) = map-of al
  apply (rule ext)
  apply (induct al rule: clearjunk.induct)
  apply  simp
  apply (simp add: map-of-filter)
```

done

lemma *length-clearjunk*: *length* (*clearjunk al*) ≤ *length al*
proof (*induct al rule*: *clearjunk.induct* [*case-names Nil Cons*])
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons p ps*)
from *Cons* **have** *length* (*clearjunk* [*q*←*ps* . *fst q* ≠ *fst p*]) ≤ *length* [*q*←*ps* . *fst q* ≠ *fst p*]
by (*simp add*: *delete-eq*)
also have ... ≤ *length ps*
by *simp*
finally show *?case*
by (*simp add*: *delete-eq*)
qed

lemma *notin-fst-filter*: *a* ∉ *fst* ‘ *set ps* ⟹ [*q*←*ps* . *fst q* ≠ *a*] = *ps*
by (*induct ps*) *auto*

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

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

2.3 *dom* and *ran*

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

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

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

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

have *ran* (*map-of* (*a* # *al*)) = {*snd a*} ∪ *ran* (*map-of al*)
proof

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

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

2.4 *update*

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

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

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

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

lemma *update-filter*:
$a \neq k \Longrightarrow$ *update k v* [$q \leftarrow ps$. *fst q* $\neq a$] = [$q \leftarrow$ *update k v ps* . *fst q* $\neq a$]
by (*induct ps*) *auto*

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

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

lemma *update-Some-unfold*:
(*map-of* (*update k v al*) *x* = *Some y*) =
(*x* = *k* ∧ *v* = *y* ∨ *x* ≠ *k* ∧ *map-of al x* = *Some y*)
by (*simp add*: *update-conv'* *map-upd-Some-unfold*)

lemma *image-update*[*simp*]: *x* ∉ *A* ⟹ *map-of* (*update x y al*) ‘ *A* = *map-of al* ‘ *A*
by (*simp add*: *update-conv'* *image-map-upd*)

2.5 *updates*

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

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

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

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

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

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

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

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

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

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

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

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

lemma *updates-append2-drop*[*simp*]:
size xs = *size ys* ⟹ *updates xs* (*ys*@*zs*) *al* = *updates xs ys al*
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

2.6 *map-ran*

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

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

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

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

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

2.7 *merge*

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

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

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

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

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

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

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

lemma *merge-Some-iff*:
(*map-of* (*merge m n*) *k* = *Some x*) =
(*map-of n k* = *Some x* ∨ *map-of n k* = *None* ∧ *map-of m k* = *Some x*)
by (*simp add*: *merge-conv' map-add-Some-iff*)

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

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

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

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

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

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

2.8 *compose*

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

lemma *compose-conv*:
shows *map-of* (*compose xs ys*) *k* = (*map-of ys* ∘$_m$ *map-of xs*) *k*
proof (*induct xs ys rule*: *compose.induct*)

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

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

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

lemma dom-compose: fst ` set (compose xs ys) ⊆ fst ` set xs
proof (induct xs ys rule: compose.induct)
  case 1 thus ?case by simp
next
  case (2 x xs ys)
```

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

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

lemma compose-delete-twist: (compose (delete k xs) ys) = delete k (compose xs
ys)
proof (induct xs ys rule: compose.induct)
  case 1 thus ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
```

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

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

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

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

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

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

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

2.9 *restrict*

lemma *restrict-eq*:
restrict A = filter (λ*p*. *fst p* ∈ *A*)
proof
fix *xs*
show *restrict A xs = filter* (λ*p*. *fst p* ∈ *A*) *xs*
by (*induct xs*) *auto*
qed

lemma *distinct-restr*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*restrict A al*))
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
apply (*induct al*)
apply (*simp add*: *restrict-eq*)
apply (*cases k*∈*A*)
apply (*auto simp add*: *restrict-eq*)
done

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

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

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

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

lemma *update-restr*:
map-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*

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

end

3 SetsAndFunctions: Operations on sets and functions

theory *SetsAndFunctions*
imports *Main*
begin

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

3.1 Basic definitions

definition
set-plus :: ('*a*::*plus*) *set* => '*a set* => '*a set* (**infixl** ⊕ *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**
$A \otimes B == \{c.\ EX\ a{:}A.\ EX\ b{:}B.\ c = a * b\}$

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

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

instance ..

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

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

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

3.2 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-add*) +*o* *B*) ⊕ *C* =
a +*o* (*B* ⊕ *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* ⊕ ((*a*::'*a*::*comm-monoid-add*) +*o* *D*) =
a +*o* (*C* ⊕ *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 ⊕ E <= D ⊕ F
by (*auto simp add*: *set-plus-def*)

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

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

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

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

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

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

lemma *set-plus-mono4-b*: (*a*::'*a*::*comm-monoid-add*) *: C ==>*

$x : a +o D ==> x : D \oplus C$
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 ⊗ 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 ⊗ ((a::'a::comm-monoid-mult) *o D) =
    a *o (C ⊗ 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 ⊗ E <= D ⊗ F
  by (auto simp add: set-times-def)

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

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

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

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

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

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

lemma *set-times-mono4-b*: $(a::'a::comm\text{-}monoid\text{-}mult) : C ==>$
$x : a *o D ==> x : D \otimes C$
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*: $(a::'a::ring\text{-}1) : (- 1) *o C ==>$
$- a : C$
by (*auto simp add*: *elt-set-times-def*)

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

end

4 BigO: Big O notation

theory *BigO*
imports *Complex-Main SetsAndFunctions*
begin

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

The main changes in this version are as follows:

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

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

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

4.1 Definitions

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

lemma *bigo-pos-const*: $(EX\ (c::'a::ordered\text{-}idom).$
$ALL\ x.\ (abs\ (h\ x)) <= (c * (abs\ (f\ x))))$
$= (EX\ c.\ 0 < c\ \&\ (ALL\ x.\ (abs(h\ x)) <= (c * (abs\ (f\ x)))))$
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*]: $f : g +o O(h) ==> O(f) <= O(g) \oplus O(h)$

proof −
assume $f : g +o O(h)$
also have ... $<= O(g) \oplus O(h)$
by (*auto del*: *subsetI*)
also have ... $= O(\%x.\ abs(g\ x)) \oplus O(\%x.\ abs(h\ x))$
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 ... $= O(\%x.\ abs(g\ x)) \oplus O(\%x.\ abs(h\ x))$
by (*rule bigo-plus-eq*, *auto*)
finally show *?thesis*
by (*simp add*: *bigo-abs3* [*symmetric*])
qed

lemma *bigo-mult* [*intro*]: $O(f) \otimes O(g) <= O(f * g)$
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*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) <= (f::'a => ('b::ordered\text{-}field)) *o O(g)$
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*: $ALL\ x.\ f\ x \sim= 0 ==>$
$O(f * g) = (f::'a => ('b::ordered\text{-}field)) *o O(g)$
apply (*rule equalityI*)
apply (*erule bigo-mult5*)
apply (*rule bigo-mult2*)
done

lemma *bigo-mult7*: $ALL\ x.\ f\ x \sim= 0 ==>$

$O(f * g) <= O(f::'a => ('b::ordered\text{-}field)) \otimes O(g)$
apply (*subst bigo-mult6*)
apply *assumption*
apply (*rule set-times-mono3*)
apply (*rule bigo-refl*)
done

lemma *bigo-mult8*: $ALL\ x.\ f\ x \sim= 0 ==> O(f * g) = O(f::'a => ('b::ordered\text{-}field)) \otimes O(g)$
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 $f +o O(g) <= O(f) \oplus O(g)$
by (*auto del*: *subsetI*)
also have $... <= O(g) \oplus O(g)$
proof −
from *a* **have** $O(f) <= O(g)$ **by** (*auto del*: *subsetI*)
thus *?thesis* **by** (*auto del*: *subsetI*)
qed
also have $... <= O(g)$ **by** (*simp add*: *bigo-plus-idemp*)
finally show *?thesis* **.**
qed
qed

lemma *bigo-plus-absorb-lemma2*: $f : O(g) ==> O(g) <= f +o O(g)$
proof −
assume *a*: $f : O(g)$
show $O(g) <= f +o O(g)$

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

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

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

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

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

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

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

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

done

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

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

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

lemma *bigo-const-mult2*: *O*(%*x*. *c* * *f x*) <= *O*(*f*)
by (*rule bigo-elt-subset*, *rule bigo-const-mult1*)

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

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

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

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

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

lemma *bigo-const-mult7* [*intro*]: $f =o\ O(g) ==> (\%x.\ c * f\ x) =o\ O(g)$
proof −
assume $f =o\ O(g)$
then have $(\%x.\ c) * f =o\ (\%x.\ c) *o\ O(g)$
by *auto*
also have $(\%x.\ c) * f = (\%x.\ c * f\ x)$
by (*simp add*: *func-times*)
also have $(\%x.\ c) *o\ O(g) <= O(g)$
by (*auto del*: *subsetI*)
finally show *?thesis* **.**
qed

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

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

4.2 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*)
apply *force*
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

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

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

lemma *bigo-fix2*:
(%*x*. *f* ((*x*::*nat*) + *1*)) =*o* (%*x*. *g*(*x* + *1*)) +*o* *O*(%*x*. *h*(*x* + *1*)) ==>
f *0* = *g* *0* ==> *f* =*o* *g* +*o* *O*(*h*)
apply (*rule set-minus-imp-plus*)
apply (*rule bigo-fix*)
apply (*subst fun-diff-def*)
apply (*subst fun-diff-def* [*symmetric*])
apply (*rule set-plus-imp-minus*)
apply *simp*
apply (*simp add*: *fun-diff-def*)
done

4.4 Less than or equal to

definition

```
  lesso :: ('a => 'b::ordered-idom) => ('a => 'b) => ('a => 'b)
    (infixl <o 70) where
  f <o g = (%x. max (f x - g x) 0)

lemma bigo-lesseq1: f =o O(h) ==> ALL x. abs (g x) <= abs (f x) ==>
    g =o O(h)
  apply (unfold bigo-def)
  apply clarsimp
  apply (rule-tac x = c in exI)
  apply (rule allI)
  apply (rule order-trans)
  apply (erule spec)+
  done

lemma bigo-lesseq2: f =o O(h) ==> ALL x. abs (g x) <= f x ==>
      g =o O(h)
  apply (erule bigo-lesseq1)
  apply (rule allI)
  apply (drule-tac x = x in spec)
  apply (rule order-trans)
  apply assumption
  apply (rule abs-ge-self)
  done

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

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

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

apply (*simp split*: *split-max*)
done

lemma *bigo-lesso2*: *f* =*o* *g* +*o* *O*(*h*) ==>
ALL x. *0* <= *k x* ==> *ALL x*. *k x* <= *f x* ==>
k <*o* *g* =*o* *O*(*h*)
apply (*unfold lesso-def*)
apply (*rule bigo-lesseq4*)
apply (*erule set-plus-imp-minus*)
apply (*rule allI*)
apply (*rule le-maxI2*)
apply (*rule allI*)
apply (*subst 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*: $f <o g =o O(k::'a=>'b::ordered\text{-}field) ==> g =o h +o O(k) ==> f <o h =o O(k)$
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-def 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

5 Binomial: Binomial Coefficients

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

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

primrec *binomial* :: $nat \Rightarrow nat \Rightarrow nat$ (**infixl** *choose 65*) **where**
binomial-0: $(0\ choose\ k) = (if\ k = 0\ then\ 1\ else\ 0)$
| *binomial-Suc*: $(Suc\ n\ choose\ k) =$
$(if\ k = 0\ then\ 1\ else\ (n\ choose\ (k - 1)) + (n\ choose\ k))$

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

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

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

by *simp*

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

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

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

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

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

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

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

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

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

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

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

Another version, with -1 instead of Suc.

lemma *times-binomial-minus1-eq*:
[|*k* ≤ *n*; *0*<*k*|] ==> (*n choose k*) ∗ *k* = *n* ∗ ((*n* − *1*) *choose* (*k* − *1*))

apply (*cut-tac* $n = n - 1$ **and** $k = k - 1$ **in** *Suc-times-binomial-eq*)
apply (*simp split add*: *nat-diff-split*, *auto*)
done

5.1 Theorems about *choose*

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

lemma *card-s-0-eq-empty*:
$\quad$ *finite* A ==> *card* $\{B.\ B \subseteq A\ \&\ \mathit{card}\ B = 0\} = 1$
apply (*simp cong add*: *conj-cong add*: *finite-subset* [*THEN card-0-eq*])
apply (*simp cong add*: *rev-conj-cong*)
done

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

lemma *finite-bex-subset*[*simp*]:
finite $B \Longrightarrow (!!A.\ A<=B \Longrightarrow \mathit{finite}\{x.\ P\ x\ A\}) \Longrightarrow \mathit{finite}\{x.\ \mathit{EX}\ A<=B.\ P\ x\ A\}$
apply(*subgoal-tac* $\{x.\ \mathit{EX}\ A<=B.\ P\ x\ A\} = (\mathit{UN}\ A{:}\mathit{Pow}\ B.\ \{x.\ P\ x\ A\})$)
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*:
$\quad$ [|*finite* A; $x \notin A$|] ==>
$\quad$ *card* $\{B.\ \mathit{EX}\ C.\ C <= A\ \&\ \mathit{card}(C) = k\ \&\ B = \mathit{insert}\ x\ C\} =$
$\quad$ *card* $\{B.\ B <= A\ \&\ \mathit{card}(B) = k\}$
apply (*rule-tac* $f = \%s.\ s - \{x\}$ **and** $g = \mathit{insert}\ x$ **in** *card-bij-eq*)
$\quad$ **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*:
$\quad$!!A. *finite* A ==> *card* $\{B.\ B <= A\ \&\ \mathit{card}\ B = k\} = (\mathit{card}\ A\ \mathit{choose}\ k)$

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

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

The binomial theorem (courtesy of Tobias Nipkow):

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

qed

5.2 Pochhammer's symbol : generalized raising factorial

definition *pochhammer* (*a*::'*a*::*comm-semiring-1*) *n* = (*if n* = *0 then 1 else setprod* (*λn. 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* (*λn. 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*} ∩ {*Suc n*} = {} **by** *auto*
have *eq*: {*0*..*Suc n*} = {*0*..*n*} ∪ {*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*} ∩ {*1*.. *Suc n*} = {} **by** *auto*
have *eq*: {*0*..*Suc n*} = {*0*} ∪ {*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 ring-simps)
    have ?thesis apply (simp add: pochhammer-def)
    unfolding setprod-insert[OF th0, unfolded eq]
    using th1 by (simp add: ring-simps)}
ultimately show ?thesis by blast
qed

lemma fact-setprod: fact n = setprod id {1 .. n}
  apply (induct n, simp)
  apply (simp only: fact-Suc atLeastAtMostSuc-conv)
  apply (subst setprod-insert)
  by simp-all

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

  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*])

5.3 Generalized binomial coefficients

definition *gbinomial* :: *'a*::{*field*, *recpower*,*ring-char-0*} ⇒ *nat* ⇒ *'a* (**infixl** *gchoose* *65*)
where *a gchoose n* = (*if n* = *0 then 1 else* (*setprod* (λ*i*. *a* − *of-nat i*) {*0* .. *n* − *1*}) / *of-nat* (*fact n*))

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

lemma *gbinomial-pochhammer*: *a gchoose n* = (− *1*) ^ *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*)) ^ *n* = *setprod* (λ*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 ≤ *n* ⟹ *fact k* ∗ *fact* (*n* − *k*) ∗ (*n choose k*) = *fact n*
proof(*induct n arbitrary*: *k rule*: *nat-less-induct*)
fix *n k* **assume** *H*: ∀ *m*<*n*. ∀ *x*≤*m*. *fact x* ∗ *fact* (*m* − *x*) ∗ (*m choose x*) =
fact m **and** *kn*: *k* ≤ *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: ring-simps)
    also have ... = (k + (m − h)) * fact m
      using H[rule-format, OF mn hm'] H[rule-format, OF mn km]
      by (simp add: ring-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, ring-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 fact-not-eq-zero 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 ring-simps of-nat-add[symmetric] del: of-nat-add)
    have th0: finite {1..n − Suc h} finite {n − h .. n}
```

$\{1..n - Suc\ h\} \cap \{n - h\ ..\ n\} = \{\}$ **and** *eq3*: $\{1..n - Suc\ h\} \cup \{n - h\ ..\ n\} = \{1..n\}$ **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-setprod 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*: *ring-simps*)
finally show *?thesis* **..**
qed

lemma *gbinomial-mult-1'*: (*a gchoose n*) ∗ *a* = *of-nat n* ∗ (*a gchoose n*) + *of-nat* (*Suc n*) ∗ (*a gchoose* (*Suc n*))
by (*simp add*: *mult-commute gbinomial-mult-1*)

lemma *gbinomial-Suc*: *a gchoose* (*Suc k*) = (*setprod* (λ*i*. *a* − *of-nat i*) {*0* .. *k*})

```
/ of-nat (fact (Suc k))
  by (simp add: gbinomial-def)

lemma gbinomial-mult-fact:
  (of-nat (fact (Suc k)) :: 'a) * ((a::'a::{field, ring-char-0,recpower}) gchoose (Suc
k)) = (setprod (λi. a − of-nat i) {0 .. k})
  unfolding gbinomial-Suc
  by (simp-all add: field-simps del: fact-Suc)

lemma gbinomial-mult-fact':
  ((a::'a::{field, ring-char-0,recpower}) gchoose (Suc k)) * (of-nat (fact (Suc k))
:: 'a) = (setprod (λi. a − of-nat i) {0 .. k})
  using gbinomial-mult-fact[of k a]
  apply (subst mult-commute) .

lemma gbinomial-Suc-Suc: ((a::'a::{field,recpower, ring-char-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: (∏ i∈{1..k}. (a + 1) − of-nat i) = (∏ i∈{0..h}. a − of-nat i)
     apply (rule strong-setprod-reindex-cong[where f = Suc])
     using h by auto

    have of-nat (fact (Suc k)) * (a gchoose k + (a gchoose (Suc k))) = ((a gchoose
Suc h) * of-nat (fact (Suc h)) * of-nat (Suc k)) + (∏ i∈{0::nat..Suc h}. a −
of-nat i)
      unfolding h
      apply (simp add: ring-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: ring-simps)
    also have ... = (∏ i∈{0..h}. a − of-nat i) * (a + 1)
      unfolding gbinomial-mult-fact' setprod-nat-ivl-Suc
      by (simp add: ring-simps h)
    also have ... = (∏ i∈{0..k}. (a + 1) − of-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
```

end

6 Bit: The Field of Integers mod 2

theory *Bit*
imports *Main*
begin

6.1 Bits as a datatype

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

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 ∀ *b*. *P* (*Abs-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*) ≠ (*1*::*bit*)
unfolding *zero-bit-def one-bit-def*
by (*simp add*: *Abs-bit-inject*)
qed

lemma *bit-not-0-iff* [*iff*]: (*x*::*bit*) ≠ *0* ⟷ *x* = *1*
by (*induct x*) *simp-all*

lemma *bit-not-1-iff* [*iff*]: (*x*::*bit*) ≠ *1* ⟷ *x* = *0*
by (*induct x*) *simp-all*

6.2 Type *bit* forms a field

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

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

definition *times-bit-def*:
$x * y = (\mathit{case}\ x\ \mathit{of}\ 0 \Rightarrow 0 \mid 1 \Rightarrow y)$

definition *uminus-bit-def* [*simp*]:
$- x = (x :: \mathit{bit})$

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

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

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

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*: $x + x = (0 :: \mathit{bit})$
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

lemma *bit-mult-eq-1-iff* [*simp*]: $x * y = (1 :: \mathit{bit}) \longleftrightarrow x = 1 \wedge y = 1$
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*: $x + y = (0 :: \mathit{bit}) \longleftrightarrow x = y$
unfolding *plus-bit-def* **by** (*simp split*: *bit.split*)

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

6.3 Numerals at type *bit*

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

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

instance proof
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

7 Boolean-Algebra: Boolean Algebras

theory *Boolean-Algebra*
imports *Main*
begin

locale *boolean* =
fixes *conj* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ⊓ *70*)
fixes *disj* :: *'a* ⇒ *'a* ⇒ *'a* (**infixr** ⊔ *65*)
fixes *compl* :: *'a* ⇒ *'a* (∼ - [*81*] *80*)
fixes *zero* :: *'a* (**0**)
fixes *one* :: *'a* (**1**)
assumes *conj-assoc*: (*x* ⊓ *y*) ⊓ *z* = *x* ⊓ (*y* ⊓ *z*)
assumes *disj-assoc*: (*x* ⊔ *y*) ⊔ *z* = *x* ⊔ (*y* ⊔ *z*)
assumes *conj-commute*: *x* ⊓ *y* = *y* ⊓ *x*
assumes *disj-commute*: *x* ⊔ *y* = *y* ⊔ *x*
assumes *conj-disj-distrib*: *x* ⊓ (*y* ⊔ *z*) = (*x* ⊓ *y*) ⊔ (*x* ⊓ *z*)
assumes *disj-conj-distrib*: *x* ⊔ (*y* ⊓ *z*) = (*x* ⊔ *y*) ⊓ (*x* ⊔ *z*)
assumes *conj-one-right* [*simp*]: *x* ⊓ **1** = *x*
assumes *disj-zero-right* [*simp*]: *x* ⊔ **0** = *x*
assumes *conj-cancel-right* [*simp*]: *x* ⊓ ∼ *x* = **0**
assumes *disj-cancel-right* [*simp*]: *x* ⊔ ∼ *x* = **1**
begin

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

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

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

7.1 Complement

lemma *complement-unique*:
assumes *1*: $a \sqcap x = \mathbf{0}$
assumes *2*: $a \sqcup x = \mathbf{1}$
assumes *3*: $a \sqcap y = \mathbf{0}$
assumes *4*: $a \sqcup y = \mathbf{1}$
shows $x = y$
proof −
have $(a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)$ **using** *1 3* **by** *simp*
hence $(x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)$ **using** *conj-commute* **by** *simp*
hence $x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)$ **using** *conj-disj-distrib* **by** *simp*
hence $x \sqcap \mathbf{1} = y \sqcap \mathbf{1}$ **using** *2 4* **by** *simp*
thus $x = y$ **using** *conj-one-right* **by** *simp*
qed

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

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

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

7.2 Conjunction

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

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

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

lemma *compl-one* [*simp*]: $\sim \mathbf{1} = \mathbf{0}$
by (*rule compl-unique* [*OF conj-zero-right disj-zero-right*])

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

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

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

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

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

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

7.3 Disjunction

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

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

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

by (*rule boolean.compl-one* [*OF dual*])

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

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

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

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

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

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

7.4 De Morgan's Laws

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

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

end

7.5 Symmetric Difference

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

begin

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

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

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

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

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

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

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

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

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

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

lemma *xor-compl-left* [*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

8 Product-ord: Order on product types

theory *Product-ord*

imports *Main*
begin

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

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

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

instance ..

end

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

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

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

end

end

9 Char-nat: Mapping between characters and natural numbers

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

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

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

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

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

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

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 Let-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*]

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

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

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

lemma *nat-of-nibble-eq*: *nat-of-nibble n* = *nat-of-nibble m* ⟷ *n* = *m*

by (*rule inj-eq*) (*rule inj-nat-of-nibble*)

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

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

Conversion between chars and nats.

definition
nibble-pair-of-nat :: *nat* ⇒ *nibble* × *nibble* **where**
nibble-pair-of-nat n = (*nibble-of-nat* (*n div 16*), *nibble-of-nat* (*n mod 16*))

lemma *nibble-of-pair* [*code*]:
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 List

code-modulename *OCaml*
Char-nat List

code-modulename *Haskell*

Char-nat List

end

10 Char-ord: Order on characters

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

instantiation *nibble* :: *linorder*
begin

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

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

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

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

setup ⟪
fold (*fn target => add-literal-char target*) [*SML*, *OCaml*, *Haskell*]
#> add-literal-list-string Haskell
⟫

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

code-reserved *SML*
char

code-reserved *OCaml*
char

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

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

end

12 Code-Integer: Pretty integer literals for code generation

theory *Code-Integer*
imports *Main*

begin

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

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

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

setup ⟪
fold (*Numeral.add-code* @{*const-name number-int-inst.number-of-int*}
true true) [*SML*, *OCaml*, *Haskell*]
⟫

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

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

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

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

code-const *uminus* :: *int* ⇒ *int*
(*SML IntInf.*~)

(*OCaml Big'-int.minus'-big'-int*)
(*Haskell negate*)

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

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

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

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

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

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

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

Evaluation

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

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

end

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

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

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

lemma [*code*]:
nat-of-char = nat o int-of-char
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*)

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

lemma [*code*, *code inline*]:
char-rec f c = split f (*nibble-pair-of-nat* (*nat-of-char c*))
by (*cases c*) (*auto simp add*: *nibble-pair-of-nat-char*)

lemma [*code*, *code inline*]:
char-case f c = split f (*nibble-pair-of-nat* (*nat-of-char c*))
by (*cases c*) (*auto simp add*: *nibble-pair-of-nat-char*)

lemma [*code*]:
size (*c*::*char*) = *0*
by (*cases c*) *auto*

code-const *int-of-char* **and** *char-of-int*
(*SML* !(*IntInf.fromInt o Char.ord*) **and** !(*Char.chr o IntInf.toInt*))
(*OCaml Big'-int.big'-int'-of'-int* (*Char.code* -) **and** *Char.chr* (*Big'-int.int'-of'-big'-int* -))
(*Haskell toInteger* (*fromEnum* (- :: *Char*)) **and** !(*let chr k* | *k* < *256* = *toEnum k* :: *Char in chr* . *fromInteger*))

end

14 Code-Index: Type of indices

theory *Code-Index*
imports *Main*

begin

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

14.1 Datatype of indices

typedef (**open**) *index* = *UNIV* :: *nat set*
  morphisms *nat-of of-nat* **by** *rule*

lemma *of-nat-nat-of* [*simp*]:
  of-nat (*nat-of k*) = *k*
  by (*rule nat-of-inverse*)

lemma *nat-of-of-nat* [*simp*]:
  nat-of (*of-nat n*) = *n*
  by (*rule of-nat-inverse*) (*rule UNIV-I*)

lemma [*measure-function*]:
  is-measure nat-of **by** (*rule is-measure-trivial*)

lemma *index*:
  (⋀*n*::*index*. *PROP P n*) ≡ (⋀*n*::*nat*. *PROP P* (*of-nat n*))
proof
  fix *n* :: *nat*
  assume ⋀*n*::*index*. *PROP P n*
  then show *PROP P* (*of-nat n*) **.**
next
  fix *n* :: *index*
  assume ⋀*n*::*nat*. *PROP P* (*of-nat n*)
  then have *PROP P* (*of-nat* (*nat-of n*)) **.**
  then show *PROP P n* **by** *simp*
qed

lemma *index-case*:
  assumes ⋀*n*. *k* = *of-nat n* ⟹ *P*
  shows *P*
  by (*rule assms* [*of nat-of k*]) *simp*

lemma *index-induct-raw*:
  assumes ⋀*n*. *P* (*of-nat n*)
  shows *P k*
proof −
  from *assms* **have** *P* (*of-nat* (*nat-of k*)) **.**
  then show *?thesis* **by** *simp*
qed

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

by (*rule nat-of-inject*)

lemma *of-nat-inject* [*simp*]:
of-nat n = of-nat m ⟷ n = m
by (*rule of-nat-inject*) (*rule UNIV-I*)+

instantiation *index* :: *zero*
begin

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

instance ..

end

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

rep-datatype *0* :: *index Suc-index*
proof −
fix *P* :: *index* ⇒ *bool*
fix *k* :: *index*
assume *P 0* **then have** *init*: *P* (*of-nat 0*) **by** *simp*
assume ⋀*k*. *P k* ⟹ *P* (*Suc-index k*)
then have ⋀*n*. *P* (*of-nat n*) ⟹ *P* (*Suc-index* (*of-nat n*)) **.**
then have *step*: ⋀*n*. *P* (*of-nat n*) ⟹ *P* (*of-nat* (*Suc n*)) **by** *simp*
from *init step* **have** *P* (*of-nat* (*nat-of k*))
by (*induct nat-of k*) *simp-all*
then show *P k* **by** *simp*
qed *simp-all*

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

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

lemma [*simp*, *code*]:
index-size = nat-of
proof (*rule ext*)
fix *k*
have *index-size k = nat-size* (*nat-of k*)
by (*induct k rule*: *index.induct*) (*simp-all del*: *zero-index-def Suc-index-def*, *simp-all*)
also have *nat-size* (*nat-of k*) = *nat-of k* **by** (*induct nat-of k*) *simp-all*
finally show *index-size k = nat-of k* **.**

qed

lemma [*simp*, *code*]:
size = *nat-of*
proof (*rule ext*)
fix *k*
show *size k* = *nat-of k*
by (*induct k*) (*simp-all del*: *zero-index-def Suc-index-def*, *simp-all*)
qed

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

lemma [*code*]:
eq-class.eq k l ⟷ *eq-class.eq* (*nat-of k*) (*nat-of l*)
by (*cases k*, *cases l*) (*simp add*: *eq*)

lemma [*code nbe*]:
eq-class.eq (*k*::*index*) *k* ⟷ *True*
by (*rule HOL.eq-refl*)

14.2 Indices as datatype of ints

instantiation *index* :: *number*
begin

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

instance ..

end

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

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

14.3 Basic arithmetic

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

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

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

definition [*simp*, *code del*]:

$n - m = \mathit{of\text{-}nat}\ (\mathit{nat\text{-}of}\ n - \mathit{nat\text{-}of}\ m)$

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

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

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

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

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

instance proof
qed (*auto simp add*: *left-distrib*)

end

lemma *zero-index-code* [*code inline*, *code*]:
$(0{::}\mathit{index}) = \mathit{Numeral0}$
by (*simp add*: *number-of-index-def Pls-def*)
lemma [*code post*]: $\mathit{Numeral0} = (0{::}\mathit{index})$
using *zero-index-code* **..**

lemma *one-index-code* [*code inline*, *code*]:
$(1{::}\mathit{index}) = \mathit{Numeral1}$
by (*simp add*: *number-of-index-def Pls-def Bit1-def*)
lemma [*code post*]: $\mathit{Numeral1} = (1{::}\mathit{index})$
using *one-index-code* **..**

lemma *plus-index-code* [*code nbe*]:
$\mathit{of\text{-}nat}\ n + \mathit{of\text{-}nat}\ m = \mathit{of\text{-}nat}\ (n + m)$
by *simp*

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

lemma *subtract-index-code* [*code nbe*]:
$\mathit{subtract\text{-}index}\ (\mathit{of\text{-}nat}\ n)\ (\mathit{of\text{-}nat}\ m) = \mathit{of\text{-}nat}\ (n - m)$
by *simp*

lemma *minus-index-code* [*code*]:
$n - m = \mathit{subtract\text{-}index}\ n\ m$
by *simp*

lemma *times-index-code* [*code nbe*]:
of-nat n ∗ *of-nat m* = *of-nat* (*n* ∗ *m*)
by *simp*

lemma *less-eq-index-code* [*code nbe*]:
of-nat n ≤ *of-nat m* ⟷ *n* ≤ *m*
by *simp*

lemma *less-index-code* [*code nbe*]:
of-nat n < *of-nat m* ⟷ *n* < *m*
by *simp*

lemma *Suc-index-minus-one*: *Suc-index n* − *1* = *n* **by** *simp*

lemma *of-nat-code* [*code*]:
of-nat = *Nat.of-nat*
proof
fix *n* :: *nat*
have *Nat.of-nat n* = *of-nat n*
by (*induct n*) *simp-all*
then show *of-nat n* = *Nat.of-nat n*
by (*rule sym*)
qed

lemma *index-not-eq-zero*: *i* ≠ *of-nat 0* ⟷ *i* ≥ *1*
by (*cases i*) *auto*

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

lemma *nat-of-aux-code* [*code*]:
nat-of-aux i n = (*if i* = *0 then n else nat-of-aux* (*i* − *1*) (*Suc n*))
by (*auto simp add*: *nat-of-aux-def index-not-eq-zero*)

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

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

lemma [*code*]:
div-mod-index n m = (*if m* = *0 then* (*0*, *n*) *else* (*n div m*, *n mod m*))
unfolding *div-mod-index-def* **by** *auto*

lemma [*code*]:
n div m = *fst* (*div-mod-index n m*)
unfolding *div-mod-index-def* **by** *simp*

lemma [*code*]:
n mod m = snd (div-mod-index n m)
unfolding *div-mod-index-def* **by** *simp*

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

14.4 ML interface

ML ⟪
structure Index =
struct

fun mk k = HOLogic.mk-number @{typ index} k;

end;
⟫

14.5 Code generator setup

Implementation of indices by bounded integers

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

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

setup ⟪
fold (Numeral.add-code @{const-name number-index-inst.number-of-index}
false false) [SML, OCaml, Haskell]
⟫

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

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

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

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

(*Haskell* **infixl** 7 ∗)

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

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

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

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

Evaluation

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

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

end

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

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

15.1 List constructors over the datatype universe

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

lemma *CONS-not-NIL* [*iff*]: *CONS M N* ≠ *NIL*
and *NIL-not-CONS* [*iff*]: *NIL* ≠ *CONS M N*
and *CONS-inject* [*iff*]: (*CONS K M*) = (*CONS L N*) = (*K* = *L* ∧ *M* = *N*)
by (*simp-all add*: *NIL-def CONS-def*)

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

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

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

lemma *List-case-NIL* [*simp*]: *List-case c h NIL* = *c*
and *List-case-CONS* [*simp*]: *List-case c h* (*CONS M N*) = *h M N*
by (*simp-all add*: *List-case-def NIL-def CONS-def*)

15.2 Corecursive lists

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

lemma *LList-mono*:
assumes *subset*: $A \subseteq B$
shows *LList A* $\subseteq$ *LList B*
— This justifies using *LList* in other recursive type definitions.
proof
fix *x*
assume $x \in LList\ A$
then show $x \in LList\ B$
proof *coinduct*
case *LList*
then show *?case* **using** *subset*
by *cases blast+*
qed
qed

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

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

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

lemma *LList-corec*:
  LList-corec a f =
    (*case f a of None* ⇒ *NIL* | *Some* (*z*, *w*) ⇒ *CONS z* (*LList-corec w f*))
  (**is** *?lhs* = *?rhs*)
proof
  show *?lhs* ⊆ *?rhs*
    apply (*unfold LList-corec-def*)
    apply (*rule UN-least*)
    apply (*case-tac k*)
     apply (*simp-all* (*no-asm-simp*) *split*: *option.splits*)
   apply (*rule allI impI subset-refl* [*THEN CONS-mono*] *UNIV-I* [*THEN UN-upper*])+
    done
  show *?rhs* ⊆ *?lhs*
    apply (*simp add*: *LList-corec-def split*: *option.splits*)
    apply (*simp add*: *CONS-UN1*)
    apply *safe*
     apply (*rule-tac a* = *Suc ?k* **in** *UN-I*, *simp*, *simp*)+
    done
qed

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

15.3 Abstract type definition

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

qed

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

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

lemma *llistI*: *x* ∈ *LList* (*range Datatype.Leaf*) ⟹ *x* ∈ *llist*
by (*simp add*: *llist-def*)

lemma *llistD*: *x* ∈ *llist* ⟹ *x* ∈ *LList* (*range Datatype.Leaf*)
by (*simp add*: *llist-def*)

lemma *Rep-llist-UNIV*: *Rep-llist x* ∈ *LList UNIV*
proof −
have *Rep-llist x* ∈ *llist* **by** (*rule Rep-llist*)
then have *Rep-llist x* ∈ *LList* (*range Datatype.Leaf*)
by (*simp add*: *llist-def*)
also have ... ⊆ *LList UNIV* **by** (*rule LList-mono*) *simp*
finally show *?thesis* **.**
qed

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

code-datatype *LNil LCons*

lemma *LCons-not-LNil* [*iff*]: *LCons x xs* ≠ *LNil*
apply (*simp add*: *LNil-def LCons-def*)
apply (*subst Abs-llist-inject*)
apply (*auto intro*: *NIL-type CONS-type Rep-llist*)
done

lemma *LNil-not-LCons* [*iff*]: *LNil* ≠ *LCons x xs*
by (*rule LCons-not-LNil* [*symmetric*])

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

lemma *Rep-llist-LNil*: *Rep-llist LNil* = *NIL*
by (*simp add*: *LNil-def add*: *Abs-llist-inverse NIL-type*)

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

by (*simp add*: *LCons-def Abs-llist-inverse CONS-type Rep-llist*)

lemma *llist-cases* [*cases type*: *llist*]:
obtains
(*LNil*) $l = LNil$
| (*LCons*) $x\ l'$ **where** $l = LCons\ x\ l'$
proof (*cases l*)
case (*Abs-llist L*)
from ‹$L \in llist$› **have** $L \in LList\ (range\ Datatype.Leaf)$ **by** (*rule llistD*)
then show *?thesis*
proof *cases*
case *NIL*
with *Abs-llist* **have** $l = LNil$ **by** (*simp add*: *LNil-def*)
with *LNil* **show** *?thesis* **.**
next
case (*CONS a K*)
then have $K \in llist$ **by** (*blast intro*: *llistI*)
then obtain l' **where** $K = Rep\text{-}llist\ l'$ **by** *cases*
with *CONS* **and** *Abs-llist* **obtain** x **where** $l = LCons\ x\ l'$
by (*auto simp add*: *LCons-def Abs-llist-inject*)
with *LCons* **show** *?thesis* **.**
qed
qed

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

syntax
LNil :: *logic*
LCons :: *logic*
translations
case p of LNil $\Rightarrow$ *a* | *LCons x l* $\Rightarrow$ *b* $\rightleftharpoons$ *CONST llist-case a* ($\lambda x\ l.$ *b*) *p*

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

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

lemma *llist-case-cert*:
assumes *CASE* $\equiv$ *llist-case c d*
shows (*CASE LNil* $\equiv$ *c*) &&& (*CASE* (*LCons M N*) $\equiv$ *d M N*)
using *assms* **by** *simp-all*

setup ⟪

```
  Code.add-case @{thm llist-case-cert}
⟫

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

lemma LList-corec-type2:
  LList-corec a
    (λz. case f z of None ⇒ None
      | Some (v, w) ⇒ Some (Datatype.Leaf v, w)) ∈ llist
  (is ?corec a ∈ -)
proof (unfold llist-def)
  let LList-corec a ?g = ?corec a
  have ∃x. ?corec a = ?corec x by blast
  then show ?corec a ∈ LList (range Datatype.Leaf)
  proof coinduct
    case (LList L)
    then obtain x where L: L = ?corec x by blast
    show ?case
    proof (cases f x)
      case None
      then have ?corec x = NIL
        by (simp add: LList-corec)
      with L have ?NIL by simp
      then show ?thesis ..
    next
      case (Some p)
      then have ?corec x =
          CONS (Datatype.Leaf (fst p)) (?corec (snd p))
        by (simp add: LList-corec split: prod.split)
      with L have ?CONS by auto
      then show ?thesis ..
    qed
  qed
qed

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

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

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

15.4 Equality as greatest fixed-point – the bisimulation principle

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

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

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

lemma Domain-EqLList: Domain (EqLList (Id-on A)) ⊆ LList A
  apply (rule subsetI)
  apply (erule LList.coinduct)
  apply (subst (asm) EqLList-unfold)
  apply (auto simp add: NIL-def CONS-def)
```

done

lemma *EqLList-Id-on*: *EqLList* (*Id-on A*) = *Id-on* (*LList A*)
(**is** *?lhs* = *?rhs*)
proof
show *?lhs* ⊆ *?rhs*
apply (*rule subsetI*)
apply (*rule-tac p* = *x* **in** *PairE*)
apply *clarify*
apply (*rule Id-on-eqI*)
apply (*rule EqLList-implies-ntrunc-equality* [*THEN ntrunc-equality*], *assumption*)
apply (*erule DomainI* [*THEN Domain-EqLList* [*THEN subsetD*]])
done
{
fix *M N* **assume** (*M*, *N*) ∈ *Id-on* (*LList A*)
then have (*M*, *N*) ∈ *EqLList* (*Id-on A*)
proof *coinduct*
case (*EqLList M N*)
then obtain *L* **where** *L*: *L* ∈ *LList A* **and** *MN*: *M* = *L N* = *L* **by** *blast*
from *L* **show** *?case*
proof *cases*
case *NIL* **with** *MN* **have** *?EqNIL* **by** *simp*
then show *?thesis* **..**
next
case *CONS* **with** *MN* **have** *?EqCONS* **by** (*simp add*: *Id-onI*)
then show *?thesis* **..**
qed
qed
}
then show *?rhs* ⊆ *?lhs* **by** *auto*
qed

lemma *EqLList-Id-on-iff* [*iff*]: (*p* ∈ *EqLList* (*Id-on A*)) = (*p* ∈ *Id-on* (*LList A*))
by (*simp only*: *EqLList-Id-on*)

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

lemma *LList-equalityI*
[*consumes 1*, *case-names EqLList*, *case-conclusion EqLList EqNIL EqCONS*]:
assumes *r*: (*M*, *N*) ∈ *r*
and *step*: ⋀*M N*. (*M*, *N*) ∈ *r* ⟹
M = *NIL* ∧ *N* = *NIL* ∨
(∃ *a b M′ N′*.
M = *CONS a M′* ∧ *N* = *CONS b N′* ∧ (*a*, *b*) ∈ *Id-on A* ∧
((*M′*, *N′*) ∈ *r* ∨ (*M′*, *N′*) ∈ *EqLList* (*Id-on A*)))
shows *M* = *N*
proof −
from *r* **have** (*M*, *N*) ∈ *EqLList* (*Id-on A*)

```
  proof coinduct
    case EqLLList
    then show ?case by (rule step)
  qed
  then show ?thesis by auto
qed

lemma LList-fun-equalityI
  [consumes 1, case-names NIL-type NIL CONS, case-conclusion CONS EqNIL
EqCONS]:
  assumes M: M ∈ LList A
    and fun-NIL: g NIL ∈ LList A  f NIL = g NIL
    and fun-CONS: ⋀x l. x ∈ A ⟹ l ∈ LList A ⟹
          (f (CONS x l), g (CONS x l)) = (NIL, NIL) ∨
          (∃ M N a b.
            (f (CONS x l), g (CONS x l)) = (CONS a M, CONS b N) ∧
              (a, b) ∈ Id-on A ∧
              (M, N) ∈ {(f u, g u) | u. u ∈ LList A} ∪ Id-on (LList A))
      (is ⋀x l. - ⟹ - ⟹ ?fun-CONS x l)
  shows f M = g M
proof −
  let ?bisim = {(f L, g L) | L. L ∈ LList A}
  have (f M, g M) ∈ ?bisim using M by blast
  then show ?thesis
  proof (coinduct taking: A rule: LList-equalityI)
    case (EqLLList M N)
    then obtain L where MN: M = f L N = g L and L: L ∈ LList A by blast
    from L show ?case
    proof (cases L)
      case NIL
      with fun-NIL and MN have (M, N) ∈ Id-on (LList A) by auto
      then have (M, N) ∈ EqLLList (Id-on A) ..
      then show ?thesis by cases simp-all
    next
      case (CONS a K)
      from fun-CONS and ‹a ∈ A› ‹K ∈ LList A›
      have ?fun-CONS a K (is ?NIL ∨ ?CONS) .
      then show ?thesis
      proof
        assume ?NIL
        with MN CONS have (M, N) ∈ Id-on (LList A) by auto
        then have (M, N) ∈ EqLLList (Id-on A) ..
        then show ?thesis by cases simp-all
      next
        assume ?CONS
        with CONS obtain a b M′ N′ where
          fg: (f L, g L) = (CONS a M′, CONS b N′)
          and ab: (a, b) ∈ Id-on A
          and M′N′: (M′, N′) ∈ ?bisim ∪ Id-on (LList A)
```

```
        by blast
      from M'N' show ?thesis
      proof
        assume (M', N') ∈ ?bisim
        with MN fg ab show ?thesis by simp
      next
        assume (M', N') ∈ Id-on (LList A)
        then have (M', N') ∈ EqLList (Id-on A) ..
        with MN fg ab show ?thesis by simp
      qed
    qed
  qed
qed
```

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

```
lemma equals-LList-corec:
  assumes h: ⋀x. h x =
    (case f x of None ⇒ NIL | Some (z, w) ⇒ CONS z (h w))
  shows h x = (λx. LList-corec x f) x
proof −
  def h' ≡ λx. LList-corec x f
  then have h': ⋀x. h' x =
      (case f x of None ⇒ NIL | Some (z, w) ⇒ CONS z (h' w))
    unfolding h'-def by (simp add: LList-corec)
  have (h x, h' x) ∈ {(h u, h' u) | u. True} by blast
  then show h x = h' x
  proof (coinduct taking: UNIV rule: LList-equalityI)
    case (EqLList M N)
    then obtain x where MN: M = h x N = h' x by blast
    show ?case
    proof (cases f x)
      case None
      with h h' MN have ?EqNIL by simp
      then show ?thesis ..
    next
      case (Some p)
      with h h' MN have M = CONS (fst p) (h (snd p))
        and N = CONS (fst p) (h' (snd p))
        by (simp-all split: prod.split)
      then have ?EqCONS by (auto iff: Id-on-iff)
      then show ?thesis ..
    qed
  qed
qed


lemma llist-equalityI
  [consumes 1, case-names Eqllist, case-conclusion Eqllist EqLNil EqLCons]:
```

```
  assumes r: (l1, l2) ∈ r
    and step: ⋀q. q ∈ r ⟹
      q = (LNil, LNil) ∨
       (∃ l1 l2 a b.
         q = (LCons a l1, LCons b l2) ∧ a = b ∧
          ((l1, l2) ∈ r ∨ l1 = l2))
      (is ⋀q. - ⟹ ?EqLNil q ∨ ?EqLCons q)
  shows l1 = l2
proof −
  def M ≡ Rep-llist l1 and N ≡ Rep-llist l2
  with r have (M, N) ∈ {(Rep-llist l1, Rep-llist l2) | l1 l2. (l1, l2) ∈ r}
    by blast
  then have M = N
  proof (coinduct taking: UNIV rule: LList-equalityI)
    case (EqLList M N)
    then obtain l1 l2 where
       MN: M = Rep-llist l1 N = Rep-llist l2 and r: (l1, l2) ∈ r
      by auto
    from step [OF r] show ?case
    proof
      assume ?EqLNil (l1, l2)
      with MN have ?EqNIL by (simp add: Rep-llist-LNil)
      then show ?thesis ..
    next
      assume ?EqLCons (l1, l2)
      with MN have ?EqCONS
        by (force simp add: Rep-llist-LCons EqLList-Id-on intro: Rep-llist-UNIV)
      then show ?thesis ..
    qed
  qed
  then show ?thesis by (simp add: M-def N-def Rep-llist-inject)
qed

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

show *?case*
proof (*cases l*)
case *LNil*
with *fun-LNil* **and** *q* **have** *q* = (*g LNil*, *g LNil*) **by** *simp*
then show *?thesis* **by** (*cases g LNil*) *simp-all*
next
case (*LCons x l'*)
with ‹*?fun-LCons x l'*› *q LCons* **show** *?thesis* **by** *blast*
qed
qed
qed

15.5 Derived operations – both on the set and abstract type

15.5.1 *Lconst*

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

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

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

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

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

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

done

15.5.2 *Lmap* and *lmap*

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

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

lemma *Lmap-type*:
assumes *M*: *M* ∈ *LList A*
and *f*: ⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B*
shows *Lmap f M* ∈ *LList B*
proof −
from *M* **have** *Lmap f M* ∈ {*Lmap f N* | *N*. *N* ∈ *LList A*} **by** *blast*
then show *?thesis*
proof *coinduct*
case (*LList L*)
then obtain *N* **where** *L*: *L* = *Lmap f N* **and** *N*: *N* ∈ *LList A* **by** *blast*
from *N* **show** *?case*
proof *cases*
case *NIL*
with *L* **have** *?NIL* **by** *simp*
then show *?thesis* **..**
next
case (*CONS K a*)
with *f L* **have** *?CONS* **by** *auto*
then show *?thesis* **..**
qed
qed
qed

lemma *Lmap-compose*:
assumes *M*: *M* ∈ *LList A*
shows *Lmap* (*f o g*) *M* = *Lmap f* (*Lmap g M*) (**is** *?lhs M* = *?rhs M*)
proof −
have (*?lhs M*, *?rhs M*) ∈ {(*?lhs N*, *?rhs N*) | *N*. *N* ∈ *LList A*}
using *M* **by** *blast*
then show *?thesis*
proof (*coinduct taking*: *range* (λ*N*. *N*) *rule*: *LList-equalityI*)
case (*EqLList L M*)
then obtain *N* **where** *LM*: *L* = *?lhs N M* = *?rhs N* **and** *N*: *N* ∈ *LList A*

by *blast*
from *N* **show** *?case*
proof *cases*
case *NIL*
with *LM* **have** *?EqNIL* **by** *simp*
then show *?thesis* **..**
next
case *CONS*
with *LM* **have** *?EqCONS* **by** *auto*
then show *?thesis* **..**
qed
qed
qed

lemma *Lmap-ident*:
assumes *M*: $M \in LList\ A$
shows $Lmap\ (\lambda x.\ x)\ M = M$ (**is** *?lmap M* = -)
proof −
have (*?lmap M*, *M*) ∈ {(*?lmap N*, *N*) | *N*. $N \in LList\ A$} **using** *M* **by** *blast*
then show *?thesis*
proof (*coinduct taking*: *range* (λ*N*. *N*) *rule*: *LList-equalityI*)
case (*EqLList L M*)
then obtain *N* **where** *LM*: $L = ?lmap\ N\ M = N$ **and** *N*: $N \in LList\ A$ **by** *blast*
from *N* **show** *?case*
proof *cases*
case *NIL*
with *LM* **have** *?EqNIL* **by** *simp*
then show *?thesis* **..**
next
case *CONS*
with *LM* **have** *?EqCONS* **by** *auto*
then show *?thesis* **..**
qed
qed
qed

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

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

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

15.5.3 *Lappend*

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

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

lemma *Lappend-NIL* [*simp*]: *M ∈ LList A ⟹ Lappend NIL M = M*
by (*erule LList-fun-equalityI*) *auto*

lemma *Lappend-NIL2*: *M ∈ LList A ⟹ Lappend M NIL = M*
by (*erule LList-fun-equalityI*) *auto*

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

```
      then show ?thesis ..
    qed
  next
    case CONS
    with L N have ?CONS by auto
    then show ?thesis ..
  qed
 qed
qed

lemma lappend-LNil-LNil [simp]: lappend LNil LNil = LNil
 and lappend-LNil-LCons [simp]: lappend LNil (LCons l l') = LCons l (lappend
LNil l')
 and lappend-LCons [simp]: lappend (LCons l l') m = LCons l (lappend l' m)
 by (simp-all add: lappend-def llist-corec)

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

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

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

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

15.6 iterates

llist-fun-equalityI cannot be used here!

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

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

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

```
    by blast
  also have iterates f (f x) = LCons (f x) (iterates f (f (f x)))
    by (subst iterates) rule
  also have iterates f x = LCons x (iterates f (f x))
    by (subst iterates) rule
  finally have ?EqLCons by auto
  then show ?case ..
 qed
qed

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

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

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


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

```
    also have ?q2 = LCons ((f ^ n) u) ((lmap f ^ Suc n) (iterates f u))
    proof −
      have ?q2 = (lmap f ^ n) (LCons u (iterates f (f u)))
        by (subst iterates) rule
      also have ... = LCons ((f ^ n) u) ((lmap f ^ n) (iterates f (f u)))
        by (rule funpow-lmap)
      also have (lmap f ^ n) (iterates f (f u)) = (lmap f ^ Suc n) (iterates f u)
        by (simp add: lmap-iterates funpow-swap1)
      finally show ?thesis .
    qed
    finally have ?EqLCons by (auto simp del: funpow.simps)
    then show ?case ..
  qed
qed

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

end
```

16 Commutative-Ring: Proving equalities in commutative rings

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

Syntax of multivariate polynomials (pol) and polynomial expressions.

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

datatype 'a polex =
    Pol 'a pol
```

```
| Add 'a polex 'a polex
| Sub 'a polex 'a polex
| Mul 'a polex 'a polex
| Pow 'a polex nat
| Neg 'a polex
```

Interpretation functions for the shadow syntax.

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

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

Create polynomial normalized polynomials given normalized inputs.

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

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

Defining the basic ring operations on normalized polynomials

```
function
  add :: 'a::{comm-ring,recpower} pol ⇒ 'a pol ⇒ 'a pol (infixl ⊕ 65)
where
    Pc a ⊕ Pc b = Pc (a + b)
  | Pc c ⊕ Pinj i P = Pinj i (P ⊕ Pc c)
  | Pinj i P ⊕ Pc c = Pinj i (P ⊕ Pc c)
  | Pc c ⊕ PX P i Q = PX P i (Q ⊕ Pc c)
  | PX P i Q ⊕ Pc c = PX P i (Q ⊕ Pc c)
```

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

function
  mul :: 'a::{comm-ring,recpower} pol ⇒ 'a pol ⇒ 'a pol (infixl ⊗ 70)
where
    Pc a ⊗ Pc b = Pc (a * b)
  | Pc c ⊗ Pinj i P =
      (if c = 0 then Pc 0 else mkPinj i (P ⊗ Pc c))
  | Pinj i P ⊗ Pc c =
      (if c = 0 then Pc 0 else mkPinj i (P ⊗ Pc c))
  | Pc c ⊗ PX P i Q =
      (if c = 0 then Pc 0 else mkPX (P ⊗ Pc c) i (Q ⊗ Pc c))
  | PX P i Q ⊗ Pc c =
      (if c = 0 then Pc 0 else mkPX (P ⊗ Pc c) i (Q ⊗ Pc c))
  | Pinj x P ⊗ Pinj y Q =
      (if x = y then mkPinj x (P ⊗ Q) else
        (if x > y then mkPinj y (Pinj (x−y) P ⊗ Q)
          else mkPinj x (Pinj (y − x) Q ⊗ P)))
  | Pinj x P ⊗ PX Q y R =
      (if x = 0 then P ⊗ PX Q y R else
        (if x = 1 then mkPX (Pinj x P ⊗ Q) y (R ⊗ P)
          else mkPX (Pinj x P ⊗ Q) y (R ⊗ Pinj (x − 1) P)))
  | PX P x R ⊗ Pinj y Q =
      (if y = 0 then PX P x R ⊗ Q else
        (if y = 1 then mkPX (Pinj y Q ⊗ P) x (R ⊗ Q)
          else mkPX (Pinj y Q ⊗ P) x (R ⊗ Pinj (y − 1) Q)))
  | PX P1 x P2 ⊗ PX Q1 y Q2 =
      mkPX (P1 ⊗ Q1) (x + y) (P2 ⊗ Q2) ⊕
        (mkPX (P1 ⊗ mkPinj 1 Q2) x (Pc 0) ⊕
          (mkPX (Q1 ⊗ mkPinj 1 P2) y (Pc 0)))
by pat-completeness auto
termination by (relation measure (λ(x, y). size x + size y))
```

(*auto simp add*: *mkPinj-def split*: *pol.split*)

Negation

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

Substraction

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

Square for Fast Exponentation

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

Fast Exponentation

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

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

Normalization of polynomial expressions

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

mkPinj preserve semantics

lemma *mkPinj-ci*: *Ipol l* (*mkPinj a B*) = *Ipol l* (*Pinj a B*)
by (*induct B*) (*auto simp add*: *mkPinj-def algebra-simps*)

mkPX preserves semantics

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

Correctness theorems for the implemented operations

Negation

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

Addition

lemma *add-ci*: *Ipol l* (*P* ⊕ *Q*) = *Ipol l P* + *Ipol l Q*
proof (*induct P Q arbitrary*: *l rule*: *add.induct*)
case (*6 x P y Q*)
show *?case*
proof (*rule linorder-cases*)
assume *x* < *y*
with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci algebra-simps*)
next
assume *x* = *y*
with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci*)
next
assume *x* > *y*
with *6* **show** *?case* **by** (*simp add*: *mkPinj-ci algebra-simps*)
qed
next
case (*7 x P Q y R*)
have *x* = *0* ∨ *x* = *1* ∨ *x* > *1* **by** *arith*
moreover
{ assume *x* = *0* **with** *7* **have** *?case* **by** *simp* **}**
moreover
{ assume *x* = *1* **with** *7* **have** *?case* **by** (*simp add*: *algebra-simps*) **}**
moreover
{ assume *x* > *1* **from** *7* **have** *?case* **by** (*cases x*) *simp-all* **}**
ultimately show *?case* **by** *blast*
next
case (*8 P x R y Q*)
have *y* = *0* ∨ *y* = *1* ∨ *y* > *1* **by** *arith*
moreover
{ assume *y* = *0* **with** *8* **have** *?case* **by** *simp* **}**
moreover
{ assume *y* = *1* **with** *8* **have** *?case* **by** *simp* **}**
moreover
{ assume *y* > *1* **with** *8* **have** *?case* **by** *simp* **}**
ultimately show *?case* **by** *blast*

next
  case (*9 P1 x P2 Q1 y Q2*)
  show *?case*
  proof (*rule linorder-cases*)
    assume *a*: $x < y$ **hence** *EX d*. $d + x = y$ **by** *arith*
    with *9 a* **show** *?case* **by** (*auto simp add*: *mkPX-ci power-add algebra-simps*)
  next
    assume *a*: $y < x$ **hence** *EX d*. $d + y = x$ **by** *arith*
    with *9 a* **show** *?case* **by** (*auto simp add*: *power-add mkPX-ci algebra-simps*)
  next
    assume $x = y$
    with *9* **show** *?case* **by** (*simp add*: *mkPX-ci algebra-simps*)
  qed
qed (*auto simp add*: *algebra-simps*)

Multiplication

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

Substraction

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

Square

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

Power

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

lemma *pow-ci*: *Ipol ls* (*pow n P*) = *Ipol ls P* ^ *n*
proof (*induct n arbitrary*: *P rule*: *nat-less-induct*)
  case (*1 k*)
  show *?case*
  proof (*cases k*)
    case *0*
    then show *?thesis* **by** *simp*
  next
    case (*Suc l*)
    show *?thesis*
    proof *cases*
      assume *even l*
      then have *Suc l div 2* = *l div 2*
        by (*simp add*: *nat-number even-nat-plus-one-div-two*)
      moreover
      from *Suc* **have** $l < k$ **by** *simp*

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

Normalization preserves semantics

```
lemma norm-ci: Ipolex l Pe = Ipol l (norm Pe)
 by (induct Pe) (simp-all add: add-ci sub-ci mul-ci neg-ci pow-ci)
```

Reflection lemma: Key to the (incomplete) decision procedure

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


use comm-ring.ML
setup CommRing.setup

end
```

17 Continuity: Continuity and iterations (of set transformers)

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

17.1 Continuity for complete lattices

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

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

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

17.2 Chains

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

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

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

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

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

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

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

17.3 Continuity

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

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

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

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

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

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

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

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

17.4 Iteration

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

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

18 ContNotDenum: Non-denumerability of the Continuum.

theory *ContNotDenum*
imports *Complex-Main*
begin

18.1 Abstract

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

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

Outline: An elegant informal proof of this result uses Cantor's Diagonalisation argument. The proof presented here is not this one. First we formalise some properties of closed intervals, then we prove the Nested Interval Property. This property relies on the completeness of the Real numbers and is the foundation for our argument. Informally it states that an intersection of countable closed intervals (where each successive interval is a subset of the last) is non-empty. We then assume a surjective function f:$\mathbb{N}\Rightarrow\mathbb{R}$ exists and find a real x such that x is not in the range of f by generating a sequence of closed intervals then using the NIP.

18.2 Closed Intervals

This section formalises some properties of closed intervals.

18.2.1 Definition

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

18.2.2 Properties

lemma *closed-int-subset*:
assumes *xy*: *x1* ≥ *x0 y1* ≤ *y0*
shows *closed-int x1 y1* ⊆ *closed-int x0 y0*
proof −
{
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*: $a \leq b$
shows $a \in$ *closed-int a b* $\wedge$ $(\forall x \in$ *closed-int a b*. $a \leq x)$
proof
from *a* **have** $a \in \{x.\ a \leq x \wedge x \leq b\}$ **by** *simp*
thus $a \in$ *closed-int a b* **by** (*unfold closed-int-def*)
next
have $\forall x \in \{x.\ a \leq x \wedge x \leq b\}.\ a \leq x$ **by** *simp*
thus $\forall x \in$ *closed-int a b*. $a \leq x$ **by** (*unfold closed-int-def*)
qed

lemma *closed-int-most*:
assumes *a*: $a \leq b$
shows $b \in$ *closed-int a b* $\wedge$ $(\forall x \in$ *closed-int a b*. $x \leq b)$
proof
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

18.3 Nested Interval Property

theorem *NIP*:
fixes f::*nat* $\Rightarrow$ *real set*
assumes *subset*: $\forall n.\ f\ (Suc\ n) \subseteq f\ n$

```
  and closed: ∀ n. ∃ a b. f n = closed-int a b ∧ a ≤ b
  shows (⋂ n. f n) ≠ {}
proof −
  let ?g = λn. (SOME c. c∈(f n) ∧ (∀ x∈(f n). c ≤ x))
  have ne: ∀ n. ∃ x. x∈(f n)
  proof
    fix n
    from closed have ∃ a b. f n = closed-int a b ∧ a ≤ b by simp
    then obtain a and b where fn: f n = closed-int a b ∧ a ≤ b by auto
    hence a ≤ b ..
    with closed-not-empty have ∃ x. x∈closed-int a b by simp
    with fn show ∃ x. x∈(f n) by simp
  qed

  have gdef: ∀ n. (?g n)∈(f n) ∧ (∀ x∈(f n). (?g n)≤x)
  proof
    fix n
    from closed have ∃ a b. f n = closed-int a b ∧ a ≤ b ..
    then obtain a and b where ff: f n = closed-int a b and a ≤ b by auto
    hence a ≤ b by simp
    hence a∈closed-int a b ∧ (∀ x∈closed-int a b. a ≤ x) by (rule closed-int-least)
    with ff have a∈(f n) ∧ (∀ x∈(f n). a ≤ x) by simp
    hence ∃ c. c∈(f n) ∧ (∀ x∈(f n). c ≤ x) ..
    thus (?g n)∈(f n) ∧ (∀ x∈(f n). (?g n)≤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 ‘ ℕ by simp
  ultimately have ∃ x. x∈A
  proof −
    have (0::nat) ∈ ℕ by simp
    moreover have ?g 0 = ?g 0 by simp
    ultimately have ?g 0 ∈ ?g ‘ ℕ by (rule rev-image-eqI)
    with Adef have ?g 0 ∈ A by simp
    thus ?thesis ..
  qed

  — Now show that A is bounded above ...
  moreover have ∃ y. isUb (UNIV::real set) A y
  proof −
    {
      fix n
      from ne have ex: ∃ x. x∈(f n) ..
      from gdef have (?g n)∈(f n) ∧ (∀ x∈(f n). (?g n)≤x) by simp
      moreover
      from closed have ∃ a b. f n = closed-int a b ∧ a ≤ b ..
      then obtain a and b where f n = closed-int a b ∧ a ≤ b by auto
      hence b∈(f n) ∧ (∀ x∈(f n). x ≤ b) using closed-int-most by blast
      ultimately have ∀ x∈(f n). (?g n) ≤ b by simp
```

```
    with ex have (?g n) ≤ b by auto
    hence ∃ b. (?g n) ≤ b by auto
  }
  hence aux: ∀ n. ∃ b. (?g n) ≤ b ..

  have fs: ∀ n::nat. f n ⊆ f 0
  proof (rule allI, induct-tac n)
    show f 0 ⊆ f 0 by simp
  next
    fix n
    assume f n ⊆ f 0
    moreover from subset have f (Suc n) ⊆ f n ..
    ultimately show f (Suc n) ⊆ f 0 by simp
  qed
  have ∀ n. (?g n)∈(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
```

18.4 Generating the intervals

18.4.1 Existence of non-singleton closed intervals

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

```
lemma closed-subset-ex:
 fixes c::real
 assumes alb: a < b
 shows
  ∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int a b ∧ c ∉ (closed-int ka kb)
proof −
 {
  assume clb: c < b
  {
   assume cla: c < a
   from alb cla clb have c ∉ closed-int a b by (unfold closed-int-def, auto)
   with alb have
    a < b ∧ closed-int a b ⊆ closed-int a b ∧ c ∉ closed-int a b
    by auto
   hence
    ∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int a b ∧ c ∉ (closed-int ka kb)
    by auto
  }
  moreover
  {
   assume ncla: ¬(c < a)
   with clb have cdef: a ≤ c ∧ 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∉(closed-int ka b) by (unfold closed-int-def, auto)
   moreover from cdef kagc have ka ≥ a by simp
   hence closed-int ka b ⊆ closed-int a b by (unfold closed-int-def, auto)
```

```
      ultimately have
        ka < b  ∧ closed-int ka b ⊆ closed-int a b ∧ c ∉ closed-int ka b
        using kalb by auto
      hence
       ∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int a b ∧ c ∉ (closed-int ka kb)
        by auto

    }
    ultimately have
     ∃ ka kb. ka < kb ∧ closed-int ka kb ⊆ closed-int a b ∧ c ∉ (closed-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
```

18.5 newInt: Interval generation

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

18.5.1 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*]

18.5.2 Properties

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

lemma *newInt-ex*:
 $\exists a\ b.\ a < b \wedge$
  newInt (*Suc n*) *f* = *closed-int a b* $\wedge$
  newInt (*Suc n*) *f* $\subseteq$ *newInt n f* $\wedge$
  f (*Suc n*) $\notin$ *newInt* (*Suc n*) *f*
proof (*induct n*)
 case *0*

 let *?e* = *SOME e.* $\exists$ *e1 e2.*
  e1 < *e2* $\wedge$
  e = *closed-int e1 e2* $\wedge$
  e $\subseteq$ *closed-int* (*f 0* + *1*) (*f 0* + *2*) $\wedge$
  f (*Suc 0*) $\notin$ *e*

 have *newInt* (*Suc 0*) *f* = *?e* **by** *auto*
 moreover
 have *f 0* + *1* < *f 0* + *2* **by** *simp*
 with *closed-subset-ex* **have**
  $\exists$ *ka kb. ka* < *kb* $\wedge$ *closed-int ka kb* $\subseteq$ *closed-int* (*f 0* + *1*) (*f 0* + *2*) $\wedge$
   f (*Suc 0*) $\notin$ (*closed-int ka kb*) **.**
 hence
  $\exists$ *e.* $\exists$ *ka kb. ka* < *kb* $\wedge$ *e* = *closed-int ka kb* $\wedge$
   e $\subseteq$ *closed-int* (*f 0* + *1*) (*f 0* + *2*) $\wedge$ *f* (*Suc 0*) $\notin$ *e* **by** *simp*
 hence
  $\exists$ *ka kb. ka* < *kb* $\wedge$ *?e* = *closed-int ka kb* $\wedge$
   ?e $\subseteq$ *closed-int* (*f 0* + *1*) (*f 0* + *2*) $\wedge$ *f* (*Suc 0*) $\notin$ *?e*
  by (*rule someI-ex*)
 ultimately have $\exists$ *e1 e2. e1* < *e2* $\wedge$
  newInt (*Suc 0*) *f* = *closed-int e1 e2* $\wedge$
  newInt (*Suc 0*) *f* $\subseteq$ *closed-int* (*f 0* + *1*) (*f 0* + *2*) $\wedge$
  f (*Suc 0*) $\notin$ *newInt* (*Suc 0*) *f* **by** *simp*
 thus
  $\exists$ *a b. a* < *b* $\wedge$ *newInt* (*Suc 0*) *f* = *closed-int a b* $\wedge$
   newInt (*Suc 0*) *f* $\subseteq$ *newInt 0 f* $\wedge$ *f* (*Suc 0*) $\notin$ *newInt* (*Suc 0*) *f*
  by *simp*
next
 case (*Suc n*)
 hence $\exists$ *a b.*
  a < *b* $\wedge$
  newInt (*Suc n*) *f* = *closed-int a b* $\wedge$
  newInt (*Suc n*) *f* $\subseteq$ *newInt n f* $\wedge$

```
  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

18.6 Final Theorem

theorem *real-non-denum*:
shows ¬ (∃*f*::*nat*⇒*real*. *surj f*)
proof — by contradiction
assume ∃*f*::*nat*⇒*real*. *surj f*
then obtain *f*::*nat*⇒*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 ∃*x*. *x* ∈ (⋂ *n*. *newInt n f*) **using** *newInt-notempty* **by** *blast*
moreover have ∀ *n*. *f n* ∉ (⋂ *n*. *newInt n f*) **by** (*rule newInt-inter*)
ultimately obtain *x* **where** *x* ∈ (⋂ *n*. *newInt n f*) **and** ∀ *n*. *f n* ≠ *x* **by** *blast*
moreover from *rangeF* **have** *x* ∈ *range f* **by** *simp*
ultimately show *False* **by** *blast*
qed

end

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

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

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

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

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

lemma *dvd2-a-x-suc-a*: *2 dvd a* ∗ (*Suc a*)
proof (*cases 2 dvd a*)
case *True*
then show *?thesis* **by** (*rule dvd-mult2*)
next
case *False*

```
  then have Suc (a mod 2) = 2 by (simp add: dvd-eq-mod-eq-0)
  then have Suc a mod 2 = 0 by (simp add: mod-Suc)
  then have 2 dvd Suc a by (simp only:dvd-eq-mod-eq-0)
  then show ?thesis by (rule dvd-mult)
qed

lemma
  assumes eq: nat2-to-nat (u,v) = nat2-to-nat (x,y)
  shows nat2-to-nat-help: u+v ≤ x+y
proof (rule classical)
  assume ¬ ?thesis
  then have contrapos: x+y < u+v
    by simp
  have nat2-to-nat (x,y) < (x+y) * Suc (x+y) div 2 + Suc (x + y)
    by (unfold nat2-to-nat-def) (simp add: Let-def)
  also have ... = (x+y)*Suc(x+y) div 2 + 2 * Suc(x+y) div 2
    by (simp only: div-mult-self1-is-m)
  also have ... = (x+y)*Suc(x+y) div 2 + 2 * Suc(x+y) div 2
    + ((x+y)*Suc(x+y) mod 2 + 2 * Suc(x+y) mod 2) div 2
  proof −
    have 2 dvd (x+y)*Suc(x+y)
      by (rule dvd2-a-x-suc-a)
    then have (x+y)*Suc(x+y) mod 2 = 0
      by (simp only: dvd-eq-mod-eq-0)
    also
    have 2 * Suc(x+y) mod 2 = 0
      by (rule mod-mult-self1-is-0)
    ultimately have
      ((x+y)*Suc(x+y) mod 2 + 2 * Suc(x+y) mod 2) div 2 = 0
      by simp
    then show ?thesis
      by simp
  qed
  also have ... = ((x+y)*Suc(x+y) + 2*Suc(x+y)) div 2
    by (rule div-add1-eq [symmetric])
  also have ... = ((x+y+2)*Suc(x+y)) div 2
    by (simp only: add-mult-distrib [symmetric])
  also from contrapos have ... ≤ ((Suc(u+v))*(u+v)) div 2
    by (simp only: mult-le-mono div-le-mono)
  also have ... ≤ nat2-to-nat (u,v)
    by (unfold nat2-to-nat-def) (simp add: Let-def)
  finally show ?thesis
    by (simp only: eq)
qed

theorem nat2-to-nat-inj: inj nat2-to-nat
proof −
  {
    fix u v x y
```

```
    assume eq1: nat2-to-nat (u,v) = nat2-to-nat (x,y)
    then have u+v ≤ x+y by (rule nat2-to-nat-help)
    also from eq1 [symmetric] have x+y ≤ u+v
      by (rule nat2-to-nat-help)
    finally have eq2: u+v = x+y .
    with eq1 have ux: u=x
      by (simp add: nat2-to-nat-def Let-def)
    with eq2 have vy: v=y by simp
    with ux have (u,v) = (x,y) by simp
  }
  then have ⋀x y. nat2-to-nat x = nat2-to-nat y ⟹ x=y by fast
  then show ?thesis unfolding inj-on-def by simp
qed

lemma nat-to-nat2-surj: surj nat-to-nat2
by (simp only: nat-to-nat2-def nat2-to-nat-inj inj-imp-surj-inv)


lemma gauss-sum-nat-upto: 2 * (∑ i≤n::nat. i) = n * (n + 1)
using gauss-sum[where 'a = nat]
by (simp add:atLeast0AtMost setsum-shift-lb-Suc0-0 numeral-2-eq-2)

lemma nat2-to-nat-surj: surj nat2-to-nat
proof (unfold surj-def)
  {
    fix z::nat
    def r ≡ Max {r. (∑ i≤r. i) ≤ z}
    def x ≡ z − (∑ i≤r. i)

    hence finite {r. (∑ i≤r. i) ≤ z}
      by (simp add: lessThan-Suc-atMost[symmetric] lessThan-Suc finite-less-ub)
    also have 0 ∈ {r. (∑ i≤r. i) ≤ z} by simp
    hence {r::nat. (∑ i≤r. i) ≤ z} ≠ {} by fast
    ultimately have a: r ∈ {r. (∑ i≤r. i) ≤ z} ∧ (∀ s ∈ {r. (∑ i≤r. i) ≤ z}. s
≤ r)
      by (simp add: r-def del:mem-Collect-eq)
    {
      assume r<x
      hence r+1≤x by simp
      hence (∑ i≤r. i)+(r+1)≤z using x-def by arith
      hence (r+1) ∈ {r. (∑ i≤r. i) ≤ z} by simp
      with a have (r+1)≤r by simp
    }
    hence b: x≤r by force

    def y ≡ r−x
    have 2∗z=2∗(∑ i≤r. i)+2∗x using x-def a by simp arith
    also have ... = r ∗ (r+1) + 2∗x  using gauss-sum-nat-upto by simp
    also have ... = (x+y)∗(x+y+1)+2∗x using y-def b by simp
```

also { have *2 dvd ((x+y)*(x+y+1))* **using** *dvd2-a-x-suc-a* **by** *simp* **}**
hence ... = *2 * nat2-to-nat(x,y)*
using *nat2-to-nat-def* **by** (*simp add*: *Let-def dvd-mult-div-cancel*)
finally have *z=nat2-to-nat (x, y)* **by** *simp*
}
thus ∀ *y*. ∃ *x*. *y* = *nat2-to-nat x* **by** *fast*
qed

19.2 A bijection between ℕ and ℤ

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

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

lemma *i2n-n2i-id*: *int-to-nat-bij* (*nat-to-int-bij n*) = *n*
by (*simp add*: *int-to-nat-bij-def nat-to-int-bij-def*) *presburger*

lemma *n2i-i2n-id*: *nat-to-int-bij*(*int-to-nat-bij i*) = *i*
proof −
have *ALL m n*::*nat*. *m>0* ⟶ *2 * m* − *Suc 0* ≠ *2 * n* **by** *presburger*
thus *?thesis*
by(*simp add*: *nat-to-int-bij-def int-to-nat-bij-def*, *simp add*:*dvd-def*)
qed

lemma *inv-nat-to-int-bij*: *inv nat-to-int-bij* = *int-to-nat-bij*
by (*simp add*: *i2n-n2i-id inv-equality n2i-i2n-id*)

lemma *inv-int-to-nat-bij*: *inv int-to-nat-bij* = *nat-to-int-bij*
by (*simp add*: *i2n-n2i-id inv-equality n2i-i2n-id*)

lemma *surj-nat-to-int-bij*: *surj nat-to-int-bij*
by (*blast intro*: *n2i-i2n-id surjI*)

lemma *surj-int-to-nat-bij*: *surj int-to-nat-bij*
by (*blast intro*: *i2n-n2i-id surjI*)

lemma *inj-nat-to-int-bij*: *inj nat-to-int-bij*
by(*simp add*:*inv-int-to-nat-bij*[*symmetric*] *surj-int-to-nat-bij surj-imp-inj-inv*)

lemma *inj-int-to-nat-bij*: *inj int-to-nat-bij*
by(*simp add*:*inv-nat-to-int-bij*[*symmetric*] *surj-nat-to-int-bij surj-imp-inj-inv*)

end

20 Countable: Encoding (almost) everything into natural numbers

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

20.1 The class of countable types

class *countable* =
assumes *ex-inj*: $\exists$ *to-nat* :: $'a \Rightarrow$ *nat*. *inj to-nat*

lemma *countable-classI*:
fixes $f :: 'a \Rightarrow nat$
assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
shows *OFCLASS*($'a$, *countable-class*)
proof (*intro-classes*, *rule exI*)
show *inj f*
by (*rule injI* [*OF assms*]) *assumption*
qed

20.2 Conversion functions

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

definition *from-nat* :: *nat* $\Rightarrow$ $'a$::*countable* **where**
from-nat = *inv* (*to-nat* :: $'a \Rightarrow$ *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* $\longleftrightarrow$ *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*)

20.3 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*]
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

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

lemma *sum-arith*: *sum n* = *n* ∗ *Suc n div 2*
by (*induct n*) *auto*

lemma *sum-mono*: *n* ≥ *m* ⟹ *sum n* ≥ *sum m*
by (*induct n m rule*: *diff-induct*) *auto*

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

lemma *inj-pair-cencode*: *inj pair-encode*
unfolding *pair-encode-def*
proof (*rule injI*, *simp only*: *split-paired-all split-conv*)
fix *a b c d*
assume *eq*: *sum* (*a* + *b*) + *a* = *sum* (*c* + *d*) + *c*
have *a* + *b* = *c* + *d* ∨ *a* + *b* ≥ *Suc* (*c* + *d*) ∨ *c* + *d* ≥ *Suc* (*a* + *b*) **by** *arith*
then
show (*a*, *b*) = (*c*, *d*)
proof (*elim disjE*)
assume *sumeq*: *a* + *b* = *c* + *d*
then have *a* = *c* **using** *eq* **by** *auto*
moreover from *sumeq this* **have** *b* = *d* **by** *auto*
ultimately show *?thesis* **by** *simp*
next
assume *a* + *b* ≥ *Suc* (*c* + *d*)
from *sum-mono*[*OF this*] *eq*
show *?thesis* **by** *auto*
next
assume *c* + *d* ≥ *Suc* (*a* + *b*)
from *sum-mono*[*OF this*] *eq*
show *?thesis* **by** *auto*

```
  qed
qed

instance ∗ :: (countable, countable) countable
by (rule countable-classI [of λ(x, y). pair-encode (to-nat x, to-nat y)])
  (auto dest: injD [OF inj-pair-cencode] injD [OF inj-to-nat])
```

Sums

```
instance +:: (countable, countable) countable
  by (rule countable-classI [of (λx. case x of Inl a ⇒ to-nat (False, to-nat a)
                                  | Inr b ⇒ to-nat (True, to-nat b))])
    (auto split:sum.splits)
```

Integers

```
lemma int-cases: (i::int) = 0 ∨ i < 0 ∨ i > 0
by presburger

lemma int-pos-neg-zero:
  obtains (zero) (z::int) = 0 sgn z = 0 abs z = 0
  | (pos) n where z = of-nat n sgn z = 1 abs z = of-nat n
  | (neg) n where z = − (of-nat n) sgn z = −1 abs z = of-nat n
apply atomize-elim
apply (insert int-cases[of z])
apply (auto simp:zsgn-def)
apply (rule-tac x=nat (−z) in exI, simp)
apply (rule-tac x=nat z in exI, simp)
done

instance int :: countable
proof (rule countable-classI [of (λi. to-nat (nat (sgn i + 1), nat (abs i)))],
    auto dest: injD [OF inj-to-nat])
  fix x y
  assume a: nat (sgn x + 1) = nat (sgn y + 1) nat (abs x) = nat (abs y)
  show x = y
  proof (cases rule: int-pos-neg-zero[of x])
    case zero
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  next
    case (pos n)
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  next
    case (neg n)
    with a show x = y by (cases rule: int-pos-neg-zero[of y]) auto
  qed
qed
```

Options

```
instance option :: (countable) countable
by (rule countable-classI[of λx. case x of None ⇒ 0
```

```
                                | Some y ⇒ Suc (to-nat y)])
(auto split:option.splits)
```

Lists

```
lemma from-nat-to-nat-map [simp]: map from-nat (map to-nat xs) = xs
  by (simp add: comp-def map-compose [symmetric])

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

instance list :: (countable) countable
proof (rule countable-classI [of list-encode])
  fix xs ys :: 'a list
  assume cenc: list-encode xs = list-encode ys
  then show xs = ys
  proof (induct xs arbitrary: ys)
    case (Nil ys)
    with cenc show ?case by (cases ys, auto)
  next
    case (Cons x xs' ys)
    thus ?case by (cases ys) auto
  qed
qed
```

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
```

20.4 The Rationals are Countably Infinite

```
definition nat-to-rat-surj :: nat ⇒ rat where
nat-to-rat-surj n = (let (a,b) = nat-to-nat2 n
                     in Fract (nat-to-int-bij a) (nat-to-int-bij b))

lemma surj-nat-to-rat-surj: surj nat-to-rat-surj
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 = inv nat-to-int-bij i; n = inv nat-to-int-bij j
              in nat-to-rat-surj(nat2-to-nat (m,n)))
      using nat2-to-nat-inj surj-f-inv-f[OF surj-nat-to-int-bij]
      by(simp add:Let-def nat-to-rat-surj-def nat-to-nat2-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
```

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

```
theory Dense-Linear-Order
imports Main
```

uses
~~/*src*/*HOL*/*Tools*/*Qelim*/*langford-data.ML*
~~/*src*/*HOL*/*Tools*/*Qelim*/*ferrante-rackoff-data.ML*
(~~/*src*/*HOL*/*Tools*/*Qelim*/*langford.ML*)
(~~/*src*/*HOL*/*Tools*/*Qelim*/*ferrante-rackoff.ML*)
begin

setup ⟪ *Langford-Data.setup #> Ferrante-Rackoff-Data.setup* ⟫

context *linorder*
begin

lemma *less-not-permute*[*noatp*]: ¬ (*x* < *y* ∧ *y* < *x*) **by** (*simp add*: *not-less linear*)

lemma *gather-simps*[*noatp*]:
shows
(∃ *x*. (∀ *y* ∈ *L*. *y* < *x*) ∧ (∀ *y* ∈ *U*. *x* < *y*) ∧ *x* < *u* ∧ *P x*) ⟷ (∃ *x*. (∀ *y* ∈ *L*. *y* < *x*) ∧ (∀ *y* ∈ (*insert u U*). *x* < *y*) ∧ *P x*)
and (∃ *x*. (∀ *y* ∈ *L*. *y* < *x*) ∧ (∀ *y* ∈ *U*. *x* < *y*) ∧ *l* < *x* ∧ *P x*) ⟷ (∃ *x*. (∀ *y* ∈ (*insert l L*). *y* < *x*) ∧ (∀ *y* ∈ *U*. *x* < *y*) ∧ *P x*)
(∃ *x*. (∀ *y* ∈ *L*. *y* < *x*) ∧ (∀ *y* ∈ *U*. *x* < *y*) ∧ *x* < *u*) ⟷ (∃ *x*. (∀ *y* ∈ *L*. *y* < *x*) ∧ (∀ *y* ∈ (*insert u U*). *x* < *y*))
and (∃ *x*. (∀ *y* ∈ *L*. *y* < *x*) ∧ (∀ *y* ∈ *U*. *x* < *y*) ∧ *l* < *x*) ⟷ (∃ *x*. (∀ *y* ∈ (*insert l L*). *y* < *x*) ∧ (∀ *y* ∈ *U*. *x* < *y*)) **by** *auto*

lemma
gather-start[*noatp*]: (∃ *x*. *P x*) ≡ (∃ *x*. (∀ *y* ∈ {}. *y* < *x*) ∧ (∀ *y*∈ {}. *x* < *y*) ∧ *P x*)
by *simp*

Theorems for ∃ *z*. ∀ *x*. *x* < *z* ⟶ (*P x* ⟷ $P_{-\infty}$)

lemma *minf-lt*[*noatp*]: ∃ *z* . ∀ *x*. *x* < *z* ⟶ (*x* < *t* ⟷ *True*) **by** *auto*
lemma *minf-gt*[*noatp*]: ∃ *z* . ∀ *x*. *x* < *z* ⟶ (*t* < *x* ⟷ *False*)
by (*simp add*: *not-less*) (*rule exI*[**where** *x*=*t*], *auto simp add*: *less-le*)

lemma *minf-le*[*noatp*]: ∃ *z*. ∀ *x*. *x* < *z* ⟶ (*x* ≤ *t* ⟷ *True*) **by** (*auto simp add*: *less-le*)
lemma *minf-ge*[*noatp*]: ∃ *z*. ∀ *x*. *x* < *z* ⟶ (*t* ≤ *x* ⟷ *False*)
by (*auto simp add*: *less-le not-less not-le*)
lemma *minf-eq*[*noatp*]: ∃ *z*. ∀ *x*. *x* < *z* ⟶ (*x* = *t* ⟷ *False*) **by** *auto*
lemma *minf-neq*[*noatp*]: ∃ *z*. ∀ *x*. *x* < *z* ⟶ (*x* ≠ *t* ⟷ *True*) **by** *auto*
lemma *minf-P*[*noatp*]: ∃ *z*. ∀ *x*. *x* < *z* ⟶ (*P* ⟷ *P*) **by** *blast*

Theorems for ∃ *z*. ∀ *x*. *x* < *z* ⟶ (*P x* ⟷ $P_{+\infty}$)

lemma *pinf-gt*[*noatp*]: ∃ *z* . ∀ *x*. *z* < *x* ⟶ (*t* < *x* ⟷ *True*) **by** *auto*
lemma *pinf-lt*[*noatp*]: ∃ *z* . ∀ *x*. *z* < *x* ⟶ (*x* < *t* ⟷ *False*)
by (*simp add*: *not-less*) (*rule exI*[**where** *x*=*t*], *auto simp add*: *less-le*)

lemma *pinf-ge*[*noatp*]: ∃ *z*. ∀ *x*. *z* < *x* ⟶ (*t* ≤ *x* ⟷ *True*) **by** (*auto simp add*: *less-le*)

lemma *pinf-le*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x \leq t \longleftrightarrow \mathit{False})$
by (*auto simp add*: *less-le not-less not-le*)
lemma *pinf-eq*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x = t \longleftrightarrow \mathit{False})$ **by** *auto*
lemma *pinf-neq*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (x \neq t \longleftrightarrow \mathit{True})$ **by** *auto*
lemma *pinf-P*[*noatp*]: $\exists\, z.\ \forall\, x.\ z < x \longrightarrow (P \longleftrightarrow P)$ **by** *blast*

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

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

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

fix $x\ l\ u\ y$ **assume** tU: $t \in U$ **and** noU: $\forall t.\ l < t \wedge t < u \longrightarrow t \notin U$ **and** lx: $l < x$
and xu: $x<u$ **and** px: $x < t$ **and** ly: $l<y$ **and** yu:$y < u$
from $tU\ noU\ ly\ yu$ **have** tny: $t \neq y$ **by** *auto*
{assume H: $t < y$
from *less-trans*[*OF lx px*] *less-trans*[*OF H yu*]
have $l < t \wedge t < u$ **by** *simp*
with $tU\ noU$ **have** *False* **by** *auto***}**
hence $\neg\ t < y$ **by** *auto* **hence** $y \leq t$ **by** (*simp add*: *not-less*)
thus $y < t$ **using** *tny* **by** (*simp add*: *less-le*)
qed

lemma *lin-dense-gt*[*noatp*]: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t< u \longrightarrow t \notin U) \wedge l < x \wedge x < u \wedge t < x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow t < y)$
proof(*clarsimp*)
fix $x\ l\ u\ y$
assume tU: $t \in U$ **and** noU: $\forall t.\ l < t \wedge t < u \longrightarrow t \notin U$ **and** lx: $l < x$ **and** xu: $x<u$
and px: $t < x$ **and** ly: $l<y$ **and** yu:$y < u$
from $tU\ noU\ ly\ yu$ **have** tny: $t \neq y$ **by** *auto*
{assume H: $y< t$
from *less-trans*[*OF ly H*] *less-trans*[*OF px xu*] **have** $l < t \wedge t < u$ **by** *simp*
with $tU\ noU$ **have** *False* **by** *auto***}**
hence $\neg\ y<t$ **by** *auto* **hence** $t \leq y$ **by** (*auto simp add*: *not-less*)
thus $t < y$ **using** *tny* **by** (*simp add*:*less-le*)
qed

lemma *lin-dense-le*[*noatp*]: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t< u \longrightarrow t \notin U) \wedge l< x \wedge x < u \wedge x \leq t \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow y\leq t)$
proof(*clarsimp*)
fix $x\ l\ u\ y$
assume tU: $t \in U$ **and** noU: $\forall t.\ l < t \wedge t < u \longrightarrow t \notin U$ **and** lx: $l < x$ **and** xu: $x<u$
and px: $x \leq t$ **and** ly: $l<y$ **and** yu:$y < u$
from $tU\ noU\ ly\ yu$ **have** tny: $t \neq y$ **by** *auto*
{assume H: $t < y$
from *less-le-trans*[*OF lx px*] *less-trans*[*OF H yu*]
have $l < t \wedge t < u$ **by** *simp*
with $tU\ noU$ **have** *False* **by** *auto***}**
hence $\neg\ t < y$ **by** *auto* **thus** $y \leq t$ **by** (*simp add*: *not-less*)
qed

lemma *lin-dense-ge*[*noatp*]: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t< u \longrightarrow t \notin U) \wedge l< x \wedge x < u \wedge t \leq x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow t \leq y)$
proof(*clarsimp*)
fix $x\ l\ u\ y$
assume tU: $t \in U$ **and** noU: $\forall t.\ l < t \wedge t < u \longrightarrow t \notin U$ **and** lx: $l < x$ **and** xu: $x<u$
and px: $t \leq x$ **and** ly: $l<y$ **and** yu:$y < u$

from *tU noU ly yu* **have** *tny*: $t \neq y$ **by** *auto*
{assume *H*: $y < t$
from *less-trans*[*OF ly H*] *le-less-trans*[*OF px xu*]
have $l < t \wedge t < u$ **by** *simp*
with *tU noU* **have** *False* **by** *auto***}**
hence $\neg\ y<t$ **by** *auto* **thus** $t \leq y$ **by** (*simp add*: *not-less*)
qed
lemma *lin-dense-eq*[*noatp*]: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t< u \longrightarrow t \notin U) \wedge l< x \wedge x < u \wedge x = t \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow y= t)$ **by** *auto*
lemma *lin-dense-neq*[*noatp*]: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t< u \longrightarrow t \notin U) \wedge l< x \wedge x < u \wedge x \neq t \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow y\neq t)$ **by** *auto*
lemma *lin-dense-P*[*noatp*]: $\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t< u \longrightarrow t \notin U) \wedge l< x \wedge x < u \wedge P \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P)$ **by** *auto*

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

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

lemma *finite-set-intervals*[*noatp*]:
assumes *px*: $P\ x$ **and** *lx*: $l \leq x$ **and** *xu*: $x \leq u$ **and** *linS*: $l\in S$
and *uinS*: $u \in S$ **and** *fS*:*finite S* **and** *lS*: $\forall\ x\in S.\ l \leq x$ **and** *Su*: $\forall\ x\in S.\ x \leq u$
shows $\exists\ a \in S.\ \exists\ b \in S.\ (\forall\ y.\ a < y \wedge y < b \longrightarrow y \notin S) \wedge a \leq x \wedge x \leq b \wedge P\ x$
proof−
let *?Mx* $= \{y.\ y\in S \wedge y \leq x\}$
let *?xM* $= \{y.\ y\in S \wedge x \leq y\}$
let *?a* $=$ *Max ?Mx*
let *?b* $=$ *Min ?xM*
have *MxS*: $?Mx \subseteq S$ **by** *blast*
hence *fMx*: *finite ?Mx* **using** *fS finite-subset* **by** *auto*

from *lx linS* **have** *linMx*: $l \in ?Mx$ **by** *blast*
hence *Mxne*: $?Mx \neq \{\}$ **by** *blast*
have *xMS*: $?xM \subseteq S$ **by** *blast*
hence *fxM*: *finite ?xM* **using** *fS finite-subset* **by** *auto*
from *xu uinS* **have** *linxM*: $u \in ?xM$ **by** *blast*
hence *xMne*: $?xM \neq \{\}$ **by** *blast*
have *ax*:$?a \leq x$ **using** *Mxne fMx* **by** *auto*
have *xb*:$x \leq ?b$ **using** *xMne fxM* **by** *auto*
have $?a \in ?Mx$ **using** *Max-in*[*OF fMx Mxne*] **by** *simp* **hence** *ainS*: $?a \in S$ **using** *MxS* **by** *blast*
have $?b \in ?xM$ **using** *Min-in*[*OF fxM xMne*] **by** *simp* **hence** *binS*: $?b \in S$ **using** *xMS* **by** *blast*
have *noy*:$\forall\ y.\ ?a < y \wedge y < ?b \longrightarrow y \notin S$
proof(*clarsimp*)
fix y **assume** *ay*: $?a < y$ **and** *yb*: $y < ?b$ **and** *yS*: $y \in S$
from *yS* **have** $y \in ?Mx \vee y \in ?xM$ **by** (*auto simp add*: *linear*)
moreover {**assume** $y \in ?Mx$ **hence** $y \leq ?a$ **using** *Mxne fMx* **by** *auto* **with** *ay* **have** *False* **by** (*simp add*: *not-le*[*symmetric*])}
moreover {**assume** $y \in ?xM$ **hence** $?b \leq y$ **using** *xMne fxM* **by** *auto* **with** *yb* **have** *False* **by** (*simp add*: *not-le*[*symmetric*])}
ultimately show *False* **by** *blast*
qed
from *ainS binS noy ax xb px* **show** *?thesis* **by** *blast*
qed

lemma *finite-set-intervals2*[*noatp*]:
assumes *px*: $P\ x$ **and** *lx*: $l \leq x$ **and** *xu*: $x \leq u$ **and** *linS*: $l \in S$
and *uinS*: $u \in S$ **and** *fS*:*finite S* **and** *lS*: $\forall\ x \in S.\ l \leq x$ **and** *Su*: $\forall\ x \in S.\ x \leq u$
shows $(\exists\ s \in S.\ P\ s) \vee (\exists\ a \in S.\ \exists\ b \in S.\ (\forall\ y.\ a < y \wedge y < b \longrightarrow y \notin S) \wedge a < x \wedge x < b \wedge P\ x)$
proof−
from *finite-set-intervals*[**where** $P{=}P$, *OF px lx xu linS uinS fS lS Su*]
obtain a **and** b **where**
as: $a \in S$ **and** *bs*: $b \in S$ **and** *noS*:$\forall\ y.\ a < y \wedge y < b \longrightarrow y \notin S$
and *axb*: $a \leq x \wedge x \leq b \wedge P\ x$ **by** *auto*
from *axb* **have** $x{=}\ a \vee x{=}\ b \vee (a < x \wedge x < b)$ **by** (*auto simp add*: *le-less*)
thus *?thesis* **using** *px as bs noS* **by** *blast*
qed

end

22 The classical QE after Langford for dense linear orders

context *dense-linear-order*
begin

lemma *interval-empty-iff*:
$\{y.\ x < y \wedge y < z\} = \{\} \longleftrightarrow \neg\ x < z$
by (*auto dest*: *dense*)

lemma *dlo-qe-bnds*[*noatp*]:
assumes *ne*: $L \neq \{\}$ **and** *neU*: $U \neq \{\}$ **and** *fL*: *finite L* **and** *fU*: *finite U*
shows $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y)) \equiv (\forall\ l \in L.\ \forall u \in U.\ l < u)$
proof (*simp only*: *atomize-eq*, *rule iffI*)
assume *H*: $\exists x.\ (\forall y{\in}L.\ y < x) \wedge (\forall y{\in}U.\ x < y)$
then obtain *x* **where** *xL*: $\forall y{\in}L.\ y < x$ **and** *xU*: $\forall y{\in}U.\ x < y$ **by** *blast*
{fix *l u* **assume** *l*: $l \in L$ **and** *u*: $u \in U$
have $l < x$ **using** *xL l* **by** *blast*
also have $x < u$ **using** *xU u* **by** *blast*
finally (*less-trans*) **have** $l < u$ **.}**
thus $\forall l{\in}L.\ \forall u{\in}U.\ l < u$ **by** *blast*
next
assume *H*: $\forall l{\in}L.\ \forall u{\in}U.\ l < u$
let *?ML* = *Max L*
let *?MU* = *Min U*
from *fL ne* **have** *th1*: *?ML* $\in L$ **and** *th1'*: $\forall l{\in}L.\ l \leq$ *?ML* **by** *auto*
from *fU neU* **have** *th2*: *?MU* $\in U$ **and** *th2'*: $\forall u{\in}U.$ *?MU* $\leq u$ **by** *auto*
from *th1 th2 H* **have** *?ML* $<$ *?MU* **by** *auto*
with *dense* **obtain** *w* **where** *th3*: *?ML* $< w$ **and** *th4*: $w <$ *?MU* **by** *blast*
from *th3 th1'* **have** $\forall l \in L.\ l < w$ **by** *auto*
moreover from *th4 th2'* **have** $\forall u \in U.\ w < u$ **by** *auto*
ultimately show $\exists x.\ (\forall y{\in}L.\ y < x) \wedge (\forall y{\in}U.\ x < y)$ **by** *auto*
qed

lemma *dlo-qe-noub*[*noatp*]:
assumes *ne*: $L \neq \{\}$ **and** *fL*: *finite L*
shows $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in \{\}.\ x < y)) \equiv$ *True*
proof(*simp add*: *atomize-eq*)
from *gt-ex*[*of Max L*] **obtain** *M* **where** *M*: *Max L* $< M$ **by** *blast*
from *ne fL* **have** $\forall x \in L.\ x \leq$ *Max L* **by** *simp*
with *M* **have** $\forall x{\in}L.\ x < M$ **by** (*auto intro*: *le-less-trans*)
thus $\exists x.\ \forall y{\in}L.\ y < x$ **by** *blast*
qed

lemma *dlo-qe-nolb*[*noatp*]:
assumes *ne*: $U \neq \{\}$ **and** *fU*: *finite U*
shows $(\exists x.\ (\forall y \in \{\}.\ y < x) \wedge (\forall y \in U.\ x < y)) \equiv$ *True*
proof(*simp add*: *atomize-eq*)
from *lt-ex*[*of Min U*] **obtain** *M* **where** *M*: $M <$ *Min U* **by** *blast*
from *ne fU* **have** $\forall x \in U.$ *Min U* $\leq x$ **by** *simp*
with *M* **have** $\forall x{\in}U.\ M < x$ **by** (*auto intro*: *less-le-trans*)
thus $\exists x.\ \forall y{\in}U.\ x < y$ **by** *blast*
qed

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

using *gt-ex*[*of t*] **by** *auto*

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

lemma *axiom*[*noatp*]: *dense-linear-order* (*op* ≤) (*op* <) **by** (*rule dense-linear-order-axioms*)
lemma *atoms*[*noatp*]:
shows *TERM* (*less* :: *'a* ⇒ -)
and *TERM* (*less-eq* :: *'a* ⇒ -)
and *TERM* (*op* = :: *'a* ⇒ -) **.**

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

end

lemma *dnf*[*noatp*]:
(*P* & (*Q* | *R*)) = ((*P*&*Q*) | (*P*&*R*))
((*Q* | *R*) & *P*) = ((*Q*&*P*) | (*R*&*P*))
by *blast+*

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

lemma *nnf-simps*[*noatp*]:
(¬(*P* ∧ *Q*)) = (¬*P* ∨ ¬*Q*) (¬(*P* ∨ *Q*)) = (¬*P* ∧ ¬*Q*) (*P* ⟶ *Q*) = (¬*P* ∨ *Q*)
(*P* = *Q*) = ((*P* ∧ *Q*) ∨ (¬*P* ∧ ¬ *Q*)) (¬ ¬(*P*)) = *P*
by *blast+*

lemma *ex-distrib*[*noatp*]: (∃ *x*. *P x* ∨ *Q x*) ⟷ ((∃ *x*. *P x*) ∨ (∃ *x*. *Q x*)) **by** *blast*

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

use *~~/src/HOL/Tools/Qelim/langford.ML*
method-setup *dlo* = ⟪
Scan.succeed (*SIMPLE-METHOD' o LangfordQE.dlo-tac*)
⟫ *Langford's algorithm for quantifier elimination in dense linear orders*

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

Linear order without upper bounds

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

less (*op* ⊏) **and**
less ((-/ ⊏ -) [*51*, *51*] *50*)

end

locale *linorder-no-ub* = *linorder-stupid-syntax* +
assumes *gt-ex*: ∃ *y*. *less x y*
begin
lemma *ge-ex*[*noatp*]: ∃ *y*. *x* ⊑ *y* **using** *gt-ex* **by** *auto*

Theorems for ∃ *z*. ∀ *x*. *z* ⊏ *x* ⟶ (*P x* ⟷ $P_{+\infty}$)

lemma *pinf-conj*[*noatp*]:
assumes *ex1*: ∃ *z1*. ∀ *x*. *z1* ⊏ *x* ⟶ (*P1 x* ⟷ *P1'*)
and *ex2*: ∃ *z2*. ∀ *x*. *z2* ⊏ *x* ⟶ (*P2 x* ⟷ *P2'*)
shows ∃ *z*. ∀ *x*. *z* ⊏ *x* ⟶ ((*P1 x* ∧ *P2 x*) ⟷ (*P1'* ∧ *P2'*))
proof−
from *ex1 ex2* **obtain** *z1* **and** *z2* **where** *z1*: ∀ *x*. *z1* ⊏ *x* ⟶ (*P1 x* ⟷ *P1'*)
and *z2*: ∀ *x*. *z2* ⊏ *x* ⟶ (*P2 x* ⟷ *P2'*) **by** *blast*
from *gt-ex* **obtain** *z* **where** *z*:*ord.max less-eq z1 z2* ⊏ *z* **by** *blast*
from *z* **have** *zz1*: *z1* ⊏ *z* **and** *zz2*: *z2* ⊏ *z* **by** *simp-all*
{fix *x* **assume** *H*: *z* ⊏ *x*
from *less-trans*[*OF zz1 H*] *less-trans*[*OF zz2 H*]
have (*P1 x* ∧ *P2 x*) ⟷ (*P1'* ∧ *P2'*) **using** *z1 zz1 z2 zz2* **by** *auto*
}
thus *?thesis* **by** *blast*
qed

lemma *pinf-disj*[*noatp*]:
assumes *ex1*: ∃ *z1*. ∀ *x*. *z1* ⊏ *x* ⟶ (*P1 x* ⟷ *P1'*)
and *ex2*: ∃ *z2*. ∀ *x*. *z2* ⊏ *x* ⟶ (*P2 x* ⟷ *P2'*)
shows ∃ *z*. ∀ *x*. *z* ⊏ *x* ⟶ ((*P1 x* ∨ *P2 x*) ⟷ (*P1'* ∨ *P2'*))
proof−
from *ex1 ex2* **obtain** *z1* **and** *z2* **where** *z1*: ∀ *x*. *z1* ⊏ *x* ⟶ (*P1 x* ⟷ *P1'*)
and *z2*: ∀ *x*. *z2* ⊏ *x* ⟶ (*P2 x* ⟷ *P2'*) **by** *blast*
from *gt-ex* **obtain** *z* **where** *z*:*ord.max less-eq z1 z2* ⊏ *z* **by** *blast*
from *z* **have** *zz1*: *z1* ⊏ *z* **and** *zz2*: *z2* ⊏ *z* **by** *simp-all*
{fix *x* **assume** *H*: *z* ⊏ *x*
from *less-trans*[*OF zz1 H*] *less-trans*[*OF zz2 H*]
have (*P1 x* ∨ *P2 x*) ⟷ (*P1'* ∨ *P2'*) **using** *z1 zz1 z2 zz2* **by** *auto*
}
thus *?thesis* **by** *blast*
qed

lemma *pinf-ex*[*noatp*]: **assumes** *ex*:∃ *z*. ∀ *x*. *z* ⊏ *x* ⟶ (*P x* ⟷ *P1*) **and** *p1*: *P1* **shows** ∃ *x*. *P x*
proof−
from *ex* **obtain** *z* **where** *z*: ∀ *x*. *z* ⊏ *x* ⟶ (*P x* ⟷ *P1*) **by** *blast*
from *gt-ex* **obtain** *x* **where** *x*: *z* ⊏ *x* **by** *blast*
from *z x p1* **show** *?thesis* **by** *blast*

qed

end

Linear order without upper bounds

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

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

lemma *minf-conj*[*noatp*]:
assumes *ex1*: $\exists\, z1.\ \forall\, x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists\, z2.\ \forall\, x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists\, z.\ \forall\, x.\ x \sqsubset\ z \longrightarrow ((P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2'))$
proof−
from *ex1 ex2* **obtain** *z1* **and** *z2* **where** *z1*: $\forall\, x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$**and** *z2*: $\forall\, x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$ **by** *blast*
from *lt-ex* **obtain** *z* **where** *z*:$z \sqsubset$ *ord.min less-eq z1 z2* **by** *blast*
from *z* **have** *zz1*: $z \sqsubset z1$ **and** *zz2*: $z \sqsubset z2$ **by** *simp-all*
{fix *x* **assume** *H*: $x \sqsubset z$
from *less-trans*[*OF H zz1*] *less-trans*[*OF H zz2*]
have $(P1\ x \wedge P2\ x) \longleftrightarrow (P1' \wedge P2')$ **using** *z1 zz1 z2 zz2* **by** *auto*
}
thus *?thesis* **by** *blast*
qed

lemma *minf-disj*[*noatp*]:
assumes *ex1*: $\exists\, z1.\ \forall\, x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$
and *ex2*: $\exists\, z2.\ \forall\, x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$
shows $\exists\, z.\ \forall\, x.\ x \sqsubset\ z \longrightarrow ((P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2'))$
proof−
from *ex1 ex2* **obtain** *z1* **and** *z2* **where** *z1*: $\forall\, x.\ x \sqsubset z1 \longrightarrow (P1\ x \longleftrightarrow P1')$**and** *z2*: $\forall\, x.\ x \sqsubset z2 \longrightarrow (P2\ x \longleftrightarrow P2')$ **by** *blast*
from *lt-ex* **obtain** *z* **where** *z*:$z \sqsubset$ *ord.min less-eq z1 z2* **by** *blast*
from *z* **have** *zz1*: $z \sqsubset z1$ **and** *zz2*: $z \sqsubset z2$ **by** *simp-all*
{fix *x* **assume** *H*: $x \sqsubset z$
from *less-trans*[*OF H zz1*] *less-trans*[*OF H zz2*]
have $(P1\ x \vee P2\ x) \longleftrightarrow (P1' \vee P2')$ **using** *z1 zz1 z2 zz2* **by** *auto*
}
thus *?thesis* **by** *blast*
qed

lemma *minf-ex*[*noatp*]: **assumes** *ex*:$\exists\, z.\ \forall\, x.\ x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P1)$ **and** *p1*: *P1* **shows** $\exists\ x.\ P\ x$
proof−
from *ex* **obtain** *z* **where** *z*: $\forall\, x.\ x \sqsubset z \longrightarrow (P\ x \longleftrightarrow P1)$ **by** *blast*
from *lt-ex* **obtain** *x* **where** *x*: $x \sqsubset z$ **by** *blast*
from *z x p1* **show** *?thesis* **by** *blast*

qed

end

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

sublocale *constr-dense-linear-order* < *dense-linear-order*
  apply *unfold-locales*
  using *gt-ex lt-ex between-less*
    by (*auto*, *rule-tac x*=*between x y* **in** *exI*, *simp*)

context *constr-dense-linear-order*
begin

lemma *rinf-U*[*noatp*]:
  assumes *fU*: *finite U*
  and *lin-dense*: ∀ *x l u*. (∀ *t*. *l* ⊏ *t* ∧ *t*⊏ *u* ⟶ *t* ∉ *U*) ∧ *l*⊏ *x* ∧ *x* ⊏ *u* ∧ *P x*
  ⟶ (∀ *y*. *l* ⊏ *y* ∧ *y* ⊏ *u* ⟶ *P y*)
  and *nmpiU*: ∀ *x*. ¬ *MP* ∧ ¬*PP* ∧ *P x* ⟶ (∃ *u*∈ *U*. ∃ *u'* ∈ *U*. *u* ⊑ *x* ∧ *x* ⊑ *u'*)
  and *nmi*: ¬ *MP* **and** *npi*: ¬ *PP* **and** *ex*: ∃ *x*. *P x*
  shows ∃ *u*∈ *U*. ∃ *u'* ∈ *U*. *P* (*between u u'*)
proof−
  from *ex* **obtain** *x* **where** *px*: *P x* **by** *blast*
  from *px nmi npi nmpiU* **have** ∃ *u*∈ *U*. ∃ *u'* ∈ *U*. *u* ⊑ *x* ∧ *x* ⊑ *u'* **by** *auto*
  then obtain *u* **and** *u'* **where** *uU*:*u*∈ *U* **and** *uU'*: *u'* ∈ *U* **and** *ux*:*u* ⊑ *x* **and** *xu'*:*x* ⊑ *u'* **by** *auto*
  from *uU* **have** *Une*: *U* ≠ {} **by** *auto*
  term *linorder.Min less-eq*
  let *?l* = *linorder.Min less-eq U*
  let *?u* = *linorder.Max less-eq U*
  have *linM*: *?l* ∈ *U* **using** *fU Une* **by** *simp*
  have *uinM*: *?u* ∈ *U* **using** *fU Une* **by** *simp*
  have *lM*: ∀ *t*∈ *U*. *?l* ⊑ *t* **using** *Une fU* **by** *auto*
  have *Mu*: ∀ *t*∈ *U*. *t* ⊑ *?u* **using** *Une fU* **by** *auto*
  have *th*:*?l* ⊑ *u* **using** *uU Une lM* **by** *auto*
  from *order-trans*[*OF th ux*] **have** *lx*: *?l* ⊑ *x* **.**
  have *th*: *u'* ⊑ *?u* **using** *uU' Une Mu* **by** *simp*
  from *order-trans*[*OF xu' th*] **have** *xu*: *x* ⊑ *?u* **.**
  from *finite-set-intervals2*[**where** *P*=*P*,*OF px lx xu linM uinM fU lM Mu*]
  have (∃ *s*∈ *U*. *P s*) ∨
      (∃ *t1*∈ *U*. ∃ *t2* ∈ *U*. (∀ *y*. *t1* ⊏ *y* ∧ *y* ⊏ *t2* ⟶ *y* ∉ *U*) ∧ *t1* ⊏ *x* ∧ *x* ⊏ *t2* ∧ *P x*) **.**
  moreover { **fix** *u* **assume** *um*: *u*∈*U* **and** *pu*: *P u*
    have *between u u* = *u* **by** (*simp add*: *between-same*)

with *um pu* **have** P (*between u u*) **by** *simp*
with *um* **have** *?thesis* **by** *blast*}
moreover{
assume $\exists$ *t1*$\in$ U. $\exists$ *t2* $\in$ U. ($\forall$ *y*. *t1* $\sqsubset$ *y* $\wedge$ *y* $\sqsubset$ *t2* $\longrightarrow$ *y* $\notin$ U) $\wedge$ *t1* $\sqsubset$ *x* $\wedge$ *x* $\sqsubset$ *t2* $\wedge$ P *x*
then obtain *t1* **and** *t2* **where** *t1M*: *t1* $\in$ U **and** *t2M*: *t2*$\in$ U
and *noM*: $\forall$ *y*. *t1* $\sqsubset$ *y* $\wedge$ *y* $\sqsubset$ *t2* $\longrightarrow$ *y* $\notin$ U **and** *t1x*: *t1* $\sqsubset$ *x* **and** *xt2*: *x* $\sqsubset$ *t2* **and** *px*: P *x*
by *blast*
from *less-trans*[*OF t1x xt2*] **have** *t1t2*: *t1* $\sqsubset$ *t2* **.**
let *?u* = *between t1 t2*
from *between-less t1t2* **have** *t1lu*: *t1* $\sqsubset$ *?u* **and** *ut2*: *?u* $\sqsubset$ *t2* **by** *auto*
from *lin-dense noM t1x xt2 px t1lu ut2* **have** P *?u* **by** *blast*
with *t1M t2M* **have** *?thesis* **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

theorem *fr-eq*[*noatp*]:
assumes *fU*: *finite U*
and *lin-dense*: $\forall$ *x l u*. ($\forall$ *t*. *l* $\sqsubset$ *t* $\wedge$ *t*$\sqsubset$ *u* $\longrightarrow$ *t* $\notin$ U) $\wedge$ *l*$\sqsubset$ *x* $\wedge$ *x* $\sqsubset$ *u* $\wedge$ P *x* $\longrightarrow$ ($\forall$ *y*. *l* $\sqsubset$ *y* $\wedge$ *y* $\sqsubset$ *u* $\longrightarrow$ P *y*)
and *nmibnd*: $\forall$ *x*. $\neg$ *MP* $\wedge$ P *x* $\longrightarrow$ ($\exists$ *u*$\in$ U. *u* $\sqsubseteq$ *x*)
and *npibnd*: $\forall$ *x*. $\neg$*PP* $\wedge$ P *x* $\longrightarrow$ ($\exists$ *u*$\in$ U. *x* $\sqsubseteq$ *u*)
and *mi*: $\exists$ *z*. $\forall$ *x*. *x* $\sqsubset$ *z* $\longrightarrow$ (P *x* = *MP*) **and** *pi*: $\exists$ *z*. $\forall$ *x*. *z* $\sqsubset$ *x* $\longrightarrow$ (P *x* = *PP*)
shows ($\exists$ *x*. P *x*) $\equiv$ (*MP* $\vee$ *PP* $\vee$ ($\exists$ *u* $\in$ U. $\exists$ *u*$'\in$ U. P (*between u u*$'$)))
(**is** - $\equiv$ (- $\vee$ - $\vee$ *?F*) **is** *?E* $\equiv$ *?D*)
proof−
{
assume *px*: $\exists$ *x*. P *x*
have *MP* $\vee$ *PP* $\vee$ ($\neg$ *MP* $\wedge$ $\neg$ *PP*) **by** *blast*
moreover {**assume** *MP* $\vee$ *PP* **hence** *?D* **by** *blast*}
moreover {**assume** *nmi*: $\neg$ *MP* **and** *npi*: $\neg$ *PP*
from *npmibnd*[*OF nmibnd npibnd*]
have *nmpiU*: $\forall$ *x*. $\neg$ *MP* $\wedge$ $\neg$*PP* $\wedge$ P *x* $\longrightarrow$ ($\exists$ *u*$\in$ U. $\exists$ *u*$'$ $\in$ U. *u* $\sqsubseteq$ *x* $\wedge$ *x* $\sqsubseteq$ *u*$'$) **.**
from *rinf-U*[*OF fU lin-dense nmpiU nmi npi px*] **have** *?D* **by** *blast*}
ultimately have *?D* **by** *blast*}
moreover
{ **assume** *?D*
moreover {**assume** *m*:*MP* **from** *minf-ex*[*OF mi m*] **have** *?E* **.**}
moreover {**assume** *p*: *PP* **from** *pinf-ex*[*OF pi p*] **have** *?E* **. }**
moreover {**assume** *f*:*?F* **hence** *?E* **by** *blast*}
ultimately have *?E* **by** *blast*}
ultimately have *?E* = *?D* **by** *blast* **thus** *?E* $\equiv$ *?D* **by** *simp*
qed

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

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

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

lemma *ferrack-axiom*[*noatp*]: *constr-dense-linear-order less-eq less between*
by (*rule constr-dense-linear-order-axioms*)
lemma *atoms*[*noatp*]:
shows *TERM* (*less* :: *'a* ⇒ -)
and *TERM* (*less-eq* :: *'a* ⇒ -)
and *TERM* (*op* = :: *'a* ⇒ -) **.**

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

declaration ⟪
let
fun simps phi = map (Morphism.thm phi) [@{thm not-less}, @{thm not-le}]
fun generic-whatis phi =
let
val [lt, le] = map (Morphism.term phi) [@{term op ⊏}, @{term op ⊑}]
fun h x t =
case term-of t of
Const(op =, -)yz => if term-of x aconv y then Ferrante-Rackoff-Data.Eq
else Ferrante-Rackoff-Data.Nox
| @{term Not}$(Const(op =, -)$y$z) => if term-of x aconv y then Ferrante-Rackoff-Data.NEq
else Ferrante-Rackoff-Data.Nox
| byz => if Term.could-unify (b, lt) then
if term-of x aconv y then Ferrante-Rackoff-Data.Lt
else if term-of x aconv z then Ferrante-Rackoff-Data.Gt
else Ferrante-Rackoff-Data.Nox
else if Term.could-unify (b, le) then
if term-of x aconv y then Ferrante-Rackoff-Data.Le
else if term-of x aconv z then Ferrante-Rackoff-Data.Ge
else Ferrante-Rackoff-Data.Nox
else Ferrante-Rackoff-Data.Nox
| - => Ferrante-Rackoff-Data.Nox
in h end
fun ss phi = HOL-ss addsimps (simps phi)
in
Ferrante-Rackoff-Data.funs @{thm ferrack-axiom}
{isolate-conv = K (K (K Thm.reflexive)), whatis = generic-whatis, simpset = ss}

end
⟩⟩

end

use ~~/*src/HOL/Tools/Qelim/ferrante-rackoff.ML*

method-setup *ferrack* = ⟨⟨
Scan.succeed (*SIMPLE-METHOD'* o *FerranteRackoff.dlo-tac*)
⟩⟩ *Ferrante and Rackoff's algorithm for quantifier elimination in dense linear orders*

23.1 Ferrante and Rackoff algorithm over ordered fields

lemma *neg-prod-lt*:(*c*::'*a*::*ordered-field*) < *0* ⟹ ((*c*∗*x* < *0*) == (*x* > *0*))
proof−
assume *H*: *c* < *0*
have *c*∗*x* < *0* = (*0*/*c* < *x*) **by** (*simp only*: *neg-divide-less-eq*[*OF H*] *algebra-simps*)
also have ... = (*0* < *x*) **by** *simp*
finally show (*c*∗*x* < *0*) == (*x* > *0*) **by** *simp*
qed

lemma *pos-prod-lt*:(*c*::'*a*::*ordered-field*) > *0* ⟹ ((*c*∗*x* < *0*) == (*x* < *0*))
proof−
assume *H*: *c* > *0*
hence *c*∗*x* < *0* = (*0*/*c* > *x*) **by** (*simp only*: *pos-less-divide-eq*[*OF H*] *algebra-simps*)
also have ... = (*0* > *x*) **by** *simp*
finally show (*c*∗*x* < *0*) == (*x* < *0*) **by** *simp*
qed

lemma *neg-prod-sum-lt*: (*c*::'*a*::*ordered-field*) < *0* ⟹ ((*c*∗*x* + *t*< *0*) == (*x* > (− *1*/*c*)∗*t*))
proof−
assume *H*: *c* < *0*
have *c*∗*x* + *t*< *0* = (*c*∗*x* < −*t*) **by** (*subst less-iff-diff-less-0* [*of c*∗*x* −*t*], *simp*)
also have ... = (−*t*/*c* < *x*) **by** (*simp only*: *neg-divide-less-eq*[*OF H*] *algebra-simps*)
also have ... = ((− *1*/*c*)∗*t* < *x*) **by** *simp*
finally show (*c*∗*x* + *t* < *0*) == (*x* > (− *1*/*c*)∗*t*) **by** *simp*
qed

lemma *pos-prod-sum-lt*:(*c*::'*a*::*ordered-field*) > *0* ⟹ ((*c*∗*x* + *t* < *0*) == (*x* < (− *1*/*c*)∗*t*))
proof−
assume *H*: *c* > *0*
have *c*∗*x* + *t*< *0* = (*c*∗*x* < −*t*) **by** (*subst less-iff-diff-less-0* [*of c*∗*x* −*t*], *simp*)
also have ... = (−*t*/*c* > *x*) **by** (*simp only*: *pos-less-divide-eq*[*OF H*] *algebra-simps*)
also have ... = ((− *1*/*c*)∗*t* > *x*) **by** *simp*
finally show (*c*∗*x* + *t* < *0*) == (*x* < (− *1*/*c*)∗*t*) **by** *simp*
qed

lemma *sum-lt*:$((x::'a::pordered\text{-}ab\text{-}group\text{-}add) + t < 0) == (x < - t)$
using *less-diff-eq*[**where** $a= x$ **and** $b=t$ **and** $c=0$] **by** *simp*

lemma *neg-prod-le*:$(c::'a::ordered\text{-}field) < 0 \Longrightarrow ((c*x <= 0) == (x >= 0))$
proof−
assume *H*: $c < 0$
have $c*x <= 0 = (0/c <= x)$ **by** (*simp only*: *neg-divide-le-eq*[*OF H*] *algebra-simps*)
also have $\ldots = (0 <= x)$ **by** *simp*
finally show $(c*x <= 0) == (x >= 0)$ **by** *simp*
qed

lemma *pos-prod-le*:$(c::'a::ordered\text{-}field) > 0 \Longrightarrow ((c*x <= 0) == (x <= 0))$
proof−
assume *H*: $c > 0$
hence $c*x <= 0 = (0/c >= x)$ **by** (*simp only*: *pos-le-divide-eq*[*OF H*] *algebra-simps*)
also have $\ldots = (0 >= x)$ **by** *simp*
finally show $(c*x <= 0) == (x <= 0)$ **by** *simp*
qed

lemma *neg-prod-sum-le*: $(c::'a::ordered\text{-}field) < 0 \Longrightarrow ((c*x + t <= 0) == (x >= (- 1/c)*t))$
proof−
assume *H*: $c < 0$
have $c*x + t <= 0 = (c*x <= -t)$ **by** (*subst le-iff-diff-le-0* [*of* $c*x$ $-t$], *simp*)
also have $\ldots = (-t/c <= x)$ **by** (*simp only*: *neg-divide-le-eq*[*OF H*] *algebra-simps*)
also have $\ldots = ((- 1/c)*t <= x)$ **by** *simp*
finally show $(c*x + t <= 0) == (x >= (- 1/c)*t)$ **by** *simp*
qed

lemma *pos-prod-sum-le*:$(c::'a::ordered\text{-}field) > 0 \Longrightarrow ((c*x + t <= 0) == (x <= (- 1/c)*t))$
proof−
assume *H*: $c > 0$
have $c*x + t <= 0 = (c*x <= -t)$ **by** (*subst le-iff-diff-le-0* [*of* $c*x$ $-t$], *simp*)
also have $\ldots = (-t/c >= x)$ **by** (*simp only*: *pos-le-divide-eq*[*OF H*] *algebra-simps*)
also have $\ldots = ((- 1/c)*t >= x)$ **by** *simp*
finally show $(c*x + t <= 0) == (x <= (- 1/c)*t)$ **by** *simp*
qed

lemma *sum-le*:$((x::'a::pordered\text{-}ab\text{-}group\text{-}add) + t <= 0) == (x <= - t)$
using *le-diff-eq*[**where** $a= x$ **and** $b=t$ **and** $c=0$] **by** *simp*

lemma *nz-prod-eq*:$(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow ((c*x = 0) == (x = 0))$ **by** *simp*
lemma *nz-prod-sum-eq*: $(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow ((c*x + t = 0) == (x = (- 1/c)*t))$
proof−
assume *H*: $c \neq 0$
have $c*x + t = 0 = (c*x = -t)$ **by** (*subst eq-iff-diff-eq-0* [*of* $c*x$ $-t$], *simp*)
also have $\ldots = (x = -t/c)$ **by** (*simp only*: *nonzero-eq-divide-eq*[*OF H*] *algebra-simps*)

```
  finally show  (c*x + t = 0) == (x = (- 1/c)*t) by simp
qed
lemma sum-eq:((x::'a::pordered-ab-group-add) + t = 0) == (x = - t)
  using eq-diff-eq[where a= x and b=t and c=0] by simp


interpretation class-ordered-field-dense-linear-order: constr-dense-linear-order
 op <= op <
   λ x y. 1/2 * ((x::'a::{ordered-field,recpower,number-ring}) + y)
proof (unfold-locales, dlo, dlo, auto)
  fix x y::'a assume lt: x < y
  from  less-half-sum[OF lt] show x < (x + y) /2 by simp
next
  fix x y::'a assume lt: x < y
  from  gt-half-sum[OF lt] show (x + y) /2 < y by simp
qed

declaration⟪
let
fun earlier [] x y = false
        | earlier (h::t) x y =
    if h aconvc y then false else if h aconvc x then true else earlier t x y;

fun dest-frac ct = case term-of ct of
   Const (@{const-name HOL.divide},-) $ a $ b=>
    Rat.rat-of-quotient (snd (HOLogic.dest-number a), snd (HOLogic.dest-number
b))
| Const(@{const-name inverse}, -)$a => Rat.rat-of-quotient(1, HOLogic.dest-number
a |> snd)
| t => Rat.rat-of-int (snd (HOLogic.dest-number t))

fun mk-frac phi cT x =
 let val (a, b) = Rat.quotient-of-rat x
 in if b = 1 then Numeral.mk-cnumber cT a
    else Thm.capply
         (Thm.capply (Drule.cterm-rule (instantiate' [SOME cT] []) @{cpat op /})
                  (Numeral.mk-cnumber cT a))
         (Numeral.mk-cnumber cT b)
 end

fun whatis x ct = case term-of ct of
  Const(@{const-name HOL.plus}, -)$(Const(@{const-name HOL.times},-)$-$y)$-
=>
     if y aconv term-of x then (c*x+t,[(funpow 2 Thm.dest-arg1) ct, Thm.dest-arg
ct])
     else (Nox,[])
| Const(@{const-name HOL.plus}, -)$y$- =>
     if y aconv term-of x then (x+t,[Thm.dest-arg ct])
     else (Nox,[])
```

```
| Const(@{const-name HOL.times}, -)$-$y =>
    if y aconv term-of x then (c*x,[Thm.dest-arg1 ct])
    else (Nox,[])
| t => if t aconv term-of x then (x,[]) else (Nox,[]);

fun xnormalize-conv ctxt [] ct = reflexive ct
| xnormalize-conv ctxt (vs as (x::-)) ct =
  case term-of ct of
  Const(@{const-name HOL.less},-)$-$Const(@{const-name HOL.zero},-) =>
   (case whatis x (Thm.dest-arg1 ct) of
   (c*x+t,[c,t]) =>
     let
      val cr = dest-frac c
      val clt = Thm.dest-fun2 ct
      val cz = Thm.dest-arg ct
      val neg = cr </ Rat.zero
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
            (Thm.capply @{cterm Trueprop}
              (if neg then Thm.capply (Thm.capply clt c) cz
                else Thm.capply (Thm.capply clt cz) c))
      val cth = equal-elim (symmetric cthp) TrueI
       val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
[c,x,t])
           (if neg then @{thm neg-prod-sum-lt} else @{thm pos-prod-sum-lt})) cth
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
             (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
    in rth end
  | (x+t,[t]) =>
     let
      val T = ctyp-of-term x
      val th = instantiate' [SOME T] [SOME x, SOME t] @{thm sum-lt}
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
           (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
     in  rth end
  | (c*x,[c]) =>
     let
      val cr = dest-frac c
      val clt = Thm.dest-fun2 ct
      val cz = Thm.dest-arg ct
      val neg = cr </ Rat.zero
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
            (Thm.capply @{cterm Trueprop}
              (if neg then Thm.capply (Thm.capply clt c) cz
                else Thm.capply (Thm.capply clt cz) c))
      val cth = equal-elim (symmetric cthp) TrueI
       val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
[c,x])
           (if neg then @{thm neg-prod-lt} else @{thm pos-prod-lt})) cth
      val rth = th
```

```
    in rth end
  | - => reflexive ct)


| Const(@{const-name HOL.less-eq},-)$-$Const(@{const-name HOL.zero},-) =>
  (case whatis x (Thm.dest-arg1 ct) of
   (c*x+t,[c,t]) =>
     let
      val T = ctyp-of-term x
      val cr = dest-frac c
      val clt = Drule.cterm-rule (instantiate' [SOME T] []) @{cpat op <}
      val cz = Thm.dest-arg ct
      val neg = cr </ Rat.zero
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
            (Thm.capply @{cterm Trueprop}
              (if neg then Thm.capply (Thm.capply clt c) cz
                else Thm.capply (Thm.capply clt cz) c))
      val cth = equal-elim (symmetric cthp) TrueI
      val th = implies-elim (instantiate' [SOME T] (map SOME [c,x,t])
          (if neg then @{thm neg-prod-sum-le} else @{thm pos-prod-sum-le})) cth
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
              (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
    in rth end
  | (x+t,[t]) =>
     let
      val T = ctyp-of-term x
      val th = instantiate' [SOME T] [SOME x, SOME t] @{thm sum-le}
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
           (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
     in  rth end
  | (c*x,[c]) =>
     let
      val T = ctyp-of-term x
      val cr = dest-frac c
      val clt = Drule.cterm-rule (instantiate' [SOME T] []) @{cpat op <}
      val cz = Thm.dest-arg ct
      val neg = cr </ Rat.zero
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
            (Thm.capply @{cterm Trueprop}
              (if neg then Thm.capply (Thm.capply clt c) cz
                else Thm.capply (Thm.capply clt cz) c))
      val cth = equal-elim (symmetric cthp) TrueI
       val th = implies-elim (instantiate' [SOME (ctyp-of-term x)] (map SOME
[c,x])
          (if neg then @{thm neg-prod-le} else @{thm pos-prod-le})) cth
      val rth = th
    in rth end
  | - => reflexive ct)
```

```
| Const(op =,-)$-$Const(@{const-name HOL.zero},-) =>
  (case whatis x (Thm.dest-arg1 ct) of
   (c*x+t,[c,t]) =>
     let
      val T = ctyp-of-term x
      val cr = dest-frac c
      val ceq = Thm.dest-fun2 ct
      val cz = Thm.dest-arg ct
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
          (Thm.capply @{cterm Trueprop}
           (Thm.capply @{cterm Not} (Thm.capply (Thm.capply ceq c) cz)))
      val cth = equal-elim (symmetric cthp) TrueI
      val th = implies-elim
            (instantiate' [SOME T] (map SOME [c,x,t]) @{thm nz-prod-sum-eq})
cth
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
                (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
     in rth end
  | (x+t,[t]) =>
     let
      val T = ctyp-of-term x
      val th = instantiate' [SOME T] [SOME x, SOME t] @{thm sum-eq}
      val rth = Conv.fconv-rule (Conv.arg-conv (Conv.binop-conv
           (Normalizer.semiring-normalize-ord-conv ctxt (earlier vs)))) th
     in  rth end
  | (c*x,[c]) =>
     let
      val T = ctyp-of-term x
      val cr = dest-frac c
      val ceq = Thm.dest-fun2 ct
      val cz = Thm.dest-arg ct
      val cthp = Simplifier.rewrite (local-simpset-of ctxt)
          (Thm.capply @{cterm Trueprop}
           (Thm.capply @{cterm Not} (Thm.capply (Thm.capply ceq c) cz)))
      val cth = equal-elim (symmetric cthp) TrueI
      val rth = implies-elim
            (instantiate' [SOME T] (map SOME [c,x]) @{thm nz-prod-eq}) cth
     in rth end
  | - => reflexive ct);

local
  val less-iff-diff-less-0 = mk-meta-eq @{thm less-iff-diff-less-0}
  val le-iff-diff-le-0 = mk-meta-eq @{thm le-iff-diff-le-0}
  val eq-iff-diff-eq-0 = mk-meta-eq @{thm eq-iff-diff-eq-0}
in
fun field-isolate-conv phi ctxt vs ct = case term-of ct of
  Const(@{const-name HOL.less},-)$a$b =>
   let val (ca,cb) = Thm.dest-binop ct
       val T = ctyp-of-term ca
```

```
      val th = instantiate' [SOME T] [SOME ca, SOME cb] less-iff-diff-less-0
      val nth = Conv.fconv-rule
        (Conv.arg-conv (Conv.arg1-conv
            (Normalizer.semiring-normalize-ord-conv @{context} (earlier vs)))) th
      val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
  in rth end
| Const(@{const-name HOL.less-eq},-)$a$b =>
  let val (ca,cb) = Thm.dest-binop ct
      val T = ctyp-of-term ca
      val th = instantiate' [SOME T] [SOME ca, SOME cb] le-iff-diff-le-0
      val nth = Conv.fconv-rule
        (Conv.arg-conv (Conv.arg1-conv
            (Normalizer.semiring-normalize-ord-conv @{context} (earlier vs)))) th
      val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
  in rth end

| Const(op =,-)$a$b =>
  let val (ca,cb) = Thm.dest-binop ct
      val T = ctyp-of-term ca
      val th = instantiate' [SOME T] [SOME ca, SOME cb] eq-iff-diff-eq-0
      val nth = Conv.fconv-rule
        (Conv.arg-conv (Conv.arg1-conv
            (Normalizer.semiring-normalize-ord-conv @{context} (earlier vs)))) th
      val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
  in rth end
| @{term Not} $(Const(op =,-)$a$b) => Conv.arg-conv (field-isolate-conv phi ctxt
vs) ct
| - => reflexive ct
end;

fun classfield-whatis phi =
 let
  fun h x t =
   case term-of t of
     Const(op =, -)$y$z => if term-of x aconv y then Ferrante-Rackoff-Data.Eq
                           else Ferrante-Rackoff-Data.Nox
  | @{term Not}$(Const(op =, -)$y$z) => if term-of x aconv y then Ferrante-Rackoff-Data.NEq
                           else Ferrante-Rackoff-Data.Nox
   | Const(@{const-name HOL.less},-)$y$z =>
      if term-of x aconv y then Ferrante-Rackoff-Data.Lt
       else if term-of x aconv z then Ferrante-Rackoff-Data.Gt
       else Ferrante-Rackoff-Data.Nox
   | Const (@{const-name HOL.less-eq},-)$y$z =>
        if term-of x aconv y then Ferrante-Rackoff-Data.Le
        else if term-of x aconv z then Ferrante-Rackoff-Data.Ge
        else Ferrante-Rackoff-Data.Nox
   | - => Ferrante-Rackoff-Data.Nox
 in h end;
fun class-field-ss phi =
```

HOL-basic-ss addsimps ([@{*thm linorder-not-less*}, @{*thm linorder-not-le*}])
addsplits [@{*thm abs-split*},@{*thm split-max*}, @{*thm split-min*}]

in
Ferrante-Rackoff-Data.funs @{*thm class-ordered-field-dense-linear-order.ferrack-axiom*}
{*isolate-conv* = *field-isolate-conv*, *whatis* = *classfield-whatis*, *simpset* = *class-field-ss*}
end
⟩⟩

lemma *upper-bound-finite-set*:
assumes *fS*: *finite S*
shows ∃(*a*::'*a*::*linorder*). ∀ *x* ∈ *S*. *f x* ≤ *a*
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **by** *simp*
next
case (*2 x F*)
from *2.hyps* **obtain** *a* **where** *a*:∀ *x* ∈*F*. *f x* ≤ *a* **by** *blast*
let *?a* = *max a* (*f x*)
have *m*: *a* ≤ *?a f x* ≤ *?a* **by** *simp-all*
{fix *y* **assume** *y*: *y* ∈ *insert x F*
{assume *y* = *x* **hence** *f y* ≤ *?a* **using** *m* **by** *simp***}**
moreover
{assume *yF*: *y*∈ *F* **from** *a*[*rule-format*, *OF yF*] *m* **have** *f y* ≤ *?a* **by** (*simp add*: *max-def*)**}**
ultimately have *f y* ≤ *?a* **using** *y* **by** *blast***}**
then show *?case* **by** *blast*
qed

lemma *lower-bound-finite-set*:
assumes *fS*: *finite S*
shows ∃(*a*::'*a*::*linorder*). ∀ *x* ∈ *S*. *f x* ≥ *a*
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **by** *simp*
next
case (*2 x F*)
from *2.hyps* **obtain** *a* **where** *a*:∀ *x* ∈*F*. *f x* ≥ *a* **by** *blast*
let *?a* = *min a* (*f x*)
have *m*: *a* ≥ *?a f x* ≥ *?a* **by** *simp-all*
{fix *y* **assume** *y*: *y* ∈ *insert x F*
{assume *y* = *x* **hence** *f y* ≥ *?a* **using** *m* **by** *simp***}**
moreover
{assume *yF*: *y*∈ *F* **from** *a*[*rule-format*, *OF yF*] *m* **have** *f y* ≥ *?a* **by** (*simp add*: *min-def*)**}**
ultimately have *f y* ≥ *?a* **using** *y* **by** *blast***}**
then show *?case* **by** *blast*
qed

lemma *bound-finite-set*: **assumes** *f*: *finite S*
shows ∃ *a*. ∀ *x* ∈*S*. (*f x*:: '*a*::*linorder*) ≤ *a*

proof−
let *?F* = *f* ‘ *S*
from *f* **have** *fF*: *finite ?F* **by** *simp*
let *?a* = *Max ?F*
{assume *S* = {} **hence** *?thesis* **by** *blast***}**
moreover
{assume *Se*: *S* ≠ {} **hence** *Fe*: *?F* ≠ {} **by** *simp*
{fix *x* **assume** *x*: *x* ∈ *S*
hence *th0*: *f x* ∈ *?F* **by** *simp*
hence *f x* ≤ *?a* **using** *Max-ge*[*OF fF th0*] **..}**
hence *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

end

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

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

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

lemma *hassize-image-inj*: **assumes** *f*: *inj-on f S* **and** *S*: *S hassize n*
shows *f* ‘ *S hassize n*
using *f S card-image*[*OF f*]
by (*simp add*: *hassize-def inj-on-def*)

24.1 Finite Cartesian products, with indexing and lambdas.

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

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

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

lemma *stupid-ext*: (∀ *x*. *f x* = *g x*) ⟷ (*f* = *g*)
apply *auto*
apply (*rule ext*)

apply *auto*
done

lemma *Cart-eq*: $((x::\ 'a\ \hat{}\ 'b) = y) \longleftrightarrow (\forall\, i.\ x\$i = y\$i)$
by (*simp add*: *Cart-nth-inject* [*symmetric*] *expand-fun-eq*)

lemma *Cart-lambda-beta* [*simp*]: *Cart-lambda g $ i = g i*
by (*simp add*: *Cart-lambda-inverse*)

lemma *Cart-lambda-unique*:
fixes *f* :: $'a\ \hat{}\ 'b$
shows $(\forall\, i.\ f\$i = g\ i) \longleftrightarrow$ *Cart-lambda g = f*
by (*auto simp add*: *Cart-eq*)

lemma *Cart-lambda-eta*: $(\chi\ i.\ (g\$i)) = g$
by (*simp add*: *Cart-eq*)

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

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

definition *fstcart*:: $'a\ \hat{}('m + 'n) \Rightarrow 'a\ \hat{}\ 'm$ **where**
fstcart f = (χ i. (f$(Inl i)))

definition *sndcart*:: $'a\ \hat{}('m + 'n) \Rightarrow 'a\ \hat{}\ 'n$ **where**
sndcart f = (χ i. (f$(Inr i)))

lemma *nth-pastecart-Inl* [*simp*]: *pastecart f g $ Inl a = f$a*
unfolding *pastecart-def* **by** *simp*

lemma *nth-pastecart-Inr* [*simp*]: *pastecart f g $ Inr b = g$b*
unfolding *pastecart-def* **by** *simp*

lemma *nth-fstcart* [*simp*]: *fstcart f $ i = f $ Inl i*
unfolding *fstcart-def* **by** *simp*

lemma *nth-sndtcart* [*simp*]: *sndcart f $ i = f $ Inr i*
unfolding *sndcart-def* **by** *simp*

lemma *finite-sum-image*: $(UNIV::('a + 'b)\ set) = range\ Inl \cup range\ Inr$
by (*auto*, *case-tac x*, *auto*)

lemma *fstcart-pastecart*: *fstcart* (*pastecart* $(x::'a\ \hat{}'m\ )\ (y::\ 'a\ \hat{}\ 'n)) = x$
by (*simp add*: *Cart-eq*)

lemma *sndcart-pastecart*: *sndcart* (*pastecart* $(x::'a\ \hat{}'m\ )\ (y::\ 'a\ \hat{}\ 'n)) = y$
by (*simp add*: *Cart-eq*)

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

by (*simp add*: *Cart-eq pastecart-def fstcart-def sndcart-def split*: *sum.split*)

lemma *pastecart-eq*: $(x = y) \longleftrightarrow (fstcart\ x = fstcart\ y) \wedge (sndcart\ x = sndcart\ y)$
using *pastecart-fst-snd*[*of x*] *pastecart-fst-snd*[*of y*] **by** *metis*

lemma *forall-pastecart*: $(\forall p.\ P\ p) \longleftrightarrow (\forall x\ y.\ P\ (pastecart\ x\ y))$
by (*metis pastecart-fst-snd fstcart-pastecart sndcart-pastecart*)

lemma *exists-pastecart*: $(\exists p.\ P\ p) \longleftrightarrow (\exists x\ y.\ P\ (pastecart\ x\ y))$
by (*metis pastecart-fst-snd fstcart-pastecart sndcart-pastecart*)

end

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

theory *Glbs*
imports *Lubs*
begin

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

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

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

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

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

lemma *leastPD1*: *greatestP P x* ==> *P x*
by (*simp add*: *greatestP-def*)

lemma *greatestPD2*: *greatestP P x* ==> *Collect P* *<= *x*
by (*simp add*: *greatestP-def*)

lemma *greatestPD3*: [| *greatestP P x*; *y*: *Collect P* |] ==> *x* >= *y*
by (*blast dest*!: *greatestPD2 setleD*)

lemma *isGlbD1*: *isGlb R S x ==> x <=* S*
by (*simp add*: *isGlb-def isLb-def greatestP-def*)

lemma *isGlbD1a*: *isGlb R S x ==> x*: *R*
by (*simp add*: *isGlb-def isLb-def greatestP-def*)

lemma *isGlb-isLb*: *isGlb R S x ==> isLb R S x*
apply (*simp add*: *isLb-def*)
apply (*blast dest*: *isGlbD1 isGlbD1a*)
done

lemma *isGlbD2*: [| *isGlb R S x*; *y* : *S* |] ==> *y >= x*
by (*blast dest*!: *isGlbD1 setgeD*)

lemma *isGlbD3*: *isGlb R S x ==> greatestP*(*isLb R S*) *x*
by (*simp add*: *isGlb-def*)

lemma *isGlbI1*: *greatestP*(*isLb R S*) *x ==> isGlb R S x*
by (*simp add*: *isGlb-def*)

lemma *isGlbI2*: [| *isLb R S x*; *Collect* (*isLb R S*) **<= x* |] ==> *isGlb R S x*
by (*simp add*: *isGlb-def greatestP-def*)

lemma *isLbD*: [| *isLb R S x*; *y* : *S* |] ==> *y >= x*
by (*simp add*: *isLb-def setge-def*)

lemma *isLbD2*: *isLb R S x ==> x <=* S*
by (*simp add*: *isLb-def*)

lemma *isLbD2a*: *isLb R S x ==> x*: *R*
by (*simp add*: *isLb-def*)

lemma *isLbI*: [| *x <=* S* ; *x*: *R* |] ==> *isLb R S x*
by (*simp add*: *isLb-def*)

lemma *isGlb-le-isLb*: [| *isGlb R S x*; *isLb R S y* |] ==> *x >= y*
apply (*simp add*: *isGlb-def*)
apply (*blast intro*!: *greatestPD3*)
done

lemma *isGlb-ubs*: *isGlb R S x ==> lbs R S *<= x*
apply (*simp add*: *lbs-def isGlb-def*)
apply (*erule greatestPD2*)
done

end

26 Infinite-Set: Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

26.1 Infinite Sets

Some elementary facts about infinite sets, mostly by Stefan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

abbreviation
  infinite :: *'a set* ⇒ *bool* **where**
  infinite S == ¬ *finite S*

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

lemma *infinite-imp-nonempty*: *infinite S* ==> $S \neq \{\}$
  by *auto*

lemma *infinite-remove*:
  infinite S ⟹ *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* $(S \cup T)$
  by *simp*

lemma *infinite-super*:
  assumes *T*: $S \subseteq T$ **and** *S*: *infinite S*
  shows *infinite T*

proof
  assume *finite T*
  with *T* **have** *finite S* **by** (*simp add*: *finite-subset*)
  with *S* **show** *False* **by** *simp*
qed

As a concrete example, we prove that the set of natural numbers is infinite.

lemma *finite-nat-bounded*:
  assumes *S*: *finite* (*S*::*nat set*)
  shows ∃ *k*. *S* ⊆ {..<*k*} (**is** ∃ *k*. *?bounded S k*)
using *S*
proof *induct*
  have *?bounded* {} *0* **by** *simp*
  then show ∃ *k*. *?bounded* {} *k* **..**
next
  fix *S x*
  assume ∃ *k*. *?bounded S k*
  then obtain *k* **where** *k*: *?bounded S k* **..**
  show ∃ *k*. *?bounded* (*insert x S*) *k*
  proof (*cases x* < *k*)
    case *True*
    with *k* **show** *?thesis* **by** *auto*
  next
    case *False*
    with *k* **have** *?bounded S* (*Suc x*) **by** *auto*
    then show *?thesis* **by** *auto*
  qed
qed

lemma *finite-nat-iff-bounded*:
  finite (*S*::*nat set*) = (∃ *k*. *S* ⊆ {..<*k*}) (**is** *?lhs* = *?rhs*)
proof
  assume *?lhs*
  then show *?rhs* **by** (*rule finite-nat-bounded*)
next
  assume *?rhs*
  then obtain *k* **where** *S* ⊆ {..<*k*} **..**
  then show *finite S*
    by (*rule finite-subset*) *simp*
qed

lemma *finite-nat-iff-bounded-le*:
  finite (*S*::*nat set*) = (∃ *k*. *S* ⊆ {..*k*}) (**is** *?lhs* = *?rhs*)
proof
  assume *?lhs*
  then obtain *k* **where** *S* ⊆ {..<*k*}
    by (*blast dest*: *finite-nat-bounded*)
  then have *S* ⊆ {..*k*} **by** *auto*

```
  then show ?rhs ..
next
  assume ?rhs
  then obtain k where S ⊆ {..k} ..
  then show finite S
    by (rule finite-subset) simp
qed

lemma infinite-nat-iff-unbounded:
  infinite (S::nat set) = (∀ m. ∃ n. m<n ∧ n∈S)
  (is ?lhs = ?rhs)
proof
  assume ?lhs
  show ?rhs
  proof (rule ccontr)
    assume ¬ ?rhs
    then obtain m where m: ∀ n. m<n ⟶ n∉S by blast
    then have S ⊆ {..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 ⊆ {..m}
      by (auto simp add: finite-nat-iff-bounded-le)
    then have ∀ n. m<n ⟶ n∉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 [simp]: infinite (UNIV :: nat set)
  by (auto simp add: infinite-nat-iff-unbounded)

lemma nat-not-finite [elim]: finite (UNIV::nat set) ⟹ R
  by simp
```

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

```
lemma range-inj-infinite:
  inj (f::nat ⇒ 'a) ⟹ infinite (range f)
proof
  assume 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* ⊆ (*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* (λ*n T*. *T* − {*SOME e*. *e* ∈ *T*})
def *pick* ≡ λ*n*. (*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)
```

26.2 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

```
definition
  Inf-many :: ('a ⇒ bool) ⇒ bool  (binder INFM  10) where
  Inf-many P = infinite {x. P x}

definition
  Alm-all :: ('a ⇒ bool) ⇒ bool  (binder MOST  10) where
  Alm-all P = (¬ (INFM x. ¬ P x))

notation (xsymbols)
  Inf-many  (binder ∃∞ 10) and
  Alm-all  (binder ∀∞ 10)

notation (HTML output)
```

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

lemma *INFM-EX*:
$(\exists_\infty x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$
unfolding *Inf-many-def*
proof (*rule ccontr*)
assume *inf*: *infinite* $\{x.\ P\ x\}$
assume $\neg$ *?thesis* **then have** $\{x.\ P\ x\} = \{\}$ **by** *simp*
then have *finite* $\{x.\ P\ x\}$ **by** *simp*
with *inf* **show** *False* **by** *simp*
qed

lemma *MOST-iff-finiteNeg*: $(\forall_\infty x.\ P\ x) = \mathit{finite}\ \{x.\ \neg\ P\ x\}$
by (*simp add*: *Alm-all-def Inf-many-def*)

lemma *ALL-MOST*: $\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$
by (*simp add*: *MOST-iff-finiteNeg*)

lemma *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 *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-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 *prems* **by** (*simp add*: *MOST-conj-distrib*)
thus *?thesis* **by** (*rule MOST-mono*) *simp*
qed

lemma *not-INFM* [*simp*]: $\neg\ (INFM\ x.\ P\ x) \longleftrightarrow (MOST\ x.\ \neg\ P\ x)$
unfolding *Alm-all-def not-not* **..**

lemma *not-MOST* [*simp*]: $\neg\ (MOST\ x.\ P\ x) \longleftrightarrow (INFM\ x.\ \neg\ P\ x)$
unfolding *Alm-all-def not-not* **..**

lemma *INFM-const* [*simp*]: $(INFM\ x{::}'a.\ P) \longleftrightarrow P \wedge infinite\ (UNIV{::}'a\ set)$
unfolding *Inf-many-def* **by** *simp*

lemma *MOST-const* [*simp*]: $(MOST\ x{::}'a.\ P) \longleftrightarrow P \vee finite\ (UNIV{::}'a\ set)$
unfolding *Alm-all-def* **by** *simp*

lemma *INFM-nat*: $(\exists_\infty n.\ P\ (n{::}nat)) = (\forall\, m.\ \exists\, n.\ m{<}n \wedge P\ n)$
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded*)

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ (n{::}nat)) = (\forall\, m.\ \exists\, n.\ m{\leq}n \wedge P\ n)$
by (*simp add*: *Inf-many-def infinite-nat-iff-unbounded-le*)

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n{::}nat)) = (\exists\, m.\ \forall\, n.\ m{<}n \longrightarrow P\ n)$
by (*simp add*: *Alm-all-def INFM-nat*)

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n{::}nat)) = (\exists\, m.\ \forall\, n.\ m{\leq}n \longrightarrow P\ n)$
by (*simp add*: *Alm-all-def INFM-nat-le*)

26.3 Enumeration of an Infinite Set

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

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

lemma *enumerate-Suc'*:
$enumerate\ S\ (Suc\ n) = enumerate\ (S - \{enumerate\ S\ 0\})\ n$
by *simp*

lemma *enumerate-in-set*: $infinite\ S \Longrightarrow enumerate\ S\ n : S$
apply (*induct n arbitrary*: *S*)
apply (*fastsimp intro*: *LeastI dest*!: *infinite-imp-nonempty*)
apply *simp*
apply (*metis Collect-def Collect-mem-eq DiffE infinite-remove*)
done

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

lemma *enumerate-step*: $infinite\ S \Longrightarrow enumerate\ S\ n < enumerate\ S\ (Suc\ n)$
apply (*induct n arbitrary*: *S*)
apply (*rule order-le-neq-trans*)

apply (*simp add*: *enumerate-0 Least-le enumerate-in-set*)
apply (*simp only*: *enumerate-Suc'*)
apply (*subgoal-tac enumerate* (*S* − {*enumerate S 0*}) *0* : *S* − {*enumerate S 0*})
apply (*blast intro*: *sym*)
apply (*simp add*: *enumerate-in-set del*: *Diff-iff*)
apply (*simp add*: *enumerate-Suc'*)
done

lemma *enumerate-mono*: *m*<*n* ⟹ *infinite S* ⟹ *enumerate S m* < *enumerate S n*
apply (*erule less-Suc-induct*)
apply (*auto intro*: *enumerate-step*)
done

26.4 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

definition
atmost-one :: *'a set* ⇒ *bool* **where**
atmost-one S = (∀ *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

27 Numeral-Type: Numeral Syntax for Types

theory *Numeral-Type*
imports *Main*
begin

27.1 Preliminary lemmas

lemma (**in** *type-definition*) *univ*:
UNIV = *Abs* ' *A*
proof
show *Abs* ' *A* ⊆ *UNIV* **by** (*rule subset-UNIV*)
show *UNIV* ⊆ *Abs* ' *A*

```
  proof
    fix x :: 'b
    have x = Abs (Rep x) by (rule Rep-inverse [symmetric])
    moreover have Rep x ∈ A by (rule Rep)
    ultimately show x ∈ Abs ' A by (rule image-eqI)
  qed
qed

lemma (in type-definition) card: card (UNIV :: 'b set) = card A
  by (simp add: univ card-image inj-on-def Abs-inject)
```

27.2 Cardinalities of types

```
syntax -type-card :: type => nat ((1CARD/(1'(-'))))

translations CARD(t) => CONST card (CONST UNIV :: t set)

typed-print-translation ⟪
let
  fun card-univ-tr' show-sorts - [Const (@{const-syntax UNIV}, Type(-,[T,-]))] =
    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 × '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 insert-None-conv-UNIV [symmetric]
  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*)

27.3 Classes with at least 1 and 2

Class finite already captures "at least 1"

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

27.4 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

27.5 Locale for modular arithmetic subtypes

locale *mod-type* =
fixes *n* :: *int*
and *Rep* :: '*a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*,*power*} ⇒ *int*
and *Abs* :: *int* ⇒ '*a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*,*power*}
assumes *type*: *type-definition Rep Abs* {*0*..<*n*}
and *size1*: *1* < *n*
and *zero-def*: *0* = *Abs 0*
and *one-def*: *1* = *Abs 1*
and *add-def*: *x* + *y* = *Abs* ((*Rep x* + *Rep y*) *mod n*)
and *mult-def*: *x* ∗ *y* = *Abs* ((*Rep x* ∗ *Rep y*) *mod n*)
and *diff-def*: *x* − *y* = *Abs* ((*Rep x* − *Rep y*) *mod n*)
and *minus-def*: − *x* = *Abs* ((− *Rep x*) *mod n*)
and *power-def*: *x* ^ *k* = *Abs* (*Rep x* ^ *k mod n*)
begin

lemma *size0*: *0* < *n*
by (*cut-tac size1*, *simp*)

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

lemma *Rep-less-n*: *Rep x* < *n*
by (*rule type-definition.Rep* [*OF type*, *simplified*, *THEN conjunct2*])

lemma *Rep-le-n*: *Rep x* ≤ *n*
by (*rule Rep-less-n* [*THEN order-less-imp-le*])

lemma *Rep-inject-sym*: *x* = *y* ⟷ *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* ∈ {*0*..<*n*} ⟹ *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 IntDiv.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 ring-simps*)
done

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

apply (*intro-classes*, *unfold definitions*)
apply (*simp-all add*: *Rep-simps zmod-simps add-ac mult-assoc*
mod-pos-pos-trivial size1)
done

end

locale *mod-ring* = *mod-type* +
constrains *n* :: *int*
and *Rep* :: *'a*::{*number-ring*,*power*} ⇒ *int*
and *Abs* :: *int* ⇒ *'a*::{*number-ring*,*power*}
begin

lemma *of-nat-eq*: *of-nat k* = *Abs* (*int k mod n*)
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

27.6 Number ring instances

Unfortunately a number ring instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation *num1* :: {*comm-ring*,*comm-monoid-mult*,*number*,*recpower*}
begin

lemma *num1-eq-iff*: (*x*::*num1*) = (*y*::*num1*) ⟷ *True*
  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*,*power*}
begin

definition *Abs-bit0'* :: *int* ⇒ *'a bit0* **where**
  Abs-bit0' x = *Abs-bit0* (*x mod int CARD*(*'a bit0*))

definition *Abs-bit1'* :: *int* ⇒ *'a bit1* **where**
  Abs-bit1' x = *Abs-bit1* (*x mod int CARD*(*'a bit1*))

definition *0* = *Abs-bit0 0*
definition *1* = *Abs-bit0 1*
definition *x* + *y* = *Abs-bit0'* (*Rep-bit0 x* + *Rep-bit0 y*)
definition *x* ∗ *y* = *Abs-bit0'* (*Rep-bit0 x* ∗ *Rep-bit0 y*)
definition *x* − *y* = *Abs-bit0'* (*Rep-bit0 x* − *Rep-bit0 y*)
definition − *x* = *Abs-bit0'* (− *Rep-bit0 x*)
definition *x* ^ *k* = *Abs-bit0'* (*Rep-bit0 x* ^ *k*)

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*)
definition *x* ^ *k* = *Abs-bit1'* (*Rep-bit1 x* ^ *k*)

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*])
apply (*rule power-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*])
apply (*rule power-bit1-def* [*unfolded Abs-bit1'-def*])
done

instance *bit0* :: (*finite*) {*comm-ring-1*,*recpower*}
  by (*rule bit0.comm-ring-1 bit0.recpower*)+

instance *bit1* :: (*finite*) {*comm-ring-1*,*recpower*}
  by (*rule bit1.comm-ring-1 bit1.recpower*)+

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*

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

27.7 Syntax

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

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

parse-translation ⟪
let

val num1-const = *Syntax.const Numeral-Type.num1*;
val num0-const = *Syntax.const Numeral-Type.num0*;
val B0-const = *Syntax.const Numeral-Type.bit0*;
val B1-const = *Syntax.const Numeral-Type.bit1*;

fun mk-bintype n =
let
fun mk-bit n = *if n* = *0 then B0-const else B1-const*;
fun bin-of n =
if n = *1 then num1-const*
else if n = *0 then num0-const*
else if n = *~1 then raise TERM* (*negative type numeral*, [])
else
let val (*q*, *r*) = *Integer.div-mod n 2*;

```
      in mk-bit r $ bin-of q end;
  in bin-of n end;

fun numeral-tr (*-NumeralType*) [Const (str, -)] =
      mk-bintype (valOf (Int.fromString str))
  | numeral-tr (*-NumeralType*) ts = raise TERM (numeral-tr, ts);

in [(-NumeralType, numeral-tr)] end;
⟫

print-translation ⟪
let
fun int-of [] = 0
  | int-of (b :: bs) = b + 2 * int-of bs;

fun bin-of (Const (num0, -)) = []
  | bin-of (Const (num1, -)) = [1]
  | bin-of (Const (bit0, -) $ bs) = 0 :: bin-of bs
  | bin-of (Const (bit1, -) $ bs) = 1 :: bin-of bs
  | bin-of t = raise TERM(bin-of, [t]);

fun bit-tr' b [t] =
  let
    val rev-digs = b :: bin-of t handle TERM - => raise Match
    val i = int-of rev-digs;
    val num = string-of-int (abs i);
  in
    Syntax.const -NumeralType $ Syntax.free num
  end
  | bit-tr' b - = raise Match;

in [(bit0, bit-tr' 0), (bit1, bit-tr' 1)] end;
⟫
```

27.8 Examples

lemma *CARD(0) = 0* **by** *simp*
lemma *CARD(17) = 17* **by** *simp*
lemma *8 * 11 ^ 3 - 6 = (2::5)* **by** *simp*

end

28 FrechetDeriv: Frechet Derivative

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

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

lemma *FDERIV-I*:
⟦*bounded-linear D*; (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *D h*) / *norm h*) −− *0* −−> *0*⟧
⟹ *FDERIV f x* :> *D*
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 ∀ *x*::*'a*. *norm* (*0*::*'b*) ≤ *norm x* ∗ *0* **by** *simp*
thus ∃ *K*. ∀ *x*::*'a*. *norm* (*0*::*'b*) ≤ *norm x* ∗ *K* **..**
qed

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

lemma *bounded-linear-ident*:
bounded-linear (λ*x*::*'a*::*real-normed-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 ∀ *x*::*'a*. *norm x* ≤ *norm x* ∗ *1* **by** *simp*
thus ∃ *K*. ∀ *x*::*'a*. *norm x* ≤ *norm x* ∗ *K* **..**
qed

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

28.1 Addition

lemma *add-diff-add*:

fixes *a b c d* :: *'a*::*ab-group-add*
shows $(a + c) - (b + d) = (a - b) + (c - d)$
by *simp*

lemma *bounded-linear-add*:
assumes *bounded-linear f*
assumes *bounded-linear g*
shows *bounded-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-normed-vector*
fixes *h* :: *'b*::*real-normed-vector*
shows *norm* $(x + y)$ / *norm h* ≤ *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* $(\lambda x.\ f\ x + g\ x)\ x$:> $(\lambda h.\ F\ h + G\ h)$
proof (*rule FDERIV-I*)
show *bounded-linear* $(\lambda h.\ F\ h + G\ h)$
apply (*rule bounded-linear-add*)
apply (*rule FDERIV-bounded-linear* [*OF f*])
apply (*rule FDERIV-bounded-linear* [*OF g*])
done
next
have *f'*: $(\lambda h.\ norm\ (f\ (x + h) - f\ x - F\ h)$ / *norm h*$)$ −− *0* −−> *0*
using *f* **by** (*rule FDERIV-D*)
have *g'*: $(\lambda h.\ norm\ (g\ (x + h) - g\ x - G\ h)$ / *norm h*$)$ −− *0* −−> *0*
using *g* **by** (*rule FDERIV-D*)

```
  from f' g'
  have (λh. 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 (λh. 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
```

28.2 Subtraction

```
lemma bounded-linear-minus:
  assumes bounded-linear f
  shows bounded-linear (λx. − 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 (λx. − f x) x :> (λh. − 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 (λx. f x − g x) x :> (λh. F h − G h)
by (simp only: diff-minus FDERIV-add FDERIV-minus)
```

28.3 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 (λh. norm (f (x + h) − f x − F h) / norm h) −− 0 −−> 0
    by (rule FDERIV-D [OF f])
  hence (λh. 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 (λh. norm (f (x + h) − f x − F h)) −− 0 −−> 0
    by (simp cong: LIM-cong)
  hence (λh. f (x + h) − f x − F h) −− 0 −−> 0
    by (rule LIM-norm-zero-cancel)
  hence (λh. f (x + h) − f x − F h + F h) −− 0 −−> 0
    by (intro LIM-add-zero F.LIM-zero LIM-ident)
  hence (λh. f (x + h) − f x) −− 0 −−> 0
    by simp
  thus isCont f x
    unfolding isCont-iff by (rule LIM-zero-cancel)
qed
```

28.4 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 (λx. 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
```

28.5 Product Rule

lemma (**in** *bounded-bilinear*) *FDERIV-lemma*:
$a' ** b' - a ** b - (a ** B + A ** b)$
$= a ** (b' - b - B) + (a' - a - A) ** b' + A ** (b' - b)$
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* $(\lambda x.\ f\ x ** g\ x)\ x$:> $(\lambda h.\ f\ x ** G\ h + F\ h ** g\ x)$
proof (*rule FDERIV-I*)
show *bounded-linear* $(\lambda h.\ f\ x ** G\ h + F\ h ** g\ x)$
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*: $\bigwedge x.\ norm\ (F\ x) \leq norm\ x * KF$ **by** *fast*

from *pos-bounded* **obtain** *K* **where** *K*: $0 < K$ **and** *norm-prod*:
$\bigwedge a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * K$ **by** *fast*

let *?Rf* $= \lambda h.\ f\ (x + h) - f\ x - F\ h$
let *?Rg* $= \lambda h.\ g\ (x + h) - g\ x - G\ h$

let *?fun1* $= \lambda h.$
norm $(f\ x ** ?Rg\ h + ?Rf\ h ** g\ (x + h) + F\ h ** (g\ (x + h) - g\ x))$ /
norm h

let *?fun2* $= \lambda h.$
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** $h \neq 0$
thus $0 \leq$ *?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
```

28.6 Powers

```
lemma FDERIV-power-Suc:
  fixes x :: 'a::{real-normed-algebra,recpower,comm-ring-1}
  shows FDERIV (λx. x ^ Suc n) x :> (λh. (1 + of-nat n) * x ^ n * h)
 apply (induct n)
  apply (simp add: power-Suc 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,recpower,comm-ring-1}
  shows FDERIV (λx. x ^ n) x :> (λh. of-nat n * x ^ (n − 1) * h)
  apply (cases n)
   apply (simp add: FDERIV-const)
  apply (simp add: FDERIV-power-Suc del: power-Suc)
  done
```

28.7 Inverse

```
lemma inverse-diff-inverse:
```

```
  ⟦(a::'a::division-ring) ≠ 0; b ≠ 0⟧
   ⟹ inverse a − inverse b = − (inverse a * (a − b) * inverse b)
by (simp add: right-diff-distrib left-diff-distrib mult-assoc)

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 ≠ 0
  shows FDERIV inverse x :> (λh. − (inverse x * h * inverse x))
      (is FDERIV ?inv - :> -)
proof (rule FDERIV-I)
  show bounded-linear (λh. − (?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 (λh. 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 ≠ 0
    assume norm (h − 0) < norm x
    hence h ≠ −x by clarsimp
    hence 2: x + h ≠ 0
      apply (rule contrapos-nn)
      apply (rule sym)
      apply (erule equals-zero-I)
      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
```

28.8 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
```

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

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

29.1 Real inner product spaces

class *real-inner* = *real-vector* + *sgn-div-norm* +
  fixes *inner* :: $'a \Rightarrow 'a \Rightarrow real$
  assumes *inner-commute*: *inner x y* = *inner y x*
  and *inner-left-distrib*: *inner* (*x* + *y*) *z* = *inner x z* + *inner y z*
  and *inner-scaleR-left*: *inner* (*scaleR r x*) *y* = *r* ∗ (*inner x y*)
  and *inner-ge-zero* [*simp*]: *0* ≤ *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-left-distrib* [*of 0 0 x*] **by** *simp*

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

Transfer distributivity rules to right argument.

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

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

lemma *inner-gt-zero-iff* [*simp*]: *0* < *inner x x* ⟷ *x* ≠ *0*

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*:
$(inner\ x\ y)^2 \leq inner\ x\ x * inner\ y\ y$
proof (*cases*)
assume $y = 0$
thus *?thesis* **by** *simp*
next
assume *y*: $y \neq 0$
let *?r* = $inner\ x\ y\ /\ inner\ y\ y$
have $0 \leq inner\ (x - scaleR\ ?r\ y)\ (x - scaleR\ ?r\ y)$
by (*rule inner-ge-zero*)
also have $\ldots = inner\ x\ x - inner\ y\ x * ?r$
by (*simp add*: *inner-diff inner-scaleR*)
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 ≤ (norm x + norm y)^2
      unfolding power2-sum power2-norm-eq-inner
      by (simp add: inner-distrib inner-commute)
    show 0 ≤ 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: inner-scaleR power2-eq-square mult-assoc)
qed

end

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-left-distrib)
 show inner x (y + z) = inner x y + inner x z
  by (rule inner-right-distrib)
 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 λx::'a::real-inner. inner x y
 by (rule inner.bounded-linear-left)

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

29.2 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

29.3 Gradient derivative

definition
gderiv ::

$['a::real\text{-}inner \Rightarrow real,\ 'a,\ 'a] \Rightarrow bool$
$((GDERIV\ (\text{-})/\ (\text{-})/\ :>\ (\text{-}))\ [1000,\ 1000,\ 60]\ 60)$
where
$GDERIV\ f\ x\ :>\ D \longleftrightarrow FDERIV\ f\ x\ :>\ (\lambda h.\ inner\ h\ D)$

lemma *deriv-fderiv*: $DERIV\ f\ x\ :>\ D \longleftrightarrow FDERIV\ f\ x\ :>\ (\lambda h.\ h * D)$
by (*simp only*: *deriv-def field-fderiv-def*)

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

lemma *GDERIV-DERIV-compose*:
$[\![GDERIV\ f\ x\ :>\ df;\ DERIV\ g\ (f\ x)\ :>\ dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ g\ (f\ x))\ x\ :>\ scaleR\ dg\ df$
unfolding *gderiv-def deriv-fderiv*
apply (*drule* (*1*) *FDERIV-compose*)
apply (*simp add*: *inner-scaleR-right mult-ac*)
done

lemma *FDERIV-subst*: $[\![FDERIV\ f\ x\ :>\ df;\ df = d]\!] \Longrightarrow FDERIV\ f\ x\ :>\ d$
by *simp*

lemma *GDERIV-subst*: $[\![GDERIV\ f\ x\ :>\ df;\ df = d]\!] \Longrightarrow GDERIV\ f\ x\ :>\ d$
by *simp*

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

lemma *GDERIV-add*:
$[\![GDERIV\ f\ x\ :>\ df;\ GDERIV\ g\ x\ :>\ dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ f\ x + g\ x)\ x\ :>\ df + dg$
unfolding *gderiv-def inner-right.add* **by** (*rule FDERIV-add*)

lemma *GDERIV-minus*:
$GDERIV\ f\ x\ :>\ df \Longrightarrow GDERIV\ (\lambda x.\ -\ f\ x)\ x\ :>\ -\ df$
unfolding *gderiv-def inner-right.minus* **by** (*rule FDERIV-minus*)

lemma *GDERIV-diff*:
$[\![GDERIV\ f\ x\ :>\ df;\ GDERIV\ g\ x\ :>\ dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ f\ x - g\ x)\ x\ :>\ df - dg$
unfolding *gderiv-def inner-right.diff* **by** (*rule FDERIV-diff*)

lemma *GDERIV-scaleR*:
$[\![DERIV\ f\ x\ :>\ df;\ GDERIV\ g\ x\ :>\ dg]\!]$
$\Longrightarrow GDERIV\ (\lambda x.\ scaleR\ (f\ x)\ (g\ x))\ x$
$:>\ (scaleR\ (f\ x)\ dg + scaleR\ df\ (g\ x))$
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 (λx. 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-distrib inner-scaleR mult-ac)
  done

lemma GDERIV-inverse:
    ⟦GDERIV f x :> df; f x ≠ 0⟧
     ⟹ GDERIV (λx. 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 (λx. norm x) x :> sgn x
proof −
  have 1: FDERIV (λx. inner x x) x :> (λh. inner x h + inner h x)
    by (intro inner.FDERIV FDERIV-ident)
  have 2: (λh. inner x h + inner h x) = (λh. inner h (scaleR 2 x))
    by (simp add: expand-fun-eq inner-scaleR 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
```

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

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

Some common special cases.

lemma *forall-1*: $(\forall i{::}1.\ P\ i) \longleftrightarrow P\ 1$
by (*metis num1-eq-iff*)

lemma *exhaust-2*:
fixes $x :: 2$ **shows** $x = 1 \vee x = 2$
proof (*induct x*)
case (*of-int z*)
then have $0 <= z$ **and** $z < 2$ **by** *simp-all*
then have $z = 0 \mid z = 1$ **by** *arith*
then show *?case* **by** *auto*
qed

lemma *forall-2*: $(\forall i{::}2.\ P\ i) \longleftrightarrow P\ 1 \wedge P\ 2$
by (*metis exhaust-2*)

lemma *exhaust-3*:
fixes $x :: 3$ **shows** $x = 1 \vee x = 2 \vee x = 3$
proof (*induct x*)
case (*of-int z*)
then have $0 <= z$ **and** $z < 3$ **by** *simp-all*
then have $z = 0 \vee z = 1 \vee z = 2$ **by** *arith*
then show *?case* **by** *auto*
qed

lemma *forall-3*: $(\forall i{::}3.\ P\ i) \longleftrightarrow P\ 1 \wedge P\ 2 \wedge P\ 3$
by (*metis exhaust-3*)

lemma *UNIV-1*: $UNIV = \{1{::}1\}$
by (*auto simp add*: *num1-eq-iff*)

lemma *UNIV-2*: $UNIV = \{1{::}2, 2{::}2\}$
using *exhaust-2* **by** *auto*

lemma *UNIV-3*: $UNIV = \{1{::}3, 2{::}3, 3{::}3\}$
using *exhaust-3* **by** *auto*

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

unfolding *UNIV-1* **by** *simp*

lemma *setsum-2*: *setsum f* (*UNIV*::*2 set*) = *f 1* + *f 2*
unfolding *UNIV-2* **by** *simp*

lemma *setsum-3*: *setsum f* (*UNIV*::*3 set*) = *f 1* + *f 2* + *f 3*
unfolding *UNIV-3* **by** (*simp add*: *add-ac*)

30.1 Basic componentwise operations on vectors.

instantiation ^ :: (*plus*,*type*) *plus*
begin
definition *vector-add-def* : *op* + ≡ (λ *x y*. (χ *i*. (*x*$*i*) + (*y*$*i*)))
instance ..
end

instantiation ^ :: (*times*,*type*) *times*
begin
definition *vector-mult-def* : *op* * ≡ (λ *x y*. (χ *i*. (*x*$*i*) * (*y*$*i*)))
instance ..
end

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

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

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

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

instance by (*intro-classes*)
end

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

Also the scalar-vector multiplication.

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

Constant Vectors

definition *vec x* = (χ *i*. *x*)

Dot products.

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

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

lemma *dot-2*[*simp*]: (*x*::*'a*::{*comm-monoid-add*, *times*} ^*2*) · *y* = (*x*$*1*) ∗ (*y*$*1*) + (*x*$*2*) ∗ (*y*$*2*)
by (*simp add*: *dot-def setsum-2*)

lemma *dot-3*[*simp*]: (*x*::*'a*::{*comm-monoid-add*, *times*} ^*3*) · *y* = (*x*$*1*) ∗ (*y*$*1*) + (*x*$*2*) ∗ (*y*$*2*) + (*x*$*3*) ∗ (*y*$*3*)
by (*simp add*: *dot-def setsum-3*)

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

method-setup *vector* = ⟪
let
val ss1 = *HOL-basic-ss addsimps* [@{*thm dot-def*}, @{*thm setsum-addf*} *RS sym*,
@{*thm setsum-subtractf*} *RS sym*, @{*thm setsum-right-distrib*},
@{*thm setsum-left-distrib*}, @{*thm setsum-negf*} *RS sym*]
val ss2 = @{*simpset*} *addsimps*
[@{*thm vector-add-def*}, @{*thm vector-mult-def*},
@{*thm vector-minus-def*}, @{*thm vector-uminus-def*},
@{*thm vector-one-def*}, @{*thm vector-zero-def*}, @{*thm vec-def*},
@{*thm vector-scaleR-def*},
@{*thm Cart-lambda-beta*}, @{*thm vector-scalar-mult-def*}]
fun vector-arith-tac ths =
simp-tac ss1
THEN' (*fn i* => *rtac* @{*thm setsum-cong2*} *i*
ORELSE rtac @{*thm setsum-0'*} *i*
ORELSE simp-tac (*HOL-basic-ss addsimps* [@{*thm Cart-eq*}]) *i*)

```
  (* THEN' TRY o clarify-tac HOL-cs  THEN' (TRY o rtac @{thm iffI}) *)
  THEN' asm-full-simp-tac (ss2 addsimps ths)
in
 Attrib.thms >> (fn ths => K (SIMPLE-METHOD' (vector-arith-tac ths)))
end
⟩ Lifts trivial vector statements to real arith statements

lemma vec-0[simp]: vec 0 = 0 by (vector vector-zero-def)
lemma vec-1[simp]: vec 1 = 1 by (vector vector-one-def)
```

Obvious "component-pushing".

```
lemma vec-component [simp]: (vec x :: 'a ^ 'n)$i = x
  by (vector vec-def)

lemma vector-add-component [simp]:
  fixes x y :: 'a::{plus} ^ 'n
  shows (x + y)$i = x$i + y$i
  by vector

lemma vector-minus-component [simp]:
  fixes x y :: 'a::{minus} ^ 'n
  shows (x - y)$i = x$i - y$i
  by vector

lemma vector-mult-component [simp]:
  fixes x y :: 'a::{times} ^ 'n
  shows (x * y)$i = x$i * y$i
  by vector

lemma vector-smult-component [simp]:
  fixes y :: 'a::{times} ^ 'n
  shows (c *s y)$i = c * (y$i)
  by vector

lemma vector-uminus-component [simp]:
  fixes x :: 'a::{uminus} ^ 'n
  shows (- x)$i = - (x$i)
  by vector

lemma vector-scaleR-component [simp]:
  fixes x :: 'a::scaleR ^ 'n
  shows (scaleR r x)$i = scaleR r (x$i)
  by vector

lemma cond-component: (if b then x else y)$i = (if b then x$i else y$i) by vector

lemmas vector-component =
  vec-component vector-add-component vector-mult-component
  vector-smult-component vector-minus-component vector-uminus-component
```

vector-scaleR-component cond-component

30.3 Some frequently useful arithmetic lemmas over vectors.

instance ^ :: (*semigroup-add*,*type*) *semigroup-add*
apply (*intro-classes*) **by** (*vector add-assoc*)

instance ^ :: (*monoid-add*,*type*) *monoid-add*
apply (*intro-classes*) **by** *vector+*

instance ^ :: (*group-add*,*type*) *group-add*
apply (*intro-classes*) **by** (*vector algebra-simps*)+

instance ^ :: (*ab-semigroup-add*,*type*) *ab-semigroup-add*
apply (*intro-classes*) **by** (*vector add-commute*)

instance ^ :: (*comm-monoid-add*,*type*) *comm-monoid-add*
apply (*intro-classes*) **by** *vector*

instance ^ :: (*ab-group-add*,*type*) *ab-group-add*
apply (*intro-classes*) **by** *vector+*

instance ^ :: (*cancel-semigroup-add*,*type*) *cancel-semigroup-add*
apply (*intro-classes*)
by (*vector Cart-eq*)+

instance ^ :: (*cancel-ab-semigroup-add*,*type*) *cancel-ab-semigroup-add*
apply (*intro-classes*)
by (*vector Cart-eq*)

instance ^ :: (*real-vector*, *type*) *real-vector*
by *default* (*vector scaleR-left-distrib scaleR-right-distrib*)+

instance ^ :: (*semigroup-mult*,*type*) *semigroup-mult*
apply (*intro-classes*) **by** (*vector mult-assoc*)

instance ^ :: (*monoid-mult*,*type*) *monoid-mult*
apply (*intro-classes*) **by** *vector+*

instance ^ :: (*ab-semigroup-mult*,*type*) *ab-semigroup-mult*
apply (*intro-classes*) **by** (*vector mult-commute*)

instance ^ :: (*ab-semigroup-idem-mult*,*type*) *ab-semigroup-idem-mult*
apply (*intro-classes*) **by** (*vector mult-idem*)

instance ^ :: (*comm-monoid-mult*,*type*) *comm-monoid-mult*
apply (*intro-classes*) **by** *vector*

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

instantiation ^ :: (*recpower*,*type*) *recpower*
begin
definition *vec-power-def*: *op* ^ ≡ *vector-power*
instance
apply (*intro-classes*) **by** (*simp-all add*: *vec-power-def*)
end

instance ^ :: (*semiring*,*type*) *semiring*
apply (*intro-classes*) **by** (*vector ring-simps*)+

instance ^ :: (*semiring-0*,*type*) *semiring-0*
apply (*intro-classes*) **by** (*vector ring-simps*)+
instance ^ :: (*semiring-1*,*type*) *semiring-1*
apply (*intro-classes*) **by** *vector*
instance ^ :: (*comm-semiring*,*type*) *comm-semiring*
apply (*intro-classes*) **by** (*vector ring-simps*)+

instance ^ :: (*comm-semiring-0*,*type*) *comm-semiring-0* **by** (*intro-classes*)
instance ^ :: (*cancel-comm-monoid-add*, *type*) *cancel-comm-monoid-add* **..**
instance ^ :: (*semiring-0-cancel*,*type*) *semiring-0-cancel* **by** (*intro-classes*)
instance ^ :: (*comm-semiring-0-cancel*,*type*) *comm-semiring-0-cancel* **by** (*intro-classes*)
instance ^ :: (*ring*,*type*) *ring* **by** (*intro-classes*)
instance ^ :: (*semiring-1-cancel*,*type*) *semiring-1-cancel* **by** (*intro-classes*)
instance ^ :: (*comm-semiring-1*,*type*) *comm-semiring-1* **by** (*intro-classes*)

instance ^ :: (*ring-1*,*type*) *ring-1* **..**

instance ^ :: (*real-algebra*,*type*) *real-algebra*
apply *intro-classes*
apply (*simp-all add*: *vector-scaleR-def ring-simps*)
apply *vector*
apply *vector*
done

instance ^ :: (*real-algebra-1*,*type*) *real-algebra-1* **..**

lemma *of-nat-index*:
(*of-nat n* :: '*a*::*semiring-1* ^'*n*)$*i* = *of-nat n*
apply (*induct n*)
apply *vector*
apply *vector*
done
lemma *zero-index*[*simp*]:
(*0* :: '*a*::*zero* ^'*n*)$*i* = *0* **by** *vector*

lemma *one-index*[*simp*]:
(*1* :: *'a*::*one* ^*'n*)$*i* = *1* **by** *vector*

lemma *one-plus-of-nat-neq-0*: (*1*::*'a*::*semiring-char-0*) + *of-nat n* ≠ *0*
proof−
have (*1*::*'a*) + *of-nat n* = *0* ⟷ *of-nat 1* + *of-nat n* = (*of-nat 0* :: *'a*) **by** *simp*
also have ... ⟷ *1* + *n* = *0* **by** (*simp only*: *of-nat-add*[*symmetric*] *of-nat-eq-iff*)
finally show *?thesis* **by** *simp*
qed

instance ^ :: (*semiring-char-0*,*type*) *semiring-char-0*
proof (*intro-classes*)
fix *m n* ::*nat*
show (*of-nat m* :: *'a*^*'b*) = *of-nat n* ⟷ *m* = *n*
by (*simp add*: *Cart-eq of-nat-index*)
qed

instance ^ :: (*comm-ring-1*,*type*) *comm-ring-1* **by** *intro-classes*
instance ^ :: (*ring-char-0*,*type*) *ring-char-0* **by** *intro-classes*

lemma *vector-smult-assoc*: *a* ∗*s* (*b* ∗*s* *x*) = ((*a*::*'a*::*semigroup-mult*) ∗ *b*) ∗*s* *x*
by (*vector mult-assoc*)
lemma *vector-sadd-rdistrib*: ((*a*::*'a*::*semiring*) + *b*) ∗*s* *x* = *a* ∗*s* *x* + *b* ∗*s* *x*
by (*vector ring-simps*)
lemma *vector-add-ldistrib*: (*c*::*'a*::*semiring*) ∗*s* (*x* + *y*) = *c* ∗*s* *x* + *c* ∗*s* *y*
by (*vector ring-simps*)
lemma *vector-smult-lzero*[*simp*]: (*0*::*'a*::*mult-zero*) ∗*s* *x* = *0* **by** *vector*
lemma *vector-smult-lid*[*simp*]: (*1*::*'a*::*monoid-mult*) ∗*s* *x* = *x* **by** *vector*
lemma *vector-ssub-ldistrib*: (*c*::*'a*::*ring*) ∗*s* (*x* − *y*) = *c* ∗*s* *x* − *c* ∗*s* *y*
by (*vector ring-simps*)
lemma *vector-smult-rneg*: (*c*::*'a*::*ring*) ∗*s* −*x* = −(*c* ∗*s* *x*) **by** *vector*
lemma *vector-smult-lneg*: − (*c*::*'a*::*ring*) ∗*s* *x* = −(*c* ∗*s* *x*) **by** *vector*
lemma *vector-sneg-minus1*: −*x* = (− (*1*::*'a*::*ring-1*)) ∗*s* *x* **by** *vector*
lemma *vector-smult-rzero*[*simp*]: *c* ∗*s* *0* = (*0*::*'a*::*mult-zero* ^ *'n*) **by** *vector*
lemma *vector-sub-rdistrib*: ((*a*::*'a*::*ring*) − *b*) ∗*s* *x* = *a* ∗*s* *x* − *b* ∗*s* *x*
by (*vector ring-simps*)

lemma *vec-eq*[*simp*]: (*vec m* = *vec n*) ⟷ (*m* = *n*)
by (*simp add*: *Cart-eq*)

30.4 Square root of sum of squares

definition
setL2 f A = *sqrt* (∑ *i*∈*A*. (*f i*)^2)

lemma *setL2-cong*:
⟦*A* = *B*; ⋀*x*. *x* ∈ *B* ⟹ *f x* = *g x*⟧ ⟹ *setL2 f A* = *setL2 g B*
unfolding *setL2-def* **by** *simp*

lemma *strong-setL2-cong*:
$[\![A = B;\ \bigwedge x.\ x \in B \Longrightarrow f\ x = g\ x]\!] \Longrightarrow$ *setL2 f A = setL2 g B*
unfolding *setL2-def simp-implies-def* **by** *simp*

lemma *setL2-infinite* [*simp*]: $\neg$ *finite A* $\Longrightarrow$ *setL2 f A = 0*
unfolding *setL2-def* **by** *simp*

lemma *setL2-empty* [*simp*]: *setL2 f {} = 0*
unfolding *setL2-def* **by** *simp*

lemma *setL2-insert* [*simp*]:
$[\![$*finite F*; *a* $\notin$ *F*$]\!] \Longrightarrow$
setL2 f (insert a F) = sqrt ((f a)² + (setL2 f F)²)
unfolding *setL2-def* **by** (*simp add*: *setsum-nonneg*)

lemma *setL2-nonneg* [*simp*]: *0* $\leq$ *setL2 f A*
unfolding *setL2-def* **by** (*simp add*: *setsum-nonneg*)

lemma *setL2-0'*: $\forall$ *a*$\in$*A. f a = 0* $\Longrightarrow$ *setL2 f A = 0*
unfolding *setL2-def* **by** *simp*

lemma *setL2-mono*:
assumes $\bigwedge i.\ i \in K \Longrightarrow f\ i \leq g\ i$
assumes $\bigwedge i.\ i \in K \Longrightarrow 0 \leq f\ i$
shows *setL2 f K* $\leq$ *setL2 g K*
unfolding *setL2-def*
by (*simp add*: *setsum-nonneg setsum-mono power-mono prems*)

lemma *setL2-right-distrib*:
0 $\leq$ *r* $\Longrightarrow$ *r * setL2 f A = setL2* $(\lambda x.\ r * f\ x)$ *A*
unfolding *setL2-def*
apply (*simp add*: *power-mult-distrib*)
apply (*simp add*: *setsum-right-distrib* [*symmetric*])
apply (*simp add*: *real-sqrt-mult setsum-nonneg*)
done

lemma *setL2-left-distrib*:
0 $\leq$ *r* $\Longrightarrow$ *setL2 f A * r = setL2* $(\lambda x.\ f\ x * r)$ *A*
unfolding *setL2-def*
apply (*simp add*: *power-mult-distrib*)
apply (*simp add*: *setsum-left-distrib* [*symmetric*])
apply (*simp add*: *real-sqrt-mult setsum-nonneg*)
done

lemma *setsum-nonneg-eq-0-iff*:
fixes *f* :: *'a* $\Rightarrow$ *'b*::*pordered-ab-group-add*
shows $[\![$*finite A*; $\forall x \in A.\ 0 \leq f\ x$$]\!] \Longrightarrow$ *setsum f A = 0* $\longleftrightarrow$ $(\forall x \in A.\ f\ x = 0)$
apply (*induct set*: *finite*, *simp*)
apply (*simp add*: *add-nonneg-eq-0-iff setsum-nonneg*)

done

lemma *setL2-eq-0-iff*: *finite A* $\Longrightarrow$ *setL2 f A = 0* $\longleftrightarrow$ $(\forall x{\in}A.\ f\ x = 0)$
unfolding *setL2-def*
by (*simp add*: *setsum-nonneg setsum-nonneg-eq-0-iff*)

lemma *setL2-triangle-ineq*:
shows *setL2* $(\lambda i.\ f\ i + g\ i)\ A \leq$ *setL2 f A + setL2 g A*
proof (*cases finite A*)
case *False*
thus *?thesis* **by** *simp*
next
case *True*
thus *?thesis*
proof (*induct set*: *finite*)
case *empty*
show *?case* **by** *simp*
next
case (*insert x F*)
hence *sqrt* $((f\ x + g\ x)^2 + (setL2\ (\lambda i.\ f\ i + g\ i)\ F)^2) \leq$
sqrt $((f\ x + g\ x)^2 + (setL2\ f\ F + setL2\ g\ F)^2)$
by (*intro real-sqrt-le-mono add-left-mono power-mono insert*
setL2-nonneg add-increasing zero-le-power2)
also have
$\ldots \leq$ *sqrt* $((f\ x)^2 + (setL2\ f\ F)^2)$ + *sqrt* $((g\ x)^2 + (setL2\ g\ F)^2)$
by (*rule real-sqrt-sum-squares-triangle-ineq*)
finally show *?case*
using *insert* **by** *simp*
qed
qed

lemma *sqrt-sum-squares-le-sum*:
$[\![0 \leq x;\ 0 \leq y]\!] \Longrightarrow$ *sqrt* $(x^2 + y^2) \leq x + y$
apply (*rule power2-le-imp-le*)
apply (*simp add*: *power2-sum*)
apply (*simp add*: *mult-nonneg-nonneg*)
apply (*simp add*: *add-nonneg-nonneg*)
done

lemma *setL2-le-setsum* [*rule-format*]:
$(\forall i{\in}A.\ 0 \leq f\ i) \longrightarrow$ *setL2 f A* $\leq$ *setsum f A*
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply *simp*
apply *clarsimp*
apply (*erule order-trans* [*OF sqrt-sum-squares-le-sum*])
apply *simp*
apply *simp*
apply *simp*

done

lemma *sqrt-sum-squares-le-sum-abs*: *sqrt* $(x^2 + y^2) \leq |x| + |y|$
apply (*rule power2-le-imp-le*)
apply (*simp add*: *power2-sum*)
apply (*simp add*: *mult-nonneg-nonneg*)
apply (*simp add*: *add-nonneg-nonneg*)
done

lemma *setL2-le-setsum-abs*: *setL2 f A* $\leq (\sum i{\in}A.\ |f\ i|)$
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply *simp*
apply *simp*
apply (*rule order-trans* [*OF sqrt-sum-squares-le-sum-abs*])
apply *simp*
apply *simp*
done

lemma *setL2-mult-ineq-lemma*:
fixes *a b c d* :: *real*
shows $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$
proof −
have $0 \leq (a * d - b * c)^2$ **by** *simp*
also have $\ldots = a^2 * d^2 + b^2 * c^2 - 2 * (a * d) * (b * c)$
by (*simp only*: *power2-diff power-mult-distrib*)
also have $\ldots = a^2 * d^2 + b^2 * c^2 - 2 * (a * c) * (b * d)$
by *simp*
finally show $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$
by *simp*
qed

lemma *setL2-mult-ineq*: $(\sum i{\in}A.\ |f\ i| * |g\ i|) \leq$ *setL2 f A* $*$ *setL2 g A*
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply *simp*
apply (*rule power2-le-imp-le*, *simp*)
apply (*rule order-trans*)
apply (*rule power-mono*)
apply (*erule add-left-mono*)
apply (*simp add*: *add-nonneg-nonneg mult-nonneg-nonneg setsum-nonneg*)
apply (*simp add*: *power2-sum*)
apply (*simp add*: *power-mult-distrib*)
apply (*simp add*: *right-distrib left-distrib*)
apply (*rule ord-le-eq-trans*)
apply (*rule setL2-mult-ineq-lemma*)
apply *simp*
apply (*intro mult-nonneg-nonneg setL2-nonneg*)
apply *simp*

done

lemma *member-le-setL2*: ⟦*finite A*; $i \in A$⟧ ⟹ $f\ i \leq$ *setL2 f A*
apply (*rule-tac s=insert i* $(A - \{i\})$ **and** *t=A* **in** *subst*)
apply *fast*
apply (*subst setL2-insert*)
apply *simp*
apply *simp*
apply *simp*
done

30.5 Norms

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

definition *vector-norm-def*:
norm (*x*::'*a*^'*b*) = *setL2* (λ*i*. *norm* (*x*$*i*)) *UNIV*

definition *vector-sgn-def*:
sgn (*x*::'*a*^'*b*) = *scaleR* (*inverse* (*norm x*)) *x*

instance proof
fix *a* :: *real* **and** *x y* :: '*a* ^ '*b*
show $0 \leq$ *norm x*
unfolding *vector-norm-def*
by (*rule setL2-nonneg*)
show *norm x* = *0* ⟷ *x* = *0*
unfolding *vector-norm-def*
by (*simp add*: *setL2-eq-0-iff Cart-eq*)
show *norm* (*x* + *y*) ≤ *norm x* + *norm y*
unfolding *vector-norm-def*
apply (*rule order-trans* [*OF - setL2-triangle-ineq*])
apply (*simp add*: *setL2-mono norm-triangle-ineq*)
done
show *norm* (*scaleR a x*) = |*a*| * *norm x*
unfolding *vector-norm-def*
by (*simp add*: *norm-scaleR setL2-right-distrib*)
show *sgn x* = *scaleR* (*inverse* (*norm x*)) *x*
by (*rule vector-sgn-def*)
qed

end

30.6 Inner products

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

definition *vector-inner-def*:

inner x y = setsum (λi. inner (x$i) (y$i)) UNIV

instance proof
fix *r* :: *real* **and** *x y z* :: *'a ^ 'b*
show *inner x y = inner y x*
unfolding *vector-inner-def*
by (*simp add*: *inner-commute*)
show *inner (x + y) z = inner x z + inner y z*
unfolding *vector-inner-def*
by (*simp add*: *inner-left-distrib setsum-addf*)
show *inner (scaleR r x) y = r * inner x y*
unfolding *vector-inner-def*
by (*simp add*: *inner-scaleR-left setsum-right-distrib*)
show *0 ≤ inner x x*
unfolding *vector-inner-def*
by (*simp add*: *setsum-nonneg*)
show *inner x x = 0 ⟷ x = 0*
unfolding *vector-inner-def*
by (*simp add*: *Cart-eq setsum-nonneg-eq-0-iff*)
show *norm x = sqrt (inner x x)*
unfolding *vector-inner-def vector-norm-def setL2-def*
by (*simp add*: *power2-norm-eq-inner*)
qed

end

30.7 Properties of the dot product.

lemma *dot-sym*: (*x*::*'a*:: {*comm-monoid-add*, *ab-semigroup-mult*} ^ *'n*) · *y* = *y* · *x*
by (*vector mult-commute*)
lemma *dot-ladd*: ((*x*::*'a*::*ring* ^ *'n*) + *y*) · *z* = (*x* · *z*) + (*y* · *z*)
by (*vector ring-simps*)
lemma *dot-radd*: *x* · (*y* + (*z*::*'a*::*ring* ^ *'n*)) = (*x* · *y*) + (*x* · *z*)
by (*vector ring-simps*)
lemma *dot-lsub*: ((*x*::*'a*::*ring* ^ *'n*) − *y*) · *z* = (*x* · *z*) − (*y* · *z*)
by (*vector ring-simps*)
lemma *dot-rsub*: (*x*::*'a*::*ring* ^ *'n*) · (*y* − *z*) = (*x* · *y*) − (*x* · *z*)
by (*vector ring-simps*)
lemma *dot-lmult*: (*c* *s *x*) · *y* = (*c*::*'a*::*ring*) * (*x* · *y*) **by** (*vector ring-simps*)
lemma *dot-rmult*: *x* · (*c* *s *y*) = (*c*::*'a*::*comm-ring*) * (*x* · *y*) **by** (*vector ring-simps*)
lemma *dot-lneg*: (−*x*) · (*y*::*'a*::*ring* ^ *'n*) = −(*x* · *y*) **by** *vector*
lemma *dot-rneg*: (*x*::*'a*::*ring* ^ *'n*) · (−*y*) = −(*x* · *y*) **by** *vector*
lemma *dot-lzero*[*simp*]: *0* · *x* = (*0*::*'a*::{*comm-monoid-add*, *mult-zero*}) **by** *vector*
lemma *dot-rzero*[*simp*]: *x* · *0* = (*0*::*'a*::{*comm-monoid-add*, *mult-zero*}) **by** *vector*
lemma *dot-pos-le*[*simp*]: (*0*::*'a*::*ordered-ring-strict*) <= *x* · *x*
by (*simp add*: *dot-def setsum-nonneg*)

lemma *setsum-squares-eq-0-iff*: **assumes** *fS*: *finite F* **and** *fp*: ∀ *x* ∈ *F*. *f x* ≥ (*0*

::'a::pordered-ab-group-add) **shows** *setsum f F = 0 ⟷ (ALL x:F. f x = 0)*
using *fS fp setsum-nonneg[OF fp]*
proof (*induct set*: *finite*)
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x F*)
from *insert.prems* **have** *Fx*: *f x ≥ 0* **and** *Fp*: *∀ a ∈ F. f a ≥ 0* **by** *simp-all*
from *insert.hyps Fp setsum-nonneg[OF Fp]*
have *h*: *setsum f F = 0 ⟷ (∀ a ∈F. f a = 0)* **by** *metis*
from *sum-nonneg-eq-zero-iff[OF Fx setsum-nonneg[OF Fp]] insert.hyps(1,2)*
show *?case* **by** (*simp add*: *h*)
qed

lemma *dot-eq-0*: *x · x = 0 ⟷ (x::'a::{ordered-ring-strict,ring-no-zero-divisors}*
^ 'n::finite) = 0
by (*simp add*: *dot-def setsum-squares-eq-0-iff Cart-eq*)

lemma *dot-pos-lt[simp]*: *(0 < x · x) ⟷ (x::'a::{ordered-ring-strict,ring-no-zero-divisors}*
^ 'n::finite) ≠ 0 **using** *dot-eq-0[of x] dot-pos-le[of x]*
by (*auto simp add*: *le-less*)

30.8 The collapse of the general concepts to dimension one.

lemma *vector-one*: *(x::'a ^1) = (χ i. (x$1))*
by (*simp add*: *Cart-eq forall-1*)

lemma *forall-one*: *(∀ (x::'a ^1). P x) ⟷ (∀ x. P(χ i. x))*
apply *auto*
apply (*erule-tac x= x$1* **in** *allE*)
apply (*simp only*: *vector-one[symmetric]*)
done

lemma *norm-vector-1*: *norm (x :: -^1) = norm (x$1)*
by (*simp add*: *vector-norm-def UNIV-1*)

lemma *norm-real*: *norm(x::real ^ 1) = abs(x$1)*
by (*simp add*: *norm-vector-1*)

Metric

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

definition *dist*:: *real ^ 'n::finite ⇒ real ^ 'n ⇒ real* **where**
dist x y = norm (x − y)

lemma *dist-real*: *dist(x::real ^ 1) y = abs((x$1) − (y$1))*
by (*auto simp add*: *norm-real dist-def*)

30.9 A connectedness or intermediate value lemma with several applications.

lemma *connected-real-lemma*:
fixes f :: *real* ⇒ *real* ^ 'n::*finite*
assumes *ab*: $a \leq b$ **and** *fa*: $f\ a \in e1$ **and** *fb*: $f\ b \in e2$
and *dst*: $\bigwedge e\ x.\ a <= x \Longrightarrow x <= b \Longrightarrow 0 < e ==> \exists\, d > 0.\ \forall\, y.\ abs(y - x) < d \longrightarrow dist(f\ y)\ (f\ x) < e$
and *e1*: $\forall\, y \in e1.\ \exists\, e > 0.\ \forall\, y'.\ dist\ y'\ y < e \longrightarrow y' \in e1$
and *e2*: $\forall\, y \in e2.\ \exists\, e > 0.\ \forall\, y'.\ dist\ y'\ y < e \longrightarrow y' \in e2$
and *e12*: $\sim(\exists\, x \geq a.\ x <= b \wedge f\ x \in e1 \wedge f\ x \in e2)$
shows $\exists\, x \geq a.\ x <= b \wedge f\ x \notin e1 \wedge f\ x \notin e2$ (**is** $\exists\ x.\ ?P\ x$)
proof−
let *?S* = $\{c.\ \forall\, x \geq a.\ x <= c \longrightarrow f\ x \in e1\}$
have *Se*: $\exists\, x.\ x \in ?S$ **apply** (*rule exI*[**where** x=a]) **by** (*auto simp add*: *fa*)
have *Sub*: $\exists\, y.$ *isUb UNIV ?S y*
apply (*rule exI*[**where** x= b])
using *ab fb e12* **by** (*auto simp add*: *isUb-def setle-def*)
from *reals-complete*[*OF Se Sub*] **obtain** l **where**
l: *isLub UNIV ?S l***by** *blast*
have *alb*: $a \leq l\ l \leq b$ **using** *l ab fa fb e12*
apply (*auto simp add*: *isLub-def leastP-def isUb-def setle-def setge-def*)
by (*metis linorder-linear*)
have *ale1*: $\forall\, z \geq a.\ z < l \longrightarrow f\ z \in e1$ **using** *l*
apply (*auto simp add*: *isLub-def leastP-def isUb-def setle-def setge-def*)
by (*metis linorder-linear not-le*)
have *th1*: $\bigwedge z\ x\ e\ d$ :: *real*. $z <= x + e \Longrightarrow e < d ==> z < x \vee abs(z - x) < d$ **by** *arith*
have *th2*: $\bigwedge e\ x$:: *real*. $0 < e ==> \sim(x + e <= x)$ **by** *arith*
have *th3*: $\bigwedge d$::*real*. $d > 0 \Longrightarrow \exists\, e > 0.\ e < d$ **by** *dlo*
{assume *le2*: $f\ l \in e2$
from *le2 fa fb e12 alb* **have** *la*: $l \neq a$ **by** *metis*
hence *lap*: $l - a > 0$ **using** *alb* **by** *arith*
from *e2*[*rule-format*, *OF le2*] **obtain** e **where**
e: $e > 0\ \forall\, y.\ dist\ y\ (f\ l) < e \longrightarrow y \in e2$ **by** *metis*
from *dst*[*OF alb e*(*1*)] **obtain** d **where**
d: $d > 0\ \forall\, y.\ |y - l| < d \longrightarrow dist\ (f\ y)\ (f\ l) < e$ **by** *metis*
have $\exists\, d'.\ d' < d \wedge d' > 0 \wedge l - d' > a$ **using** *lap d*(*1*)
apply *ferrack* **by** *arith*
then obtain d' **where** d': $d' > 0\ d' < d\ l - d' > a$ **by** *metis*
from d e **have** *th0*: $\forall\, y.\ |y - l| < d \longrightarrow f\ y \in e2$ **by** *metis*
from *th0*[*rule-format*, *of* $l - d'$] d' **have** $f\ (l - d') \in e2$ **by** *auto*
moreover
have $f\ (l - d') \in e1$ **using** *ale1*[*rule-format*, *of* $l - d'$] d' **by** *auto*
ultimately have *False* **using** *e12 alb* d' **by** *auto*}
moreover
{assume *le1*: $f\ l \in e1$
from *le1 fa fb e12 alb* **have** *lb*: $l \neq b$ **by** *metis*
hence *blp*: $b - l > 0$ **using** *alb* **by** *arith*
from *e1*[*rule-format*, *OF le1*] **obtain** e **where**

e: $e > 0 \; \forall y.\; dist\ y\ (f\ l) < e \longrightarrow y \in e1$ **by** *metis*
from *dst*[*OF alb e*(*1*)] **obtain** *d* **where**
d: $d > 0 \; \forall y.\; |y - l| < d \longrightarrow dist\ (f\ y)\ (f\ l) < e$ **by** *metis*
have $\exists d'.\; d' < d \wedge d' > 0$ **using** *d*(*1*) **by** *dlo*
then obtain d' **where** d': $d' > 0\ d' < d$ **by** *metis*
from *d e* **have** *th0*: $\forall y.\; |y - l| < d \longrightarrow f\ y \in e1$ **by** *auto*
hence $\forall y.\; l \leq y \wedge y \leq l + d' \longrightarrow f\ y \in e1$ **using** d' **by** *auto*
with *ale1* **have** $\forall y.\; a \leq y \wedge y \leq l + d' \longrightarrow f\ y \in e1$ **by** *auto*
with *l* d' **have** *False*
by (*auto simp add*: *isLub-def isUb-def setle-def setge-def leastP-def*) **}**
ultimately show *?thesis* **using** *alb* **by** *metis*
qed

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

lemma *square-bound-lemma*: $(x::real) < (1 + x) * (1 + x)$
proof−
have $(x + 1/2)\hat{}2 + 3/4 > 0$ **using** *zero-le-power2*[*of x+1/2*] **by** *arith*
thus *?thesis* **by** (*simp add*: *ring-simps power2-eq-square*)
qed

lemma *square-continuous*: $0 < (e::real) ==> \exists d.\; 0 < d \wedge (\forall y.\; abs(y - x) < d \longrightarrow abs(y * y - x * x) < e)$
using *isCont-power*[*OF isCont-ident*, *of 2*, *unfolded isCont-def LIM-def*, *rule-format*, *of e x*] **apply** (*auto simp add*: *power2-eq-square*)
apply (*rule-tac x=s* **in** *exI*)
apply *auto*
apply (*erule-tac x=y* **in** *allE*)
apply *auto*
done

lemma *real-le-lsqrt*: $0 <= x \Longrightarrow 0 <= y \Longrightarrow x <= y\hat{}2 ==> sqrt\ x <= y$
using *real-sqrt-le-iff*[*of x y^2*] **by** *simp*

lemma *real-le-rsqrt*: $x\hat{}2 \leq y \Longrightarrow x \leq sqrt\ y$
using *real-sqrt-le-mono*[*of x^2 y*] **by** *simp*

lemma *real-less-rsqrt*: $x\hat{}2 < y \Longrightarrow x < sqrt\ y$
using *real-sqrt-less-mono*[*of x^2 y*] **by** *simp*

lemma *sqrt-even-pow2*: **assumes** *n*: *even n*
shows $sqrt(2\ \hat{}\ n) = 2\ \hat{}\ (n\ div\ 2)$
proof−
from *n* **obtain** *m* **where** *m*: $n = 2*m$ **unfolding** *even-nat-equiv-def2*
by (*auto simp add*: *nat-number*)
from *m* **have** $sqrt(2\ \hat{}\ n) = sqrt\ ((2\ \hat{}\ m)\ \hat{}\ 2)$
by (*simp only*: *power-mult*[*symmetric*] *mult-commute*)
then show *?thesis* **using** *m* **by** *simp*

qed

lemma *real-div-sqrt*: *0 <= x ==> x / sqrt(x) = sqrt(x)*
apply (*cases x = 0*, *simp-all*)
using *sqrt-divide-self-eq*[*of x*]
apply (*simp add*: *inverse-eq-divide real-sqrt-ge-0-iff field-simps*)
done

Hence derive more interesting properties of the norm.

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

lemma *norm-0*: *norm* (*0*::*real* ^ -) = *0*
by (*rule norm-zero*)

lemma *norm-mul*[*simp*]: *norm*(*a* *s *x*) = *abs*(*a*) * *norm x*
by (*simp add*: *vector-norm-def vector-component setL2-right-distrib*
abs-mult cong: *strong-setL2-cong*)
lemma *norm-eq-0-dot*: (*norm x = 0*) ⟷ (*x* • *x* = (*0*::*real*))
by (*simp add*: *vector-norm-def dot-def setL2-def power2-eq-square*)
lemma *real-vector-norm-def*: *norm x* = *sqrt* (*x* • *x*)
by (*simp add*: *vector-norm-def setL2-def dot-def power2-eq-square*)
lemma *norm-pow-2*: *norm x* ^ *2* = *x* • *x*
by (*simp add*: *real-vector-norm-def*)
lemma *norm-eq-0-imp*: *norm x = 0 ==> x* = (*0*::*real* ^'*n*::*finite*) **by** (*metis*
norm-eq-zero)
lemma *vector-mul-eq-0*[*simp*]: (*a* *s *x = 0*) ⟷ *a* = (*0*::'*a*::*idom*) ∨ *x = 0*
by *vector*
lemma *vector-mul-lcancel*[*simp*]: *a* *s *x* = *a* *s *y* ⟷ *a* = (*0*::*real*) ∨ *x* = *y*
by (*metis eq-iff-diff-eq-0 vector-mul-eq-0 vector-ssub-ldistrib*)
lemma *vector-mul-rcancel*[*simp*]: *a* *s *x* = *b* *s *x* ⟷ (*a*::*real*) = *b* ∨ *x = 0*
by (*metis eq-iff-diff-eq-0 vector-mul-eq-0 vector-sub-rdistrib*)
lemma *vector-mul-lcancel-imp*: *a* ≠ (*0*::*real*) ==> *a* *s *x* = *a* *s *y ==>* (*x* =
y)
by (*metis vector-mul-lcancel*)
lemma *vector-mul-rcancel-imp*: *x* ≠ *0* ⟹ (*a*::*real*) *s *x* = *b* *s *x ==> a* = *b*
by (*metis vector-mul-rcancel*)
lemma *norm-cauchy-schwarz*:
fixes *x y* :: *real* ^ '*n*::*finite*
shows *x* • *y <= norm x* * *norm y*
proof−
{assume *norm x = 0*
hence *?thesis* **by** (*simp add*: *dot-lzero dot-rzero*)**}**
moreover
{assume *norm y = 0*
hence *?thesis* **by** (*simp add*: *dot-lzero dot-rzero*)**}**
moreover
{assume *h*: *norm x* ≠ *0 norm y* ≠ *0*
let *?z = norm y* *s *x* − *norm x* *s *y*
from *h* **have** *p*: *norm x* * *norm y* > *0* **by** (*metis norm-ge-zero le-less zero-compare-simps*)
from *dot-pos-le*[*of ?z*]

```
    have (norm x * norm y) * (x · y) ≤ norm x ^2 * norm y ^2
      apply (simp add: dot-rsub dot-lsub dot-lmult dot-rmult ring-simps)
      by (simp add: norm-pow-2[symmetric] power2-eq-square dot-sym)
    hence x·y ≤ (norm x ^2 * norm y ^2) / (norm x * norm y) using p
      by (simp add: field-simps)
    hence ?thesis using h by (simp add: power2-eq-square)}
  ultimately show ?thesis by metis
qed

lemma norm-cauchy-schwarz-abs:
  fixes x y :: real ^ 'n::finite
  shows |x · y| ≤ norm x * norm y
  using norm-cauchy-schwarz[of x y] norm-cauchy-schwarz[of x -y]
  by (simp add: real-abs-def dot-rneg)

lemma norm-triangle-sub: norm (x::real ^'n::finite) <= norm(y) + norm(x - y)
  using norm-triangle-ineq[of y x - y] by (simp add: ring-simps)
lemma norm-triangle-le: norm(x::real ^'n::finite) + norm y <= e ==> norm(x
+ y) <= e
  by (metis order-trans norm-triangle-ineq)
lemma norm-triangle-lt: norm(x::real ^'n::finite) + norm(y) < e ==> norm(x
+ y) < e
  by (metis basic-trans-rules(21) norm-triangle-ineq)

lemma setsum-delta:
  assumes fS: finite S
  shows setsum (λk. if k=a then b k else 0) S = (if a ∈ S then b a else 0)
proof-
  let ?f = (λk. if k=a then b k else 0)
  {assume a: a ∉ S
    hence ∀ k∈ S. ?f k = 0 by simp
    hence ?thesis  using a by simp}
  moreover
  {assume a: a ∈ S
    let ?A = S - {a}
    let ?B = {a}
    have eq: S = ?A ∪ ?B using a by blast
    have dj: ?A ∩ ?B = {} by simp
    from fS have fAB: finite ?A finite ?B by auto
    have setsum ?f S = setsum ?f ?A + setsum ?f ?B
      using setsum-Un-disjoint[OF fAB dj, of ?f, unfolded eq[symmetric]]
      by simp
    then have ?thesis  using a by simp}
  ultimately show ?thesis by blast
qed

lemma component-le-norm: |x$i| <= norm (x::real ^ 'n::finite)
  apply (simp add: vector-norm-def)
  apply (rule member-le-setL2, simp-all)
```

done

lemma *norm-bound-component-le*: *norm*(*x*::*real* ^ *'n*::*finite*) <= *e*
==> |*x*$*i*| <= *e*
by (*metis component-le-norm order-trans*)

lemma *norm-bound-component-lt*: *norm*(*x*::*real* ^ *'n*::*finite*) < *e*
==> |*x*$*i*| < *e*
by (*metis component-le-norm basic-trans-rules*(*21*))

lemma *norm-le-l1*: *norm* (*x*:: *real* ^*'n*::*finite*) <= *setsum*(λ*i*. |*x*$*i*|) *UNIV*
by (*simp add*: *vector-norm-def setL2-le-setsum*)

lemma *real-abs-norm*: |*norm x*| = *norm* (*x* :: *real* ^ -)
by (*rule abs-norm-cancel*)
lemma *real-abs-sub-norm*: |*norm*(*x*::*real* ^*'n*::*finite*) − *norm y*| <= *norm*(*x* − *y*)
by (*rule norm-triangle-ineq3*)
lemma *norm-le*: *norm*(*x*::*real* ^ -) <= *norm*(*y*) ⟷ *x* · *x* <= *y* · *y*
by (*simp add*: *real-vector-norm-def*)
lemma *norm-lt*: *norm*(*x*::*real* ^ -) < *norm*(*y*) ⟷ *x* · *x* < *y* · *y*
by (*simp add*: *real-vector-norm-def*)
lemma *norm-eq*: *norm* (*x*::*real* ^ -) = *norm y* ⟷ *x* · *x* = *y* · *y*
by (*simp add*: *order-eq-iff norm-le*)
lemma *norm-eq-1*: *norm*(*x*::*real* ^ -) = *1* ⟷ *x* · *x* = *1*
by (*simp add*: *real-vector-norm-def*)

Squaring equations and inequalities involving norms.

lemma *dot-square-norm*: *x* · *x* = *norm*(*x*) ^*2*
by (*simp add*: *real-vector-norm-def*)

lemma *norm-eq-square*: *norm*(*x*) = *a* ⟷ *0* <= *a* ∧ *x* · *x* = *a*^*2*
by (*auto simp add*: *real-vector-norm-def*)

lemma *real-abs-le-square-iff*: |*x*| ≤ |*y*| ⟷ (*x*::*real*) ^*2* ≤ *y*^*2*
proof−
have *x*^*2* ≤ *y*^*2* ⟷ (*x* −*y*) ∗ (*y* + *x*) ≤ *0* **by** (*simp add*: *ring-simps power2-eq-square*)
also have ... ⟷ |*x*| ≤ |*y*| **apply** (*simp add*: *zero-compare-simps real-abs-def not-less*) **by** *arith*
finally show *?thesis* **..**
qed

lemma *norm-le-square*: *norm*(*x*) <= *a* ⟷ *0* <= *a* ∧ *x* · *x* <= *a*^*2*
apply (*simp add*: *dot-square-norm real-abs-le-square-iff*[*symmetric*])
using *norm-ge-zero*[*of x*]
apply *arith*
done

lemma *norm-ge-square*: *norm*(*x*) >= *a* ⟷ *a* <= *0* ∨ *x* · *x* >= *a* ^ *2*
apply (*simp add*: *dot-square-norm real-abs-le-square-iff*[*symmetric*])

using *norm-ge-zero*[*of x*]
apply *arith*
done

lemma *norm-lt-square*: $norm(x) < a \longleftrightarrow 0 < a \wedge x \cdot x < a\hat{}2$
by (*metis not-le norm-ge-square*)
lemma *norm-gt-square*: $norm(x) > a \longleftrightarrow a < 0 \vee x \cdot x > a\hat{}2$
by (*metis norm-le-square not-less*)

Dot product in terms of the norm rather than conversely.

lemma *dot-norm*: $x \cdot y = (norm(x + y)\ \hat{}2 - norm\ x\ \hat{}\ 2 - norm\ y\ \hat{}\ 2) / 2$
by (*simp add*: *norm-pow-2 dot-ladd dot-radd dot-sym*)

lemma *dot-norm-neg*: $x \cdot y = ((norm\ x\ \hat{}\ 2 + norm\ y\ \hat{}\ 2) - norm(x - y)\ \hat{}\ 2) / 2$
by (*simp add*: *norm-pow-2 dot-ladd dot-radd dot-lsub dot-rsub dot-sym*)

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

lemma *vector-eq*: $(x::\ real\ \hat{}\ 'n::finite) = y \longleftrightarrow x \cdot x = x \cdot y \wedge y \cdot y = x \cdot x$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs* **then show** *?rhs* **by** *simp*
next
assume *?rhs*
then have $x \cdot x - x \cdot y = 0 \wedge x \cdot y - y \cdot y = 0$ **by** *simp*
hence $x \cdot (x - y) = 0 \wedge y \cdot (x - y) = 0$
by (*simp add*: *dot-rsub dot-lsub dot-sym*)
then have $(x - y) \cdot (x - y) = 0$ **by** (*simp add*: *ring-simps dot-lsub dot-rsub*)
then show $x = y$ **by** (*simp add*: *dot-eq-0*)
qed

30.10 General linear decision procedure for normed spaces.

lemma *norm-cmul-rule-thm*: $b >= norm(x) ==> |c| * b >= norm(c *s x)$
apply (*clarsimp simp add*: *norm-mul*)
apply (*rule mult-mono1*)
apply *simp-all*
done

lemma *norm-add-rule-thm*: $b1 >= norm(x1 :: real\ \hat{}\ 'n::finite) \Longrightarrow b2 >= norm(x2) ==> b1 + b2 >= norm(x1 + x2)$
apply (*rule norm-triangle-le*) **by** *simp*

lemma *ge-iff-diff-ge-0*: $(a::'a::ordered\text{-}ring) \geq b == a - b \geq 0$
by (*simp add*: *ring-simps*)

lemma *pth-1*: $(x::real\hat{}\ 'n) == 1 *s x$ **by** (*simp only*: *vector-smult-lid*)
lemma *pth-2*: $x - (y::real\hat{}\ 'n) == x + -y$ **by** (*atomize* (*full*)) *simp*
lemma *pth-3*: $(-x::real\hat{}\ 'n) == -1 *s x$ **by** *vector*

lemma *pth-4*: $0 *s (x::real\hat{}'n) == 0\ c *s 0 = (0::real\ \hat{}\ 'n)$ **by** *vector+*
lemma *pth-5*: $c *s (d *s x) == (c * d) *s (x::real\ \hat{}\ 'n)$ **by** (*atomize* (*full*)) *vector*
lemma *pth-6*: $(c::real) *s (x + y) == c *s x + c *s y$ **by** (*atomize* (*full*)) (*vector ring-simps*)
lemma *pth-7*: $0 + x == (x::real\hat{}'n)\ x + 0 == x$ **by** *simp-all*
lemma *pth-8*: $(c::real) *s x + d *s x == (c + d) *s x$ **by** (*atomize* (*full*)) (*vector ring-simps*)
lemma *pth-9*: $((c::real) *s x + z) + d *s x == (c + d) *s x + z$
$c *s x + (d *s x + z) == (c + d) *s x + z$
$(c *s x + w) + (d *s x + z) == (c + d) *s x + (w + z)$ **by** ((*atomize* (*full*)), *vector ring-simps*)+
lemma *pth-a*: $(0::real) *s x + y == y$ **by** (*atomize* (*full*)) *vector*
lemma *pth-b*: $(c::real) *s x + d *s y == c *s x + d *s y$
$(c *s x + z) + d *s y == c *s x + (z + d *s y)$
$c *s x + (d *s y + z) == c *s x + (d *s y + z)$
$(c *s x + w) + (d *s y + z) == c *s x + (w + (d *s y + z))$
by ((*atomize* (*full*)), *vector*)+
lemma *pth-c*: $(c::real) *s x + d *s y == d *s y + c *s x$
$(c *s x + z) + d *s y == d *s y + (c *s x + z)$
$c *s x + (d *s y + z) == d *s y + (c *s x + z)$
$(c *s x + w) + (d *s y + z) == d *s y + ((c *s x + w) + z)$ **by** ((*atomize* (*full*)), *vector*)+
lemma *pth-d*: $x + (0::real\ \hat{}'n) == x$ **by** (*atomize* (*full*)) *vector*

lemma *norm-imp-pos-and-ge*: $norm\ (x::real\ \hat{}\ \text{-}) == n \Longrightarrow norm\ x \geq 0 \wedge n \geq norm\ x$
by (*atomize*) (*auto simp add*: *norm-ge-zero*)

lemma *real-eq-0-iff-le-ge-0*: $(x::real) = 0 == x \geq 0 \wedge -x \geq 0$ **by** *arith*

lemma *norm-pths*:
$(x::real\ \hat{}'n::finite) = y \longleftrightarrow norm\ (x - y) \leq 0$
$x \neq y \longleftrightarrow \neg\ (norm\ (x - y) \leq 0)$
using *norm-ge-zero*[*of* $x - y$] **by** *auto*

use *normarith.ML*

method-setup *norm* = ⟪ *Scan.succeed* (*SIMPLE-METHOD' o NormArith.norm-arith-tac*) ⟫ *Proves simple linear statements about vector norms*

Hence more metric properties.

lemma *dist-refl*[*simp*]: $dist\ x\ x = 0$ **by** *norm*

lemma *dist-sym*: $dist\ x\ y = dist\ y\ x$**by** *norm*

lemma *dist-pos-le*[*simp*]: $0 <= dist\ x\ y$ **by** *norm*

lemma *dist-triangle*: $dist\ x\ z <= dist\ x\ y + dist\ y\ z$ **by** *norm*

lemma *dist-triangle-alt*: *dist y z <= dist x y + dist x z* **by** *norm*

lemma *dist-eq-0*[*simp*]: *dist x y = 0 ⟷ x = y* **by** *norm*

lemma *dist-pos-lt*: *x ≠ y ==> 0 < dist x y* **by** *norm*
lemma *dist-nz*: *x ≠ y ⟷ 0 < dist x y* **by** *norm*

lemma *dist-triangle-le*: *dist x z + dist y z <= e ⟹ dist x y <= e* **by** *norm*

lemma *dist-triangle-lt*: *dist x z + dist y z < e ==> dist x y < e* **by** *norm*

lemma *dist-triangle-half-l*: *dist x1 y < e / 2 ⟹ dist x2 y < e / 2 ==> dist x1 x2 < e* **by** *norm*

lemma *dist-triangle-half-r*: *dist y x1 < e / 2 ⟹ dist y x2 < e / 2 ==> dist x1 x2 < e* **by** *norm*

lemma *dist-triangle-add*: *dist (x + y) (x′ + y′) <= dist x x′ + dist y y′*
by *norm*

lemma *dist-mul*[*simp*]: *dist (c *s x) (c *s y) = |c| * dist x y*
unfolding *dist-def vector-ssub-ldistrib*[*symmetric*] *norm-mul* **..**

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

lemma *dist-le-0*[*simp*]: *dist x y <= 0 ⟷ x = y* **by** *norm*

lemma *setsum-component* [*simp*]:
fixes *f*:: *'a ⇒ ('b::comm-monoid-add) ^'n*
shows *(setsum f S)$i = setsum (λx. (f x)$i) S*
by (*cases finite S*, *induct S set*: *finite*, *simp-all*)

lemma *setsum-eq*: *setsum f S = (χ i. setsum (λx. (f x)$i) S)*
by (*simp add*: *Cart-eq*)

lemma *setsum-clauses*:
shows *setsum f {} = 0*
and *finite S ⟹ setsum f (insert x S) =*
(if x ∈ S then setsum f S else f x + setsum f S)
by (*auto simp add*: *insert-absorb*)

lemma *setsum-cmul*:
fixes *f*:: *'c ⇒ ('a::semiring-1) ^'n*
shows *setsum (λx. c *s f x) S = c *s setsum f S*
by (*simp add*: *Cart-eq setsum-right-distrib*)

lemma *setsum-norm*:
fixes *f* :: *'a ⇒ 'b::real-normed-vector*

assumes *fS*: *finite S*
shows *norm* (*setsum f S*) <= *setsum* (λ*x*. *norm*(*f x*)) *S*
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **by** *simp*
next
case (*2 x S*)
from *2.hyps* **have** *norm* (*setsum f* (*insert x S*)) ≤ *norm* (*f x*) + *norm* (*setsum f S*) **by** (*simp add*: *norm-triangle-ineq*)
also have ... ≤ *norm* (*f x*) + *setsum* (λ*x*. *norm*(*f x*)) *S*
using *2.hyps* **by** *simp*
finally **show** *?case* **using** *2.hyps* **by** *simp*
qed

lemma *real-setsum-norm*:
fixes *f* :: *'a* ⇒ *real* ^*'n*::*finite*
assumes *fS*: *finite S*
shows *norm* (*setsum f S*) <= *setsum* (λ*x*. *norm*(*f x*)) *S*
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **by** *simp*
next
case (*2 x S*)
from *2.hyps* **have** *norm* (*setsum f* (*insert x S*)) ≤ *norm* (*f x*) + *norm* (*setsum f S*) **by** (*simp add*: *norm-triangle-ineq*)
also have ... ≤ *norm* (*f x*) + *setsum* (λ*x*. *norm*(*f x*)) *S*
using *2.hyps* **by** *simp*
finally **show** *?case* **using** *2.hyps* **by** *simp*
qed

lemma *setsum-norm-le*:
fixes *f* :: *'a* ⇒ *'b*::*real-normed-vector*
assumes *fS*: *finite S*
and *fg*: ∀ *x* ∈ *S*. *norm* (*f x*) ≤ *g x*
shows *norm* (*setsum f S*) ≤ *setsum g S*
proof−
from *fg* **have** *setsum* (λ*x*. *norm*(*f x*)) *S* <= *setsum g S*
by − (*rule setsum-mono*, *simp*)
then show *?thesis* **using** *setsum-norm*[*OF fS*, *of f*] *fg*
by *arith*
qed

lemma *real-setsum-norm-le*:
fixes *f* :: *'a* ⇒ *real* ^ *'n*::*finite*
assumes *fS*: *finite S*
and *fg*: ∀ *x* ∈ *S*. *norm* (*f x*) ≤ *g x*
shows *norm* (*setsum f S*) ≤ *setsum g S*
proof−
from *fg* **have** *setsum* (λ*x*. *norm*(*f x*)) *S* <= *setsum g S*
by − (*rule setsum-mono*, *simp*)
then show *?thesis* **using** *real-setsum-norm*[*OF fS*, *of f*] *fg*

by *arith*
qed

lemma *setsum-norm-bound*:
fixes *f* :: *'a* ⇒ *'b*::*real-normed-vector*
assumes *fS*: *finite S*
and *K*: ∀ *x* ∈ *S*. *norm* (*f x*) ≤ *K*
shows *norm* (*setsum f S*) ≤ *of-nat* (*card S*) ∗ *K*
using *setsum-norm-le*[*OF fS K*] *setsum-constant*[*symmetric*]
by *simp*

lemma *real-setsum-norm-bound*:
fixes *f* :: *'a* ⇒ *real* ^ *'n*::*finite*
assumes *fS*: *finite S*
and *K*: ∀ *x* ∈ *S*. *norm* (*f x*) ≤ *K*
shows *norm* (*setsum f S*) ≤ *of-nat* (*card S*) ∗ *K*
using *real-setsum-norm-le*[*OF fS K*] *setsum-constant*[*symmetric*]
by *simp*

lemma *setsum-vmul*:
fixes *f* :: *'a* ⇒ *'b*::{*real-normed-vector*,*semiring*, *mult-zero*}
assumes *fS*: *finite S*
shows *setsum f S* ∗*s v* = *setsum* (λ*x*. *f x* ∗*s v*) *S*
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **then show** *?case* **by** (*simp add*: *vector-smult-lzero*)
next
case (*2 x F*)
from *2.hyps* **have** *setsum f* (*insert x F*) ∗*s v* = (*f x* + *setsum f F*) ∗*s v*
by *simp*
also have ... = *f x* ∗*s v* + *setsum f F* ∗*s v*
by (*simp add*: *vector-sadd-rdistrib*)
also have ... = *setsum* (λ*x*. *f x* ∗*s v*) (*insert x F*) **using** *2.hyps* **by** *simp*
finally show *?case* **.**
qed

lemma *setsum-add-split*: **assumes** *mn*: (*m*::*nat*) ≤ *n* + *1*
shows *setsum f* {*m*..*n* + *p*} = *setsum f* {*m*..*n*} + *setsum f* {*n* + *1*..*n* + *p*}
proof−
let *?A* = {*m* .. *n*}
let *?B* = {*n* + *1* .. *n* + *p*}
have *eq*: {*m* .. *n*+*p*} = *?A* ∪ *?B* **using** *mn* **by** *auto*
have *d*: *?A* ∩ *?B* = {} **by** *auto*
from *setsum-Un-disjoint*[*of ?A ?B f*] *eq d* **show** *?thesis* **by** *auto*
qed

lemma *setsum-natinterval-left*:
assumes *mn*: (*m*::*nat*) <= *n*

shows *setsum f {m..n} = f m + setsum f {m + 1..n}*
proof−
from *mn* **have** *{m .. n} = insert m {m+1 .. n}* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *setsum-natinterval-difff*:
fixes *f*:: *nat* ⇒ ('*a*::*ab-group-add*)
shows *setsum* (λ*k*. *f k* − *f*(*k* + *1*)) {(*m*::*nat*) .. *n*} =
(*if m <= n then f m* − *f*(*n* + *1*) *else 0*)
by (*induct n*, *auto simp add*: *ring-simps not-le le-Suc-eq*)

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

lemma *setsum-setsum-restrict*:
finite S ⟹ *finite T* ⟹ *setsum* (λ*x*. *setsum* (λ*y*. *f x y*) {*y*. *y*∈ *T* ∧ *R x y*}) *S*
= *setsum* (λ*y*. *setsum* (λ*x*. *f x y*) {*x*. *x* ∈ *S* ∧ *R x y*}) *T*
apply (*simp add*: *setsum-restrict-set'*[*unfolded mem-def*] *mem-def*)
by (*rule setsum-commute*)

lemma *setsum-image-gen*: **assumes** *fS*: *finite S*
shows *setsum g S = setsum* (λ*y*. *setsum g* {*x*. *x* ∈ *S* ∧ *f x* = *y*}) (*f* ' *S*)
proof−
{fix *x* **assume** *x* ∈ *S* **then have** {*y*. *y*∈ *f*'*S* ∧ *f x* = *y*} = {*f x*} **by** *auto***}**
note *th0* = *this*
have *setsum g S = setsum* (λ*x*. *setsum* (λ*y*. *g x*) {*y*. *y*∈ *f*'*S* ∧ *f x* = *y*}) *S*
apply (*rule setsum-cong2*)
by (*simp add*: *th0*)
also have ... = *setsum* (λ*y*. *setsum g* {*x*. *x* ∈ *S* ∧ *f x* = *y*}) (*f* ' *S*)
apply (*rule setsum-setsum-restrict*[*OF fS*])
by (*rule finite-imageI*[*OF fS*])
finally show *?thesis* **.**
qed

lemma *setsum-group*:
assumes *fS*: *finite S* **and** *fT*: *finite T* **and** *fST*: *f* ' *S* ⊆ *T*
shows *setsum* (λ*y*. *setsum g* {*x*. *x*∈ *S* ∧ *f x* = *y*}) *T* = *setsum g S*

apply (*subst setsum-image-gen*[*OF fS*, *of g f*])
apply (*rule setsum-mono-zero-right*[*OF fT fST*])
by (*auto intro*: *setsum-0'*)

lemma *vsum-norm-allsubsets-bound*:
fixes *f*:: '*a* ⇒ *real* ^'*n*::*finite*
assumes *fP*: *finite P* **and** *fPs*: ⋀*Q*. *Q* ⊆ *P* ⟹ *norm* (*setsum f Q*) ≤ *e*
shows *setsum* (λ*x*. *norm* (*f x*)) *P* ≤ *2* ∗ *real CARD*('*n*) ∗ *e*
proof−
let *?d* = *real CARD*('*n*)

let *?nf* = $\lambda x.\ norm\ (f\ x)$
let *?U* = *UNIV* :: *'n set*
have *th0*: *setsum* $(\lambda x.\ setsum\ (\lambda i.\ |f\ x\ \$\ i|)\ ?U)\ P = setsum\ (\lambda i.\ setsum\ (\lambda x.\ |f\ x\ \$\ i|)\ P)\ ?U$
by (*rule setsum-commute*)
have *th1*: *2 * ?d * e = of-nat (card ?U) * (2 * e)* **by** (*simp add*: *real-of-nat-def*)
have *setsum ?nf P* $\leq$ *setsum* $(\lambda x.\ setsum\ (\lambda i.\ |f\ x\ \$\ i|)\ ?U)\ P$
apply (*rule setsum-mono*)
by (*rule norm-le-l1*)
also have ... $\leq$ *2 * ?d * e*
unfolding *th0 th1*
proof(*rule setsum-bounded*)
fix *i* **assume** *i*: $i \in ?U$
let *?Pp* = $\{x.\ x\in P \wedge f\ x\ \$\ i \geq 0\}$
let *?Pn* = $\{x.\ x \in P \wedge f\ x\ \$\ i < 0\}$
have *thp*: $P = ?Pp \cup ?Pn$ **by** *auto*
have *thp0*: $?Pp \cap ?Pn = \{\}$ **by** *auto*
have *PpP*: $?Pp \subseteq P$ **and** *PnP*: $?Pn \subseteq P$ **by** *blast+*
have *Ppe*:*setsum* $(\lambda x.\ |f\ x\ \$\ i|)\ ?Pp \leq e$
using *component-le-norm*[*of setsum* $(\lambda x.\ f\ x)$ *?Pp i*] *fPs*[*OF PpP*]
by (*auto intro*: *abs-le-D1*)
have *Pne*: *setsum* $(\lambda x.\ |f\ x\ \$\ i|)\ ?Pn \leq e$
using *component-le-norm*[*of setsum* $(\lambda x.\ -\ f\ x)$ *?Pn i*] *fPs*[*OF PnP*]
by (*auto simp add*: *setsum-negf intro*: *abs-le-D1*)
have *setsum* $(\lambda x.\ |f\ x\ \$\ i|)\ P = setsum\ (\lambda x.\ |f\ x\ \$\ i|)\ ?Pp + setsum\ (\lambda x.\ |f\ x\ \$\ i|)\ ?Pn$
apply (*subst thp*)
apply (*rule setsum-Un-zero*)
using *fP thp0* **by** *auto*
also have ... $\leq$ *2*e* **using** *Pne Ppe* **by** *arith*
finally show *setsum* $(\lambda x.\ |f\ x\ \$\ i|)\ P \leq$ *2*e* **.**
qed
finally show *?thesis* **.**
qed

lemma *dot-lsum*: *finite S* $\Longrightarrow$ *setsum f S* $\cdot$ $(y::'a::\{comm\text{-}ring\}\ \hat{}\ 'n) = setsum\ (\lambda x.\ f\ x \cdot y)\ S$
by (*induct rule*: *finite-induct*, *auto simp add*: *dot-lzero dot-ladd dot-radd*)

lemma *dot-rsum*: *finite S* $\Longrightarrow$ $(y::'a::\{comm\text{-}ring\}\ \hat{}\ 'n) \cdot setsum\ f\ S = setsum\ (\lambda x.\ y \cdot f\ x)\ S$
by (*induct rule*: *finite-induct*, *auto simp add*: *dot-rzero dot-radd*)

30.11 Basis vectors in coordinate directions.

definition *basis k* = $(\chi\ i.\ if\ i = k\ then\ 1\ else\ 0)$

lemma *basis-component* [*simp*]: *basis k \$ i* = (*if k=i then 1 else 0*)
unfolding *basis-def* **by** *simp*

lemma *delta-mult-idempotent*:
(*if k=a then 1 else (0::'a::semiring-1)*) ∗ (*if k=a then 1 else 0*) = (*if k=a then 1 else 0*) **by** (*cases k=a*, *auto*)

lemma *norm-basis*:
shows *norm* (*basis k* :: *real* ^'*n*::*finite*) = *1*
apply (*simp add*: *basis-def real-vector-norm-def dot-def*)
apply (*vector delta-mult-idempotent*)
using *setsum-delta*[*of UNIV* :: '*n set k* λ*k*. *1*::*real*]
apply *auto*
done

lemma *norm-basis-1*: *norm*(*basis 1* :: *real* ^'*n*::{*finite*,*one*}) = *1*
by (*rule norm-basis*)

lemma *vector-choose-size*: *0* <= *c* ==> ∃(*x*::*real*^'*n*::*finite*). *norm x* = *c*
apply (*rule exI*[**where** *x*=*c* ∗*s basis arbitrary*])
by (*simp only*: *norm-mul norm-basis*)

lemma *vector-choose-dist*: **assumes** *e*: *0* <= *e*
shows ∃(*y*::*real*^'*n*::*finite*). *dist x y* = *e*
proof−
from *vector-choose-size*[*OF e*] **obtain** *c*:: *real* ^'*n* **where** *norm c* = *e*
by *blast*
then have *dist x* (*x* − *c*) = *e* **by** (*simp add*: *dist-def*)
then show *?thesis* **by** *blast*
qed

lemma *basis-inj*: *inj* (*basis* :: '*n* ⇒ *real* ^'*n*::*finite*)
by (*simp add*: *inj-on-def Cart-eq*)

lemma *cond-value-iff*: *f* (*if b then x else y*) = (*if b then f x else f y*)
by *auto*

lemma *basis-expansion*:
setsum (λ*i*. (*x*$*i*) ∗*s basis i*) *UNIV* = (*x*::('*a*::*ring-1*) ^'*n*::*finite*) (**is** *?lhs* = *?rhs* **is** *setsum ?f ?S* = -)
by (*auto simp add*: *Cart-eq cond-value-iff setsum-delta*[*of ?S*, **where** *?'b* = '*a*, *simplified*] *cong del*: *if-weak-cong*)

lemma *basis-expansion-unique*:
setsum (λ*i*. *f i* ∗*s basis i*) *UNIV* = (*x*::('*a*::*comm-ring-1*) ^'*n*::*finite*) ⟷ (∀ *i*. *f i* = *x*$*i*)
by (*simp add*: *Cart-eq setsum-delta cond-value-iff cong del*: *if-weak-cong*)

lemma *cond-application-beta*: (*if b then f else g*) *x* = (*if b then f x else g x*)
by *auto*

lemma *dot-basis*:
shows *basis i · x = x$i x · (basis i :: 'a^'n::finite) = (x$i :: 'a::semiring-1)*
by (*auto simp add*: *dot-def basis-def cond-application-beta cond-value-iff setsum-delta cong del*: *if-weak-cong*)

lemma *basis-eq-0*: *basis i = (0::'a::semiring-1^'n) ⟷ False*
by (*auto simp add*: *Cart-eq*)

lemma *basis-nonzero*:
shows *basis k ≠ (0:: 'a::semiring-1 ^'n)*
by (*simp add*: *basis-eq-0*)

lemma *vector-eq-ldot*: *(∀ x. x · y = x · z) ⟷ y = (z::'a::semiring-1^'n::finite)*
apply (*auto simp add*: *Cart-eq dot-basis*)
apply (*erule-tac x=basis i* **in** *allE*)
apply (*simp add*: *dot-basis*)
apply (*subgoal-tac y = z*)
apply *simp*
apply (*simp add*: *Cart-eq*)
done

lemma *vector-eq-rdot*: *(∀ z. x · z = y · z) ⟷ x = (y::'a::semiring-1^'n::finite)*
apply (*auto simp add*: *Cart-eq dot-basis*)
apply (*erule-tac x=basis i* **in** *allE*)
apply (*simp add*: *dot-basis*)
apply (*subgoal-tac x = y*)
apply *simp*
apply (*simp add*: *Cart-eq*)
done

30.12 Orthogonality.

definition *orthogonal x y ⟷ (x · y = 0)*

lemma *orthogonal-basis*:
shows *orthogonal (basis i :: 'a^'n::finite) x ⟷ x$i = (0::'a::ring-1)*
by (*auto simp add*: *orthogonal-def dot-def basis-def cond-value-iff cond-application-beta setsum-delta cong del*: *if-weak-cong*)

lemma *orthogonal-basis-basis*:
shows *orthogonal (basis i :: 'a::ring-1^'n::finite) (basis j) ⟷ i ≠ j*
unfolding *orthogonal-basis*[*of i*] *basis-component*[*of j*] **by** *simp*

lemma *orthogonal-clauses*:
orthogonal a (0::'a::comm-ring ^'n)
*orthogonal a x ==> orthogonal a (c *s x)*
orthogonal a x ==> orthogonal a (−x)
orthogonal a x ⟹ orthogonal a y ==> orthogonal a (x + y)

orthogonal a x $\Longrightarrow$ *orthogonal a y* ==> *orthogonal a* $(x - y)$
orthogonal 0 a
orthogonal x a ==> *orthogonal* $(c *s x)$ *a*
orthogonal x a ==> *orthogonal* $(-x)$ *a*
orthogonal x a $\Longrightarrow$ *orthogonal y a* ==> *orthogonal* $(x + y)$ *a*
orthogonal x a $\Longrightarrow$ *orthogonal y a* ==> *orthogonal* $(x - y)$ *a*
unfolding *orthogonal-def dot-rneg dot-rmult dot-radd dot-rsub dot-lzero dot-rzero dot-lneg dot-lmult dot-ladd dot-lsub*
by *simp-all*

lemma *orthogonal-commute*: *orthogonal* $(x::'a::\{$*ab-semigroup-mult*,*comm-monoid-add*$\}\hat{}'n)y \longleftrightarrow$ *orthogonal y x*
by (*simp add*: *orthogonal-def dot-sym*)

30.13 Explicit vector construction from lists.

primrec *from-nat* :: *nat* $\Rightarrow$ $'a::\{$*monoid-add*,*one*$\}$
where *from-nat 0 = 0* | *from-nat* (*Suc n*) = *1* + *from-nat n*

lemma *from-nat* [*simp*]: *from-nat = of-nat*
by (*rule ext*, *induct-tac x*, *simp-all*)

primrec
list-fun :: *nat* $\Rightarrow$ - *list* $\Rightarrow$ - $\Rightarrow$ -
where
list-fun n [] = $(\lambda x.\ 0)$
| *list-fun n* (*x* # *xs*) = *fun-upd* (*list-fun* (*Suc n*) *xs*) (*from-nat n*) *x*

definition *vector l* = $(\chi\ i.$ *list-fun 1 l i*$)$

lemma *vector-1*: (*vector*[*x*]) $1 = x$
unfolding *vector-def* **by** *simp*

lemma *vector-2*:
(*vector*[*x*,*y*]) $1 = x$
(*vector*[*x*,*y*] :: $'a\hat{}2)2 = (y::'a::zero)$
unfolding *vector-def* **by** *simp-all*

lemma *vector-3*:
(*vector* [*x*,*y*,*z*] ::$('a::zero)\hat{}3)1 = x$
(*vector* [*x*,*y*,*z*] ::$('a::zero)\hat{}3)2 = y$
(*vector* [*x*,*y*,*z*] ::$('a::zero)\hat{}3)3 = z$
unfolding *vector-def* **by** *simp-all*

lemma *forall-vector-1*: $(\forall\, v::'a::zero\hat{}1.\ P\ v) \longleftrightarrow (\forall\, x.\ P(vector[x]))$
apply *auto*
apply (*erule-tac x=v*1 **in** *allE*)
apply (*subgoal-tac vector* [*v*1] = *v*)

apply *simp*
apply (*vector vector-def*)
apply (*simp add*: *forall-1*)
done

lemma *forall-vector-2*: $(\forall\, v{::}'a{::}zero\hat{}2.\ P\ v) \longleftrightarrow (\forall\, x\ y.\ P(vector[x,\ y]))$
apply *auto*
apply (*erule-tac x=v$1* **in** *allE*)
apply (*erule-tac x=v$2* **in** *allE*)
apply (*subgoal-tac vector* $[v\$1,\ v\$2] = v$)
apply *simp*
apply (*vector vector-def*)
apply (*simp add*: *forall-2*)
done

lemma *forall-vector-3*: $(\forall\, v{::}'a{::}zero\hat{}3.\ P\ v) \longleftrightarrow (\forall\, x\ y\ z.\ P(vector[x,\ y,\ z]))$
apply *auto*
apply (*erule-tac x=v$1* **in** *allE*)
apply (*erule-tac x=v$2* **in** *allE*)
apply (*erule-tac x=v$3* **in** *allE*)
apply (*subgoal-tac vector* $[v\$1,\ v\$2,\ v\$3] = v$)
apply *simp*
apply (*vector vector-def*)
apply (*simp add*: *forall-3*)
done

30.14 Linear functions.

definition $linear\ f \longleftrightarrow (\forall\, x\ y.\ f(x + y) = f\ x + f\ y) \wedge (\forall\, c\ x.\ f(c\ *s\ x) = c\ *s\ f\ x)$

lemma *linear-compose-cmul*: $linear\ f \Longrightarrow linear\ (\lambda x.\ (c{::}'a{::}comm\text{-}semiring)\ *s\ f\ x)$
by (*vector linear-def Cart-eq ring-simps*)

lemma *linear-compose-neg*: $linear\ (f :: 'a\ \hat{}\ 'n \Rightarrow 'a{::}comm\text{-}ring\ \hat{}\ 'm) \Longrightarrow linear\ (\lambda x.\ -(f(x)))$ **by** (*vector linear-def Cart-eq*)

lemma *linear-compose-add*: $linear\ (f :: 'a\ \hat{}\ 'n \Rightarrow 'a{::}semiring\text{-}1\ \hat{}\ 'm) \Longrightarrow linear\ g \Longrightarrow linear\ (\lambda x.\ f(x) + g(x))$
by (*vector linear-def Cart-eq ring-simps*)

lemma *linear-compose-sub*: $linear\ (f :: 'a\ \hat{}\ 'n \Rightarrow 'a{::}ring\text{-}1\ \hat{}\ 'm) \Longrightarrow linear\ g \Longrightarrow linear\ (\lambda x.\ f\ x - g\ x)$
by (*vector linear-def Cart-eq ring-simps*)

lemma *linear-compose*: $linear\ f \Longrightarrow linear\ g \Longrightarrow linear\ (g\ o\ f)$
by (*simp add*: *linear-def*)

lemma *linear-id*: *linear id* **by** (*simp add*: *linear-def id-def*)

lemma *linear-zero*: *linear* (λ*x*. *0*::*'a*::*semiring-1* ^ *'n*) **by** (*simp add*: *linear-def*)

lemma *linear-compose-setsum*:
assumes *fS*: *finite S* **and** *lS*: ∀ *a* ∈ *S*. *linear* (*f a* :: *'a*::*semiring-1* ^ *'n* ⇒ *'a* ^ *'m*)
shows *linear*(λ*x*. *setsum* (λ*a*. *f a x* :: *'a*::*semiring-1* ^*'m*) *S*)
using *lS*
apply (*induct rule*: *finite-induct*[*OF fS*])
by (*auto simp add*: *linear-zero intro*: *linear-compose-add*)

lemma *linear-vmul-component*:
fixes *f*:: *'a*::*semiring-1*^*'m* ⇒ *'a*^*'n*
assumes *lf*: *linear f*
shows *linear* (λ*x*. *f x* $ *k* ∗*s v*)
using *lf*
apply (*auto simp add*: *linear-def*)
by (*vector ring-simps*)+

lemma *linear-0*: *linear f* ==> *f 0* = (*0*::*'a*::*semiring-1* ^*'n*)
unfolding *linear-def*
apply *clarsimp*
apply (*erule allE*[**where** *x*=*0*::*'a*])
apply *simp*
done

lemma *linear-cmul*: *linear f* ==> *f*(*c*∗*s x*) = *c* ∗*s f x* **by** (*simp add*: *linear-def*)

lemma *linear-neg*: *linear* (*f* :: *'a*::*ring-1* ^*'n* ⇒ -) ==> *f* (−*x*) = − *f x*
unfolding *vector-sneg-minus1*
using *linear-cmul*[*of f*] **by** *auto*

lemma *linear-add*: *linear f* ==> *f*(*x* + *y*) = *f x* + *f y* **by** (*metis linear-def*)

lemma *linear-sub*: *linear* (*f*::*'a*::*ring-1* ^*'n* ⇒ -) ==> *f*(*x* − *y*) = *f x* − *f y*
by (*simp add*: *diff-def linear-add linear-neg*)

lemma *linear-setsum*:
fixes *f*:: *'a*::*semiring-1*^*'n* ⇒ -
assumes *lf*: *linear f* **and** *fS*: *finite S*
shows *f* (*setsum g S*) = *setsum* (*f o g*) *S*
proof (*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **by** (*simp add*: *linear-0*[*OF lf*])
next
case (*2 x F*)
have *f* (*setsum g* (*insert x F*)) = *f* (*g x* + *setsum g F*) **using** *2.hyps*
by *simp*
also have ... = *f* (*g x*) + *f* (*setsum g F*) **using** *linear-add*[*OF lf*] **by** *simp*

```
  also have ... = setsum (f o g) (insert x F) using 2.hyps by simp
  finally show ?case .
qed

lemma linear-setsum-mul:
  fixes f:: 'a ^'n ⇒ 'a::semiring-1^'m
  assumes lf: linear f and fS: finite S
  shows f (setsum (λi. c i *s v i) S) = setsum (λi. c i *s f (v i)) S
  using linear-setsum[OF lf fS, of λi. c i *s v i , unfolded o-def]
  linear-cmul[OF lf] by simp

lemma linear-injective-0:
  assumes lf: linear (f:: 'a::ring-1 ^ 'n ⇒ -)
  shows inj f ⟷ (∀x. f x = 0 ⟶ x = 0)
proof−
  have inj f ⟷ (∀ x y. f x = f y ⟶ x = y) by (simp add: inj-on-def)
  also have ... ⟷ (∀ x y. f x − f y = 0 ⟶ x − y = 0) by simp
  also have ... ⟷ (∀ x y. f (x − y) = 0 ⟶ x − y = 0)
    by (simp add: linear-sub[OF lf])
  also have ... ⟷ (∀ x. f x = 0 ⟶ x = 0) by auto
  finally show ?thesis .
qed

lemma linear-bounded:
  fixes f:: real ^'m::finite ⇒ real ^'n::finite
  assumes lf: linear f
  shows ∃B. ∀x. norm (f x) ≤ B * norm x
proof−
  let ?S = UNIV:: 'm set
  let ?B = setsum (λi. norm(f(basis i))) ?S
  have fS: finite ?S by simp
  {fix x:: real ^ 'm
    let ?g = (λi. (x$i) *s (basis i) :: real ^ 'm)
    have norm (f x) = norm (f (setsum (λi. (x$i) *s (basis i)) ?S))
      by (simp only:  basis-expansion)
    also have ... = norm (setsum (λi. (x$i) *s f (basis i))?S)
      using linear-setsum[OF lf fS, of ?g, unfolded o-def] linear-cmul[OF lf]
      by auto
    finally have th0: norm (f x) = norm (setsum (λi. (x$i) *s f (basis i))?S) .
    {fix i assume i: i ∈ ?S
      from component-le-norm[of x i]
      have norm ((x$i) *s f (basis i :: real ^'m)) ≤ norm (f (basis i)) * norm x
      unfolding norm-mul
      apply (simp only: mult-commute)
      apply (rule mult-mono)
      by (auto simp add: ring-simps norm-ge-zero) }
   then have th: ∀i∈ ?S. norm ((x$i) *s f (basis i :: real ^'m)) ≤ norm (f (basis
i)) * norm x by metis
    from real-setsum-norm-le[OF fS, of λi. (x$i) *s (f (basis i)), OF th]
```

```
    have norm (f x) ≤ ?B * norm x unfolding th0 setsum-left-distrib by metis}
  then show ?thesis by blast
qed

lemma linear-bounded-pos:
  fixes f:: real ^'n::finite ⇒ real ^ 'm::finite
  assumes lf: linear f
  shows ∃ B > 0. ∀ x. norm (f x) ≤ B * norm x
proof−
  from linear-bounded[OF lf] obtain B where
    B: ∀ x. norm (f x) ≤ B * norm x by blast
  let ?K = |B| + 1
  have Kp: ?K > 0 by arith
    {assume C: B < 0
      have norm (1::real ^ 'n) > 0 by (simp add: zero-less-norm-iff)
      with C have B * norm (1:: real ^ 'n) < 0
        by (simp add: zero-compare-simps)
      with B[rule-format, of 1] norm-ge-zero[of f 1] have False by simp
    }
    then have Bp: B ≥ 0 by ferrack
    {fix x::real ^ 'n
      have norm (f x) ≤ ?K *  norm x
      using B[rule-format, of x] norm-ge-zero[of x] norm-ge-zero[of f x] Bp
      apply (auto simp add: ring-simps split add: abs-split)
      apply (erule order-trans, simp)
      done
  }
  then show ?thesis using Kp by blast
qed
```

30.15 Bilinear functions.

```
definition bilinear f ⟷ (∀ x. linear(λy. f x y)) ∧ (∀ y. linear(λx. f x y))

lemma bilinear-ladd: bilinear h ==> h (x + y) z = (h x z) + (h y z)
  by (simp add: bilinear-def linear-def)
lemma bilinear-radd: bilinear h ==> h x (y + z) = (h x y) + (h x z)
  by (simp add: bilinear-def linear-def)

lemma bilinear-lmul: bilinear h ==> h (c *s x) y = c *s (h x y)
  by (simp add: bilinear-def linear-def)

lemma bilinear-rmul: bilinear h ==> h x (c *s y) = c *s (h x y)
  by (simp add: bilinear-def linear-def)

lemma bilinear-lneg: bilinear h ==> h (− (x:: 'a::ring-1 ^ 'n)) y = −(h x y)
  by (simp only: vector-sneg-minus1 bilinear-lmul)

lemma bilinear-rneg: bilinear h ==> h x (− (y:: 'a::ring-1 ^ 'n)) = − h x y
```

by (*simp only*: *vector-sneg-minus1 bilinear-rmul*)

lemma (**in** *ab-group-add*) *eq-add-iff*: $x = x + y \longleftrightarrow y = 0$
using *add-imp-eq*[*of x y 0*] **by** *auto*

lemma *bilinear-lzero*:
fixes $h :: 'a::ring\hat{}'n \Rightarrow$ - **assumes** *bh*: *bilinear h* **shows** $h\ 0\ x = 0$
using *bilinear-ladd*[*OF bh*, *of 0 0 x*]
by (*simp add*: *eq-add-iff ring-simps*)

lemma *bilinear-rzero*:
fixes $h :: 'a::ring\hat{}'n \Rightarrow$ - **assumes** *bh*: *bilinear h* **shows** $h\ x\ 0 = 0$
using *bilinear-radd*[*OF bh*, *of x 0 0*]
by (*simp add*: *eq-add-iff ring-simps*)

lemma *bilinear-lsub*: *bilinear h* ==> $h\ (x - (y::'a::ring\text{-}1\ \hat{}\ 'n))\ z = h\ x\ z - h\ y\ z$
by (*simp add*: *diff-def bilinear-ladd bilinear-lneg*)

lemma *bilinear-rsub*: *bilinear h* ==> $h\ z\ (x - (y::'a::ring\text{-}1\ \hat{}\ 'n)) = h\ z\ x - h\ z\ y$
by (*simp add*: *diff-def bilinear-radd bilinear-rneg*)

lemma *bilinear-setsum*:
fixes $h::'a\ \hat{}'n \Rightarrow 'a::semiring\text{-}1\hat{}'m \Rightarrow 'a\ \hat{}\ 'k$
assumes *bh*: *bilinear h* **and** *fS*: *finite S* **and** *fT*: *finite T*
shows $h\ (setsum\ f\ S)\ (setsum\ g\ T) = setsum\ (\lambda(i,j).\ h\ (f\ i)\ (g\ j))\ (S \times T)$
proof−
have $h\ (setsum\ f\ S)\ (setsum\ g\ T) = setsum\ (\lambda x.\ h\ (f\ x)\ (setsum\ g\ T))\ S$
apply (*rule linear-setsum*[*unfolded o-def*])
using *bh fS* **by** (*auto simp add*: *bilinear-def*)
also have $\ldots = setsum\ (\lambda x.\ setsum\ (\lambda y.\ h\ (f\ x)\ (g\ y))\ T)\ S$
apply (*rule setsum-cong*, *simp*)
apply (*rule linear-setsum*[*unfolded o-def*])
using *bh fT* **by** (*auto simp add*: *bilinear-def*)
finally show *?thesis* **unfolding** *setsum-cartesian-product* **.**
qed

lemma *bilinear-bounded*:
fixes $h::real\ \hat{}'m::finite \Rightarrow real\hat{}'n::finite \Rightarrow real\ \hat{}\ 'k::finite$
assumes *bh*: *bilinear h*
shows $\exists B.\ \forall x\ y.\ norm\ (h\ x\ y) \leq B * norm\ x * norm\ y$
proof−
let $?M = UNIV :: 'm\ set$
let $?N = UNIV :: 'n\ set$
let $?B = setsum\ (\lambda(i,j).\ norm\ (h\ (basis\ i)\ (basis\ j)))\ (?M \times ?N)$
have *fM*: *finite ?M* **and** *fN*: *finite ?N* **by** *simp-all*
{fix $x::real\ \hat{}\ 'm$ **and** $y :: real\hat{}'n$
have $norm\ (h\ x\ y) = norm\ (h\ (setsum\ (\lambda i.\ (x\$i) *s basis\ i)\ ?M)\ (setsum\ (\lambda i.$

(*y*$*i*) ∗*s basis i*) *?N*)) **unfolding** *basis-expansion* **..**
also have ... = *norm* (*setsum* (λ (*i*,*j*). *h* ((*x*$*i*) ∗*s basis i*) ((*y*$*j*) ∗*s basis j*)) (*?M* × *?N*)) **unfolding** *bilinear-setsum*[*OF bh fM fN*] **..**
finally have *th*: *norm* (*h x y*) = ... **.**
have *norm* (*h x y*) ≤ *?B* ∗ *norm x* ∗ *norm y*
apply (*simp add*: *setsum-left-distrib th*)
apply (*rule real-setsum-norm-le*)
using *fN fM*
apply *simp*
apply (*auto simp add*: *bilinear-rmul*[*OF bh*] *bilinear-lmul*[*OF bh*] *norm-mul ring-simps*)
apply (*rule mult-mono*)
apply (*auto simp add*: *norm-ge-zero zero-le-mult-iff component-le-norm*)
apply (*rule mult-mono*)
apply (*auto simp add*: *norm-ge-zero zero-le-mult-iff component-le-norm*)
done}
then show *?thesis* **by** *metis*
qed

lemma *bilinear-bounded-pos*:
fixes *h*:: *real* ^ *'m*::*finite* ⇒ *real* ^ *'n*::*finite* ⇒ *real* ^ *'k*::*finite*
assumes *bh*: *bilinear h*
shows ∃ *B* > *0*. ∀ *x y*. *norm* (*h x y*) ≤ *B* ∗ *norm x* ∗ *norm y*
proof−
from *bilinear-bounded*[*OF bh*] **obtain** *B* **where**
B: ∀ *x y*. *norm* (*h x y*) ≤ *B* ∗ *norm x* ∗ *norm y* **by** *blast*
let *?K* = |*B*| + *1*
have *Kp*: *?K* > *0* **by** *arith*
have *KB*: *B* < *?K* **by** *arith*
{fix *x*::*real* ^ *'m* **and** *y* :: *real* ^ *'n*
from *KB Kp*
have *B* ∗ *norm x* ∗ *norm y* ≤ *?K* ∗ *norm x* ∗ *norm y*
apply −
apply (*rule mult-right-mono*, *rule mult-right-mono*)
by (*auto simp add*: *norm-ge-zero*)
then have *norm* (*h x y*) ≤ *?K* ∗ *norm x* ∗ *norm y*
using *B*[*rule-format*, *of x y*] **by** *simp***}**
with *Kp* **show** *?thesis* **by** *blast*
qed

30.16 Adjoints.

definition *adjoint f* = (*SOME f'*. ∀ *x y*. *f x* · *y* = *x* · *f' y*)

lemma *choice-iff*: (∀ *x*. ∃ *y*. *P x y*) ⟷ (∃ *f*. ∀ *x*. *P x* (*f x*)) **by** *metis*

lemma *adjoint-works-lemma*:
fixes *f*:: *'a*::*ring-1* ^ *'n*::*finite* ⇒ *'a* ^ *'m*::*finite*
assumes *lf*: *linear f*

shows $\forall x\ y.\ f\ x \cdot y = x \cdot adjoint\ f\ y$
proof−
let *?N* = *UNIV* :: *'n set*
let *?M* = *UNIV* :: *'m set*
have *fN*: *finite ?N* **by** *simp*
have *fM*: *finite ?M* **by** *simp*
{fix *y*:: *'a ^ 'm*
let *?w* = $(\chi\ i.\ (f\ (basis\ i) \cdot y))$ :: *'a ^ 'n*
{fix *x*
have $f\ x \cdot y = f\ (setsum\ (\lambda i.\ (x\$i) *s\ basis\ i)\ ?N) \cdot y$
by (*simp only*: *basis-expansion*)
also have $\ldots = (setsum\ (\lambda i.\ (x\$i) *s\ f\ (basis\ i))\ ?N) \cdot y$
unfolding *linear-setsum*[*OF lf fN*]
by (*simp add*: *linear-cmul*[*OF lf*])
finally have $f\ x \cdot y = x \cdot ?w$
apply (*simp only*:)
apply (*simp add*: *dot-def setsum-left-distrib setsum-right-distrib setsum-commute*[*of - ?M ?N*] *ring-simps*)
done}
}
then show *?thesis* **unfolding** *adjoint-def*
some-eq-ex[*of* $\lambda f'.\ \forall x\ y.\ f\ x \cdot y = x \cdot f'\ y$]
using *choice-iff*[*of* $\lambda a\ b.\ \forall x.\ f\ x \cdot a = x \cdot b$]
by *metis*
qed

lemma *adjoint-works*:
fixes *f*:: *'a*::*ring-1 ^'n*::*finite* ⇒ *'a ^ 'm*::*finite*
assumes *lf*: *linear f*
shows $x \cdot adjoint\ f\ y = f\ x \cdot y$
using *adjoint-works-lemma*[*OF lf*] **by** *metis*

lemma *adjoint-linear*:
fixes *f* :: *'a*::*comm-ring-1 ^'n*::*finite* ⇒ *'a ^ 'm*::*finite*
assumes *lf*: *linear f*
shows *linear* (*adjoint f*)
by (*simp add*: *linear-def vector-eq-ldot*[*symmetric*] *dot-radd dot-rmult adjoint-works*[*OF lf*])

lemma *adjoint-clauses*:
fixes *f*:: *'a*::*comm-ring-1 ^'n*::*finite* ⇒ *'a ^ 'm*::*finite*
assumes *lf*: *linear f*
shows $x \cdot adjoint\ f\ y = f\ x \cdot y$
and $adjoint\ f\ y \cdot x = y \cdot f\ x$
by (*simp-all add*: *adjoint-works*[*OF lf*] *dot-sym*)

lemma *adjoint-adjoint*:
fixes *f*:: *'a*::*comm-ring-1 ^ 'n*::*finite* ⇒ *'a ^ 'm*::*finite*

assumes *lf*: *linear f*
shows *adjoint* (*adjoint f*) = *f*
apply (*rule ext*)
by (*simp add*: *vector-eq-ldot*[*symmetric*] *adjoint-clauses*[*OF adjoint-linear*[*OF lf*]] *adjoint-clauses*[*OF lf*])

lemma *adjoint-unique*:
fixes *f*:: *'a*::*comm-ring-1* ^ *'n*::*finite* ⇒ *'a* ^ *'m*::*finite*
assumes *lf*: *linear f* **and** *u*: ∀ *x y*. *f' x* ⋅ *y* = *x* ⋅ *f y*
shows *f'* = *adjoint f*
apply (*rule ext*)
using *u*
by (*simp add*: *vector-eq-rdot*[*symmetric*] *adjoint-clauses*[*OF lf*])

Matrix notation. NB: an MxN matrix is of type *'a* ^ *'n* ^ *'m*, not *'a* ^ *'m* ^ *'n*

consts *generic-mult* :: *'a* ⇒ *'b* ⇒ *'c* (**infixr** ⋆ *75*)

defs (**overloaded**)
matrix-matrix-mult-def: (*m*:: (*'a*::*semiring-1*) ^*'n*^*'m*) ⋆ (*m'* :: *'a* ^*'p*^*'n*) ≡ (χ *i j*. *setsum* (λ*k*. ((*m*$*i*)$*k*) ∗ ((*m'*$*k*)$*j*)) (*UNIV* :: *'n set*)) ::*'a* ^ *'p* ^*'m*

abbreviation
matrix-matrix-mult' :: (*'a*::*semiring-1*) ^*'n*^*'m* ⇒ *'a* ^*'p*^*'n* ⇒ *'a* ^ *'p* ^*'m* (**infixl** ∗∗ *70*)
where *m* ∗∗ *m'* == *m*⋆ *m'*

defs (**overloaded**)
matrix-vector-mult-def: (*m*::(*'a*::*semiring-1*) ^*'n*^*'m*) ⋆ (*x*::*'a* ^*'n*) ≡ (χ *i*. *setsum* (λ*j*. ((*m*$*i*)$*j*) ∗ (*x*$*j*)) (*UNIV* ::*'n set*)) :: *'a*^*'m*

abbreviation
matrix-vector-mult' :: (*'a*::*semiring-1*) ^*'n*^*'m* ⇒ *'a* ^*'n* ⇒ *'a* ^ *'m* (**infixl** ∗*v* *70*)
where
m ∗*v v* == *m* ⋆ *v*

defs (**overloaded**)
vector-matrix-mult-def: (*x*::*'a*^*'m*) ⋆ (*m*::(*'a*::*semiring-1*) ^*'n*^*'m*) ≡ (χ *j*. *setsum* (λ*i*. ((*m*$*i*)$*j*) ∗ (*x*$*i*)) (*UNIV* :: *'m set*)) :: *'a*^*'n*

abbreviation
vactor-matrix-mult' :: *'a* ^ *'m* ⇒ (*'a*::*semiring-1*) ^*'n*^*'m* ⇒ *'a* ^*'n* (**infixl** *v*∗ *70*)
where
v v∗ *m* == *v* ⋆ *m*

definition (*mat*::*'a*::*zero* => *'a* ^*'n*^*'n*) *k* = (χ *i j*. *if i* = *j then k else 0*)
definition (*transp*::*'a*^*'n*^*'m* ⇒ *'a*^*'m*^*'n*) *A* = (χ *i j*. ((*A*$*j*)$*i*))

definition (*row*::'*m* => '*a* ^'*n*^'*m* ⇒ '*a* ^'*n*) *i A* = (χ *j*. ((*A*$*i*)$*j*))
definition (*column*::'*n* =>'*a*^'*n*^'*m* =>'*a*^'*m*) *j A* = (χ *i*. ((*A*$*i*)$*j*))
definition *rows*(*A*::'*a*^'*n*^'*m*) = { *row i A* | *i*. *i* ∈ (*UNIV* :: '*m set*)}
definition *columns*(*A*::'*a*^'*n*^'*m*) = { *column i A* | *i*. *i* ∈ (*UNIV* :: '*n set*)}

lemma *mat-0*[*simp*]: *mat 0* = *0* **by** (*vector mat-def*)
lemma *matrix-add-ldistrib*: (*A* ** (*B* + *C*)) = (*A* * *B*) + (*A* * *C*)
by (*vector matrix-matrix-mult-def setsum-addf*[*symmetric*] *ring-simps*)

lemma *setsum-delta'*:
assumes *fS*: *finite S* **shows**
setsum (λ*k*. *if a* = *k then b k else 0*) *S* =
(*if a*∈ *S then b a else 0*)
using *setsum-delta*[*OF fS*, *of a b*, *symmetric*]
by (*auto intro*: *setsum-cong*)

lemma *matrix-mul-lid*:
fixes *A* :: '*a*::*semiring-1* ^ '*m* ^ '*n*::*finite*
shows *mat 1* ** *A* = *A*
apply (*simp add*: *matrix-matrix-mult-def mat-def*)
apply *vector*
by (*auto simp only*: *cond-value-iff cond-application-beta setsum-delta'*[*OF finite*]
mult-1-left mult-zero-left if-True UNIV-I)

lemma *matrix-mul-rid*:
fixes *A* :: '*a*::*semiring-1* ^ '*m*::*finite* ^ '*n*
shows *A* ** *mat 1* = *A*
apply (*simp add*: *matrix-matrix-mult-def mat-def*)
apply *vector*
by (*auto simp only*: *cond-value-iff cond-application-beta setsum-delta*[*OF finite*]
mult-1-right mult-zero-right if-True UNIV-I cong: *if-cong*)

lemma *matrix-mul-assoc*: *A* ** (*B* ** *C*) = (*A* ** *B*) ** *C*
apply (*vector matrix-matrix-mult-def setsum-right-distrib setsum-left-distrib mult-assoc*)
apply (*subst setsum-commute*)
apply *simp*
done

lemma *matrix-vector-mul-assoc*: *A* *v* (*B* *v* *x*) = (*A* ** *B*) *v* *x*
apply (*vector matrix-matrix-mult-def matrix-vector-mult-def setsum-right-distrib*
setsum-left-distrib mult-assoc)
apply (*subst setsum-commute*)
apply *simp*
done

lemma *matrix-vector-mul-lid*: *mat 1* *v* *x* = (*x*::'*a*::*semiring-1* ^ '*n*::*finite*)
apply (*vector matrix-vector-mult-def mat-def*)
by (*simp add*: *cond-value-iff cond-application-beta*

setsum-delta' cong del: *if-weak-cong*)

lemma *matrix-transp-mul*: *transp*(*A* ** *B*) = *transp B* ** *transp* (*A*::'*a*::*comm-semiring-1*^'*m*^'*n*)
by (*simp add*: *matrix-matrix-mult-def transp-def Cart-eq mult-commute*)

lemma *matrix-eq*:
fixes *A B* :: '*a*::*semiring-1* ^ '*n*::*finite* ^ '*m*
shows *A* = *B* ⟷ (∀ *x*. *A* *v *x* = *B* *v *x*) (**is** *?lhs* ⟷ *?rhs*)
apply *auto*
apply (*subst Cart-eq*)
apply *clarify*
apply (*clarsimp simp add*: *matrix-vector-mult-def basis-def cond-value-iff cond-application-beta Cart-eq cong del*: *if-weak-cong*)
apply (*erule-tac x*=*basis ia* **in** *allE*)
apply (*erule-tac x*=*i* **in** *allE*)
by (*auto simp add*: *basis-def cond-value-iff cond-application-beta setsum-delta*[*OF finite*] *cong del*: *if-weak-cong*)

lemma *matrix-vector-mul-component*:
shows ((*A*::'*a*::*semiring-1*^'*n*'^'*m*) *v *x*)$*k* = (*A*$*k*) · *x*
by (*simp add*: *matrix-vector-mult-def dot-def*)

lemma *dot-lmul-matrix*: ((*x*::'*a*::*comm-semiring-1* ^'*n*) *v** *A*) · *y* = *x* · (*A* *v *y*)
apply (*simp add*: *dot-def matrix-vector-mult-def vector-matrix-mult-def setsum-left-distrib setsum-right-distrib mult-ac*)
apply (*subst setsum-commute*)
by *simp*

lemma *transp-mat*: *transp* (*mat n*) = *mat n*
by (*vector transp-def mat-def*)

lemma *transp-transp*: *transp*(*transp A*) = *A*
by (*vector transp-def*)

lemma *row-transp*:
fixes *A*:: '*a*::*semiring-1*^'*n*^'*m*
shows *row i* (*transp A*) = *column i A*
by (*simp add*: *row-def column-def transp-def Cart-eq*)

lemma *column-transp*:
fixes *A*:: '*a*::*semiring-1*^'*n*^'*m*
shows *column i* (*transp A*) = *row i A*
by (*simp add*: *row-def column-def transp-def Cart-eq*)

lemma *rows-transp*: *rows*(*transp* (*A*::'*a*::*semiring-1*^'*n*^'*m*)) = *columns A*
by (*auto simp add*: *rows-def columns-def row-transp intro*: *set-ext*)

lemma *columns-transp*: *columns*(*transp* (*A*::'*a*::*semiring-1*^'*n*^'*m*)) = *rows A* **by** (*metis transp-transp rows-transp*)

Two sometimes fruitful ways of looking at matrix-vector multiplication.

lemma *matrix-mult-dot*: $A *v x = (\chi\ i.\ A\$i \cdot x)$
by (*simp add*: *matrix-vector-mult-def dot-def*)

lemma *matrix-mult-vsum*: $(A::'a::comm\text{-}semiring\text{-}1\hat{}'n\hat{}'m) *v x = setsum\ (\lambda i.\ (x\$i) *s column\ i\ A)\ (UNIV::'n\ set)$
by (*simp add*: *matrix-vector-mult-def Cart-eq column-def mult-commute*)

lemma *vector-componentwise*:
$(x::'a::ring\text{-}1\hat{}'n::finite) = (\chi\ j.\ setsum\ (\lambda i.\ (x\$i) * (basis\ i :: 'a\hat{}'n)\$j)\ (UNIV :: 'n\ set))$
apply (*subst basis-expansion*[*symmetric*])
by (*vector Cart-eq setsum-component*)

lemma *linear-componentwise*:
fixes $f:: 'a::ring\text{-}1\ \hat{}\ 'm::finite \Rightarrow 'a\ \hat{}\ 'n$
assumes *lf*: *linear f*
shows $(f\ x)\$j = setsum\ (\lambda i.\ (x\$i) * (f\ (basis\ i)\$j))\ (UNIV :: 'm\ set)$ (**is** *?lhs = ?rhs*)
proof−
let $?M = (UNIV :: 'm\ set)$
let $?N = (UNIV :: 'n\ set)$
have *fM*: *finite ?M* **by** *simp*
have $?rhs = (setsum\ (\lambda i.(x\$i) *s f\ (basis\ i)\ )\ ?M)\$j$
unfolding *vector-smult-component*[*symmetric*]
unfolding *setsum-component*[*of* $(\lambda i.(x\$i) *s f\ (basis\ i :: 'a\hat{}'m))\ ?M$]
..
then show *?thesis* **unfolding** *linear-setsum-mul*[*OF lf fM*, *symmetric*] *basis-expansion*
..
qed

Inverse matrices (not necessarily square)

definition $invertible(A::'a::semiring\text{-}1\hat{}'n\hat{}'m) \longleftrightarrow (\exists A'::'a\hat{}'m\hat{}'n.\ A ** A' = mat\ 1 \wedge A' ** A = mat\ 1)$

definition $matrix\text{-}inv(A::'a::semiring\text{-}1\hat{}'n\hat{}'m) =$
$(SOME\ A'::'a\hat{}'m\hat{}'n.\ A ** A' = mat\ 1 \wedge A' ** A = mat\ 1)$

Correspondence between matrices and linear operators.

definition $matrix::('a::\{plus,times,\ one,\ zero\}\hat{}'m \Rightarrow 'a\ \hat{}\ 'n) \Rightarrow 'a\hat{}'m\hat{}'n$
where $matrix\ f = (\chi\ i\ j.\ (f(basis\ j))\$i)$

lemma *matrix-vector-mul-linear*: $linear(\lambda x.\ A *v (x::'a::comm\text{-}semiring\text{-}1\ \hat{}\ 'n))$
by (*simp add*: *linear-def matrix-vector-mult-def Cart-eq ring-simps setsum-right-distrib setsum-addf*)

lemma *matrix-works*: **assumes** *lf*: *linear f* **shows** $matrix\ f *v x = f\ (x::'a::comm\text{-}ring\text{-}1\ \hat{}\ 'n::finite)$
apply (*simp add*: *matrix-def matrix-vector-mult-def Cart-eq mult-commute*)

apply *clarify*
apply (*rule linear-componentwise*[*OF lf*, *symmetric*])
done

lemma *matrix-vector-mul*: *linear f* ==> *f* = (λ*x*. *matrix f* ∗*v* (*x*::'*a*::*comm-ring-1* ^ '*n*::*finite*)) **by** (*simp add*: *ext matrix-works*)

lemma *matrix-of-matrix-vector-mul*: *matrix*(λ*x*. *A* ∗*v* (*x* :: '*a*:: *comm-ring-1* ^ '*n*::*finite*)) = *A*
by (*simp add*: *matrix-eq matrix-vector-mul-linear matrix-works*)

lemma *matrix-compose*:
assumes *lf*: *linear* (*f*::'*a*::*comm-ring-1*^'*n*::*finite* ⇒ '*a*^'*m*::*finite*)
and *lg*: *linear* (*g*::'*a*::*comm-ring-1*^'*m*::*finite* ⇒ '*a*^'*k*)
shows *matrix* (*g o f*) = *matrix g* ∗∗ *matrix f*
using *lf lg linear-compose*[*OF lf lg*] *matrix-works*[*OF linear-compose*[*OF lf lg*]]
by (*simp add*: *matrix-eq matrix-works matrix-vector-mul-assoc*[*symmetric*] *o-def*)

lemma *matrix-vector-column*:(*A*::'*a*::*comm-semiring-1*^'*n*^'*m*) ∗*v x* = *setsum* (λ*i*. (*x*$*i*) ∗*s* ((*transp A*)$*i*)) (*UNIV*:: '*n set*)
by (*simp add*: *matrix-vector-mult-def transp-def Cart-eq mult-commute*)

lemma *adjoint-matrix*: *adjoint*(λ*x*. (*A*::'*a*::*comm-ring-1*^'*n*::*finite*^'*m*::*finite*) ∗*v x*) = (λ*x*. *transp A* ∗*v x*)
apply (*rule adjoint-unique*[*symmetric*])
apply (*rule matrix-vector-mul-linear*)
apply (*simp add*: *transp-def dot-def matrix-vector-mult-def setsum-left-distrib setsum-right-distrib*)
apply (*subst setsum-commute*)
apply (*auto simp add*: *mult-ac*)
done

lemma *matrix-adjoint*: **assumes** *lf*: *linear* (*f* :: '*a*::*comm-ring-1*^'*n*::*finite* ⇒ '*a* ^ '*m*::*finite*)
shows *matrix*(*adjoint f*) = *transp*(*matrix f*)
apply (*subst matrix-vector-mul*[*OF lf*])
unfolding *adjoint-matrix matrix-of-matrix-vector-mul* **..**

30.17 Interlude: Some properties of real sets

lemma *seq-mono-lemma*: **assumes** ∀(*n*::*nat*) ≥ *m*. (*d n* :: *real*) < *e n* **and** ∀ *n* ≥ *m*. *e n* <= *e m*
shows ∀ *n* ≥ *m*. *d n* < *e m*
using *prems* **apply** *auto*
apply (*erule-tac x*=*n* **in** *allE*)
apply (*erule-tac x*=*n* **in** *allE*)
apply *auto*
done

lemma *real-convex-bound-lt*:
assumes *xa*: $(x::real) < a$ **and** *ya*: $y < a$ **and** *u*: $0 <= u$ **and** *v*: $0 <= v$
and *uv*: $u + v = 1$
shows $u * x + v * y < a$
proof−
have *uv'*: $u = 0 \longrightarrow v \neq 0$ **using** *u v uv* **by** *arith*
have $a = a * (u + v)$ **unfolding** *uv* **by** *simp*
hence *th*: $u * a + v * a = a$ **by** (*simp add*: *ring-simps*)
from *xa u* **have** $u \neq 0 \Longrightarrow u*x < u*a$ **by** (*simp add*: *mult-compare-simps*)
from *ya v* **have** $v \neq 0 \Longrightarrow v * y < v * a$ **by** (*simp add*: *mult-compare-simps*)
from *xa ya u v* **have** $u * x + v * y < u * a + v * a$
apply (*cases* $u = 0$, *simp-all add*: *uv'*)
apply(*rule mult-strict-left-mono*)
using *uv'* **apply** *simp-all*

apply (*rule add-less-le-mono*)
apply(*rule mult-strict-left-mono*)
apply *simp-all*
apply (*rule mult-left-mono*)
apply *simp-all*
done
thus *?thesis* **unfolding** *th* **.**
qed

lemma *real-convex-bound-le*:
assumes *xa*: $(x::real) \leq a$ **and** *ya*: $y \leq a$ **and** *u*: $0 <= u$ **and** *v*: $0 <= v$
and *uv*: $u + v = 1$
shows $u * x + v * y \leq a$
proof−
from *xa ya u v* **have** $u * x + v * y \leq u * a + v * a$ **by** (*simp add*: *add-mono mult-left-mono*)
also have $\ldots \leq (u + v) * a$ **by** (*simp add*: *ring-simps*)
finally show *?thesis* **unfolding** *uv* **by** *simp*
qed

lemma *infinite-enumerate*: **assumes** *fS*: *infinite S*
shows $\exists r.$ *subseq* $r \wedge (\forall n.\ r\ n \in S)$
unfolding *subseq-def*
using *enumerate-in-set*[*OF fS*] *enumerate-mono*[*of - - S*] *fS* **by** *auto*

lemma *approachable-lt-le*: $(\exists (d::real)>0.\ \forall x.\ f\ x < d \longrightarrow P\ x) \longleftrightarrow (\exists d>0.\ \forall x.\ f\ x \leq d \longrightarrow P\ x)$
apply *auto*
apply (*rule-tac x*=*d*/*2* **in** *exI*)
apply *auto*
done

lemma *triangle-lemma*:
assumes *x*: *0* <= (*x*::*real*) **and** *y*:*0* <= *y* **and** *z*: *0* <= *z* **and** *xy*: *x^2* <= *y^2* + *z^2*
shows *x* <= *y* + *z*
proof−
have *y^2* + *z^2* ≤ *y^2* + *2***y***z* + *z^2* **using** *z y* **by** (*simp add*: *zero-compare-simps*)
with *xy* **have** *th*: *x ^2* ≤ (*y*+*z*)*^2* **by** (*simp add*: *power2-eq-square ring-simps*)
from *y z* **have** *yz*: *y* + *z* ≥ *0* **by** *arith*
from *power2-le-imp-le*[*OF th yz*] **show** *?thesis* **.**
qed

lemma *lambda-skolem*: (∀ *i*. ∃ *x*. *P i x*) ⟷
(∃ *x*::*'a ^ 'n*. ∀ *i*. *P i* (*x*$*i*)) (**is** *?lhs* ⟷ *?rhs*)
proof−
let *?S* = (*UNIV* :: *'n set*)
{assume *H*: *?rhs*
then have *?lhs* **by** *auto***}**
moreover
{assume *H*: *?lhs*
then obtain *f* **where** *f*:∀ *i*. *P i* (*f i*) **unfolding** *choice-iff* **by** *metis*
let *?x* = (χ *i*. (*f i*)) :: *'a ^ 'n*
{fix *i*
from *f* **have** *P i* (*f i*) **by** *metis*
then have *P i* (*?x*$*i*) **by** *auto*
}
hence ∀ *i*. *P i* (*?x*$*i*) **by** *metis*
hence *?rhs* **by** *metis* **}**
ultimately show *?thesis* **by** *metis*
qed

definition *rsup*:: *real set* ⇒ *real* **where**
rsup S = (*SOME a*. *isLub UNIV S a*)

lemma *rsup-alt*: *rsup S* = (*SOME a*. (∀ *x* ∈ *S*. *x* ≤ *a*) ∧ (∀ *b*. (∀ *x* ∈ *S*. *x* ≤ *b*) ⟶ *a* ≤ *b*)) **by** (*auto simp add*: *isLub-def rsup-def leastP-def isUb-def setle-def setge-def*)

lemma *rsup*: **assumes** *Se*: *S* ≠ {} **and** *b*: ∃ *b*. *S* *<= *b*
shows *isLub UNIV S* (*rsup S*)
using *Se b*
unfolding *rsup-def*
apply *clarify*
apply (*rule someI-ex*)
apply (*rule reals-complete*)
by (*auto simp add*: *isUb-def setle-def*)

lemma *rsup-le*: **assumes** *Se*: $S \neq \{\}$ **and** *Sb*: $S *<= b$ **shows** *rsup* $S \leq b$
proof−
from *Sb* **have** *bu*: *isUb UNIV S b* **by** (*simp add*: *isUb-def setle-def*)
from *rsup*[*OF Se*] *Sb* **have** *isLub UNIV S* (*rsup S*) **by** *blast*
then show *?thesis* **using** *bu* **by** (*auto simp add*: *isLub-def leastP-def setle-def setge-def*)
qed

lemma *rsup-finite-Max*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows *rsup S* = *Max S*
using *fS Se*
proof−
let *?m* = *Max S*
from *Max-ge*[*OF fS*] **have** *Sm*: $\forall\ x{\in}\ S.\ x \leq ?m$ **by** *metis*
with *rsup*[*OF Se*] **have** *lub*: *isLub UNIV S* (*rsup S*) **by** (*metis setle-def*)
from *Max-in*[*OF fS Se*] *lub* **have** *mrS*: *?m* ≤ *rsup S*
by (*auto simp add*: *isLub-def leastP-def setle-def setge-def isUb-def*)
moreover
have *rsup S* ≤ *?m* **using** *Sm lub*
by (*auto simp add*: *isLub-def leastP-def isUb-def setle-def setge-def*)
ultimately show *?thesis* **by** *arith*
qed

lemma *rsup-finite-in*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows *rsup S* ∈ *S*
using *rsup-finite-Max*[*OF fS Se*] *Max-in*[*OF fS Se*] **by** *metis*

lemma *rsup-finite-Ub*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows *isUb S S* (*rsup S*)
using *rsup-finite-Max*[*OF fS Se*] *rsup-finite-in*[*OF fS Se*] *Max-ge*[*OF fS*]
unfolding *isUb-def setle-def* **by** *metis*

lemma *rsup-finite-ge-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a \leq$ *rsup S* $\longleftrightarrow$ $(\exists\ x \in S.\ a \leq x)$
using *rsup-finite-Ub*[*OF fS Se*] **by** (*auto simp add*: *isUb-def setle-def*)

lemma *rsup-finite-le-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a \geq$ *rsup S* $\longleftrightarrow$ $(\forall\ x \in S.\ a \geq x)$
using *rsup-finite-Ub*[*OF fS Se*] **by** (*auto simp add*: *isUb-def setle-def*)

lemma *rsup-finite-gt-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a <$ *rsup S* $\longleftrightarrow$ $(\exists\ x \in S.\ a < x)$
using *rsup-finite-Ub*[*OF fS Se*] **by** (*auto simp add*: *isUb-def setle-def*)

lemma *rsup-finite-lt-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a >$ *rsup S* $\longleftrightarrow$ $(\forall\ x \in S.\ a > x)$
using *rsup-finite-Ub*[*OF fS Se*] **by** (*auto simp add*: *isUb-def setle-def*)

lemma *rsup-unique*: **assumes** b: S *<= b **and** S: $\forall b' < b.\ \exists x \in S.\ b' < x$
  shows *rsup* $S = b$
using b S
unfolding *setle-def rsup-alt*
apply $-$
apply (*rule some-equality*)
apply (*metis linorder-not-le order-eq-iff*[*symmetric*])+
done

lemma *rsup-le-subset*: $S \neq \{\} \Longrightarrow S \subseteq T \Longrightarrow (\exists b.\ T$ *<= $b) \Longrightarrow$ *rsup* $S \leq$ *rsup* T
  apply (*rule rsup-le*)
  apply *simp*
   using *rsup*[*of* T] **by** (*auto simp add*: *isLub-def leastP-def setge-def setle-def isUb-def*)

lemma *isUb-def'*: *isUb* R $S = (\lambda x.\ S$ *<= $x \wedge x \in R$)
  apply (*rule ext*)
  by (*metis isUb-def*)

lemma *UNIV-trivial*: *UNIV* x **using** *UNIV-I*[*of* x] **by** (*metis mem-def*)
lemma *rsup-bounds*: **assumes** *Se*: $S \neq \{\}$ **and** l: a <=* S **and** u: S *<= b
  shows $a \leq$ *rsup* $S \wedge$ *rsup* $S \leq b$
proof−
  from *rsup*[*OF Se*] u **have** *lub*: *isLub UNIV S* (*rsup S*) **by** *blast*
  hence b: *rsup* $S \leq b$ **using** u **by** (*auto simp add*: *isLub-def leastP-def setle-def setge-def isUb-def'*)
  from *Se* **obtain** y **where** y: $y \in S$ **by** *blast*
  from *lub* l **have** $a \leq$ *rsup* S **apply** (*auto simp add*: *isLub-def leastP-def setle-def setge-def isUb-def'*)
    apply (*erule ballE*[**where** x=y])
    apply (*erule ballE*[**where** x=y])
    apply *arith*
    using y **apply** *auto*
    done
  with b **show** *?thesis* **by** *blast*
qed

lemma *rsup-abs-le*: $S \neq \{\} \Longrightarrow (\forall x{\in}S.\ |x| \leq a) \Longrightarrow |$*rsup* $S| \leq a$
  unfolding *abs-le-interval-iff* **using** *rsup-bounds*[*of* S $-a$ a]
  by (*auto simp add*: *setge-def setle-def*)

lemma *rsup-asclose*: **assumes** S:$S \neq \{\}$ **and** b: $\forall x{\in}S.\ |x - l| \leq e$ **shows** $|$*rsup* $S - l| \leq e$
proof−
  have *th*: $\bigwedge$(x::*real*) l e. $|x - l| \leq e \longleftrightarrow l - e \leq x \wedge x \leq l + e$ **by** *arith*
  show *?thesis* **using** S b *rsup-bounds*[*of* S l $-$ e l+e] **unfolding** *th*
    by (*auto simp add*: *setge-def setle-def*)
qed

definition *rinf*:: *real set* ⇒ *real* **where**
rinf S = (*SOME a*. *isGlb UNIV S a*)

lemma *rinf-alt*: *rinf S* = (*SOME a*. (∀ *x* ∈ *S*. *x* ≥ *a*) ∧ (∀ *b*. (∀ *x* ∈ *S*. *x* ≥ *b*) ⟶ *a* ≥ *b*)) **by** (*auto simp add*: *isGlb-def rinf-def greatestP-def isLb-def setle-def setge-def*)

lemma *reals-complete-Glb*: **assumes** *Se*: ∃ *x*. *x* ∈ *S* **and** *lb*: ∃ *y*. *isLb UNIV S y*
shows ∃ (*t*::*real*). *isGlb UNIV S t*
proof−
let *?M* = *uminus* ' *S*
from *lb* **have** *th*: ∃ *y*. *isUb UNIV ?M y* **apply** (*auto simp add*: *isUb-def isLb-def setle-def setge-def*)
by (*rule-tac x*=−*y* **in** *exI*, *auto*)
from *Se* **have** *Me*: ∃ *x*. *x* ∈ *?M* **by** *blast*
from *reals-complete*[*OF Me th*] **obtain** *t* **where** *t*: *isLub UNIV ?M t* **by** *blast*
have *isGlb UNIV S* (− *t*) **using** *t*
apply (*auto simp add*: *isLub-def isGlb-def leastP-def greatestP-def setle-def setge-def isUb-def isLb-def*)
apply (*erule-tac x*=−*y* **in** *allE*)
apply *auto*
done
then show *?thesis* **by** *metis*
qed

lemma *rinf*: **assumes** *Se*: *S* ≠ {} **and** *b*: ∃ *b*. *b* <=∗ *S*
shows *isGlb UNIV S* (*rinf S*)
using *Se b*
unfolding *rinf-def*
apply *clarify*
apply (*rule someI-ex*)
apply (*rule reals-complete-Glb*)
apply (*auto simp add*: *isLb-def setle-def setge-def*)
done

lemma *rinf-ge*: **assumes** *Se*: *S* ≠ {} **and** *Sb*: *b* <=∗ *S* **shows** *rinf S* ≥ *b*
proof−
from *Sb* **have** *bu*: *isLb UNIV S b* **by** (*simp add*: *isLb-def setge-def*)
from *rinf*[*OF Se*] *Sb* **have** *isGlb UNIV S* (*rinf S*) **by** *blast*
then show *?thesis* **using** *bu* **by** (*auto simp add*: *isGlb-def greatestP-def setle-def setge-def*)
qed

lemma *rinf-finite-Min*: **assumes** *fS*: *finite S* **and** *Se*: *S* ≠ {}
shows *rinf S* = *Min S*
using *fS Se*
proof−
let *?m* = *Min S*

from *Min-le*[*OF fS*] **have** *Sm*: $\forall\ x{\in}\ S.\ x \geq ?m$ **by** *metis*
with *rinf*[*OF Se*] **have** *glb*: *isGlb UNIV S* (*rinf S*) **by** (*metis setge-def*)
from *Min-in*[*OF fS Se*] *glb* **have** *mrS*: $?m \geq$ *rinf S*
by (*auto simp add*: *isGlb-def greatestP-def setle-def setge-def isLb-def*)
moreover
have *rinf S* $\geq$ *?m* **using** *Sm glb*
by (*auto simp add*: *isGlb-def greatestP-def isLb-def setle-def setge-def*)
ultimately show *?thesis* **by** *arith*
qed

lemma *rinf-finite-in*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows *rinf S* $\in S$
using *rinf-finite-Min*[*OF fS Se*] *Min-in*[*OF fS Se*] **by** *metis*

lemma *rinf-finite-Lb*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows *isLb S S* (*rinf S*)
using *rinf-finite-Min*[*OF fS Se*] *rinf-finite-in*[*OF fS Se*] *Min-le*[*OF fS*]
unfolding *isLb-def setge-def* **by** *metis*

lemma *rinf-finite-ge-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a \leq$ *rinf S* $\longleftrightarrow (\forall\ x \in S.\ a \leq x)$
using *rinf-finite-Lb*[*OF fS Se*] **by** (*auto simp add*: *isLb-def setge-def*)

lemma *rinf-finite-le-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a \geq$ *rinf S* $\longleftrightarrow (\exists\ x \in S.\ a \geq x)$
using *rinf-finite-Lb*[*OF fS Se*] **by** (*auto simp add*: *isLb-def setge-def*)

lemma *rinf-finite-gt-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a <$ *rinf S* $\longleftrightarrow (\forall\ x \in S.\ a < x)$
using *rinf-finite-Lb*[*OF fS Se*] **by** (*auto simp add*: *isLb-def setge-def*)

lemma *rinf-finite-lt-iff*: **assumes** *fS*: *finite S* **and** *Se*: $S \neq \{\}$
shows $a >$ *rinf S* $\longleftrightarrow (\exists\ x \in S.\ a > x)$
using *rinf-finite-Lb*[*OF fS Se*] **by** (*auto simp add*: *isLb-def setge-def*)

lemma *rinf-unique*: **assumes** *b*: $b <=* S$ **and** *S*: $\forall\, b' > b.\ \exists\, x \in S.\ b' > x$
shows *rinf S* $= b$
using *b S*
unfolding *setge-def rinf-alt*
apply $-$
apply (*rule some-equality*)
apply (*metis linorder-not-le order-eq-iff*[*symmetric*])+
done

lemma *rinf-ge-subset*: $S{\neq}\{\} \Longrightarrow S \subseteq T \Longrightarrow (\exists\, b.\ b <=* T) \Longrightarrow$ *rinf S* $>=$ *rinf T*
apply (*rule rinf-ge*)
apply *simp*
using *rinf*[*of T*] **by** (*auto simp add*: *isGlb-def greatestP-def setge-def setle-def*

isLb-def)

lemma *isLb-def'*: *isLb R S* = (λ*x*. *x* <=∗ *S* ∧ *x* ∈ *R*)
apply (*rule ext*)
by (*metis isLb-def*)

lemma *rinf-bounds*: **assumes** *Se*: *S* ≠ {} **and** *l*: *a* <=∗ *S* **and** *u*: *S* ∗<= *b*
shows *a* ≤ *rinf S* ∧ *rinf S* ≤ *b*
proof−
from *rinf*[*OF Se*] *l* **have** *lub*: *isGlb UNIV S* (*rinf S*) **by** *blast*
hence *b*: *a* ≤ *rinf S* **using** *l* **by** (*auto simp add*: *isGlb-def greatestP-def setle-def setge-def isLb-def'*)
from *Se* **obtain** *y* **where** *y*: *y* ∈ *S* **by** *blast*
from *lub u* **have** *b* ≥ *rinf S* **apply** (*auto simp add*: *isGlb-def greatestP-def setle-def setge-def isLb-def'*)
apply (*erule ballE*[**where** *x*=*y*])
apply (*erule ballE*[**where** *x*=*y*])
apply *arith*
using *y* **apply** *auto*
done
with *b* **show** *?thesis* **by** *blast*
qed

lemma *rinf-abs-ge*: *S* ≠ {} ⟹ (∀ *x*∈*S*. |*x*| ≤ *a*) ⟹ |*rinf S*| ≤ *a*
unfolding *abs-le-interval-iff* **using** *rinf-bounds*[*of S* −*a a*]
by (*auto simp add*: *setge-def setle-def*)

lemma *rinf-asclose*: **assumes** *S*:*S* ≠ {} **and** *b*: ∀ *x*∈*S*. |*x* − *l*| ≤ *e* **shows** |*rinf S* − *l*| ≤ *e*
proof−
have *th*: ⋀(*x*::*real*) *l e*. |*x* − *l*| ≤ *e* ⟷ *l* − *e* ≤ *x* ∧ *x* ≤ *l* + *e* **by** *arith*
show *?thesis* **using** *S b rinf-bounds*[*of S l* − *e l*+*e*] **unfolding** *th*
by (*auto simp add*: *setge-def setle-def*)
qed

30.18 Operator norm.

definition *onorm f* = *rsup* {*norm* (*f x*)| *x*. *norm x* = *1*}

lemma *norm-bound-generalize*:
fixes *f*:: *real* ^'*n*::*finite* ⇒ *real*^'*m*::*finite*
assumes *lf*: *linear f*
shows (∀ *x*. *norm x* = *1* ⟶ *norm* (*f x*) ≤ *b*) ⟷ (∀ *x*. *norm* (*f x*) ≤ *b* ∗ *norm x*) (**is** *?lhs* ⟷ *?rhs*)
proof−
{**assume** *H*: *?rhs*
{**fix** *x* :: *real*^'*n* **assume** *x*: *norm x* = *1*
from *H*[*rule-format*, *of x*] *x* **have** *norm* (*f x*) ≤ *b* **by** *simp*}
then have *?lhs* **by** *blast* }

```
  moreover
  {assume H: ?lhs
   from H[rule-format, of basis arbitrary]
   have bp: b ≥ 0 using norm-ge-zero[of f (basis arbitrary)]
    by (auto simp add: norm-basis elim: order-trans [OF norm-ge-zero])
   {fix x :: real ^'n
    {assume x = 0
     then have norm (f x) ≤ b * norm x by (simp add: linear-0[OF lf] bp)}
    moreover
    {assume x0: x ≠ 0
     hence n0: norm x ≠ 0 by (metis norm-eq-zero)
     let ?c = 1/ norm x
     have norm (?c*s x) = 1 using x0 by (simp add: n0 norm-mul)
     with H have norm (f(?c*s x)) ≤ b by blast
     hence ?c * norm (f x) ≤ b
      by (simp add: linear-cmul[OF lf] norm-mul)
     hence norm (f x) ≤ b * norm x
      using n0 norm-ge-zero[of x] by (auto simp add: field-simps)}
    ultimately have norm (f x) ≤ b * norm x by blast}
   then have ?rhs by blast}
  ultimately show ?thesis by blast
qed

lemma onorm:
  fixes f:: real ^'n::finite ⇒ real ^'m::finite
  assumes lf: linear f
  shows norm (f x) <= onorm f * norm x
  and ∀ x. norm (f x) <= b * norm x ⟹ onorm f <= b
proof−
  {
   let ?S = {norm (f x) |x. norm x = 1}
   have Se: ?S ≠ {} using  norm-basis by auto
   from linear-bounded[OF lf] have b: ∃ b. ?S *<= b
      unfolding norm-bound-generalize[OF lf, symmetric] by (auto simp add:
setle-def)
   {from rsup[OF Se b, unfolded onorm-def[symmetric]]
    show norm (f x) <= onorm f * norm x
      apply −
      apply (rule spec[where x = x])
      unfolding norm-bound-generalize[OF lf, symmetric]
      by (auto simp add: isLub-def isUb-def leastP-def setge-def setle-def)}
   {
    show ∀ x. norm (f x) <= b * norm x ⟹ onorm f <= b
      using rsup[OF Se b, unfolded onorm-def[symmetric]]
      unfolding norm-bound-generalize[OF lf, symmetric]
      by (auto simp add: isLub-def isUb-def leastP-def setge-def setle-def)}
  }
qed
```

lemma *onorm-pos-le*: **assumes** *lf*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real ^'m*::*finite*)
shows *0 <= onorm f*
using *order-trans*[*OF norm-ge-zero onorm*(*1*)[*OF lf*, *of basis arbitrary*], *unfolded norm-basis*] **by** *simp*

lemma *onorm-eq-0*: **assumes** *lf*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real ^'m*::*finite*)
shows *onorm f = 0* ⟷ (∀ *x*. *f x = 0*)
using *onorm*[*OF lf*]
apply (*auto simp add*: *onorm-pos-le*)
apply *atomize*
apply (*erule allE*[**where** *x=0*::*real*])
using *onorm-pos-le*[*OF lf*]
apply *arith*
done

lemma *onorm-const*: *onorm*(λ*x*::*real^'n*::*finite*. (*y*::*real ^ 'm*::*finite*)) = *norm y*
proof−
let *?f* = λ*x*::*real^'n*. (*y*::*real ^ 'm*)
have *th*: {*norm* (*?f x*)| *x*. *norm x = 1*} = {*norm y*}
by(*auto intro*: *vector-choose-size set-ext*)
show *?thesis*
unfolding *onorm-def th*
apply (*rule rsup-unique*) **by** (*simp-all add*: *setle-def*)
qed

lemma *onorm-pos-lt*: **assumes** *lf*: *linear* (*f*::*real ^ 'n*::*finite* ⇒ *real ^'m*::*finite*)
shows *0 < onorm f* ⟷ ~(∀ *x*. *f x = 0*)
unfolding *onorm-eq-0*[*OF lf*, *symmetric*]
using *onorm-pos-le*[*OF lf*] **by** *arith*

lemma *onorm-compose*:
assumes *lf*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real ^'m*::*finite*)
and *lg*: *linear* (*g*::*real^'k*::*finite* ⇒ *real^'n*::*finite*)
shows *onorm* (*f o g*) <= *onorm f* ∗ *onorm g*
apply (*rule onorm*(*2*)[*OF linear-compose*[*OF lg lf*], *rule-format*])
unfolding *o-def*
apply (*subst mult-assoc*)
apply (*rule order-trans*)
apply (*rule onorm*(*1*)[*OF lf*])
apply (*rule mult-mono1*)
apply (*rule onorm*(*1*)[*OF lg*])
apply (*rule onorm-pos-le*[*OF lf*])
done

lemma *onorm-neg-lemma*: **assumes** *lf*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real^'m*::*finite*)
shows *onorm* (λ*x*. − *f x*) ≤ *onorm f*
using *onorm*[*OF linear-compose-neg*[*OF lf*]] *onorm*[*OF lf*]
unfolding *norm-minus-cancel* **by** *metis*

lemma *onorm-neg*: **assumes** *lf*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real^'m*::*finite*)
shows *onorm* (λ*x*. − *f x*) = *onorm f*
using *onorm-neg-lemma*[*OF lf*] *onorm-neg-lemma*[*OF linear-compose-neg*[*OF lf*]]
by *simp*

lemma *onorm-triangle*:
assumes *lf*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real ^'m*::*finite*) **and** *lg*: *linear g*
shows *onorm* (λ*x*. *f x* + *g x*) <= *onorm f* + *onorm g*
apply(*rule onorm*(*2*)[*OF linear-compose-add*[*OF lf lg*], *rule-format*])
apply (*rule order-trans*)
apply (*rule norm-triangle-ineq*)
apply (*simp add*: *distrib*)
apply (*rule add-mono*)
apply (*rule onorm*(*1*)[*OF lf*])
apply (*rule onorm*(*1*)[*OF lg*])
done

lemma *onorm-triangle-le*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real ^'m*::*finite*) ⟹ *linear g* ⟹ *onorm*(*f*) + *onorm*(*g*) <= *e*
⟹ *onorm*(λ*x*. *f x* + *g x*) <= *e*
apply (*rule order-trans*)
apply (*rule onorm-triangle*)
apply *assumption*+
done

lemma *onorm-triangle-lt*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real ^'m*::*finite*) ⟹ *linear g* ⟹ *onorm*(*f*) + *onorm*(*g*) < *e*
==> *onorm*(λ*x*. *f x* + *g x*) < *e*
apply (*rule order-le-less-trans*)
apply (*rule onorm-triangle*)
by *assumption*+

definition *vec1*:: *'a* ⇒ *'a ^ 1* **where** *vec1 x* = (χ *i*. *x*)

definition *dest-vec1*:: *'a ^1* ⇒ *'a*
where *dest-vec1 x* = (*x*$*1*)

lemma *vec1-component*[*simp*]: (*vec1 x*)$*1* = *x*
by (*simp add*: *vec1-def*)

lemma *vec1-dest-vec1*[*simp*]: *vec1*(*dest-vec1 x*) = *x dest-vec1*(*vec1 y*) = *y*
by (*simp-all add*: *vec1-def dest-vec1-def Cart-eq forall-1*)

lemma *forall-vec1*: (∀ *x*. *P x*) ⟷ (∀ *x*. *P* (*vec1 x*)) **by** (*metis vec1-dest-vec1*)

lemma *exists-vec1*: (∃ *x*. *P x*) ⟷ (∃ *x*. *P*(*vec1 x*)) **by** (*metis vec1-dest-vec1*)

lemma *forall-dest-vec1*: $(\forall\, x.\ P\ x) \longleftrightarrow (\forall\, x.\ P(\textit{dest-vec1}\ x))$ **by** (*metis vec1-dest-vec1*)

lemma *exists-dest-vec1*: $(\exists\, x.\ P\ x) \longleftrightarrow (\exists\, x.\ P(\textit{dest-vec1}\ x))$**by** (*metis vec1-dest-vec1*)

lemma *vec1-eq*[*simp*]: $\textit{vec1}\ x = \textit{vec1}\ y \longleftrightarrow x = y$ **by** (*metis vec1-dest-vec1*)

lemma *dest-vec1-eq*[*simp*]: $\textit{dest-vec1}\ x = \textit{dest-vec1}\ y \longleftrightarrow x = y$ **by** (*metis vec1-dest-vec1*)

lemma *vec1-in-image-vec1*: $\textit{vec1}\ x \in (\textit{vec1}\ `\ S) \longleftrightarrow x \in S$ **by** *auto*

lemma *vec1-vec*: $\textit{vec1}\ x = \textit{vec}\ x$ **by** (*vector vec1-def*)

lemma *vec1-add*: $\textit{vec1}(x + y) = \textit{vec1}\ x + \textit{vec1}\ y$ **by** (*vector vec1-def*)
lemma *vec1-sub*: $\textit{vec1}(x - y) = \textit{vec1}\ x - \textit{vec1}\ y$ **by** (*vector vec1-def*)
lemma *vec1-cmul*: $\textit{vec1}(c* x) = c *s \textit{vec1}\ x$ **by** (*vector vec1-def*)
lemma *vec1-neg*: $\textit{vec1}(- x) = - \textit{vec1}\ x$ **by** (*vector vec1-def*)

lemma *vec1-setsum*: **assumes** *fS*: *finite S*
shows $\textit{vec1}(\textit{setsum}\ f\ S) = \textit{setsum}\ (\textit{vec1}\ o\ f)\ S$
apply (*induct rule*: *finite-induct*[*OF fS*])
apply (*simp add*: *vec1-vec*)
apply (*auto simp add*: *vec1-add*)
done

lemma *dest-vec1-lambda*: $\textit{dest-vec1}(\chi\ i.\ x\ i) = x\ 1$
by (*simp add*: *dest-vec1-def*)

lemma *dest-vec1-vec*: $\textit{dest-vec1}(\textit{vec}\ x) = x$
by (*simp add*: *vec1-vec*[*symmetric*])

lemma *dest-vec1-add*: $\textit{dest-vec1}(x + y) = \textit{dest-vec1}\ x + \textit{dest-vec1}\ y$
by (*metis vec1-dest-vec1 vec1-add*)

lemma *dest-vec1-sub*: $\textit{dest-vec1}(x - y) = \textit{dest-vec1}\ x - \textit{dest-vec1}\ y$
by (*metis vec1-dest-vec1 vec1-sub*)

lemma *dest-vec1-cmul*: $\textit{dest-vec1}(c*sx) = c * \textit{dest-vec1}\ x$
by (*metis vec1-dest-vec1 vec1-cmul*)

lemma *dest-vec1-neg*: $\textit{dest-vec1}(- x) = - \textit{dest-vec1}\ x$
by (*metis vec1-dest-vec1 vec1-neg*)

lemma *dest-vec1-0*[*simp*]: $\textit{dest-vec1}\ 0 = 0$ **by** (*metis vec-0 dest-vec1-vec*)

lemma *dest-vec1-sum*: **assumes** *fS*: *finite S*
shows $\textit{dest-vec1}(\textit{setsum}\ f\ S) = \textit{setsum}\ (\textit{dest-vec1}\ o\ f)\ S$
apply (*induct rule*: *finite-induct*[*OF fS*])
apply (*simp add*: *dest-vec1-vec*)

apply (*auto simp add*: *dest-vec1-add*)
done

lemma *norm-vec1*: *norm*(*vec1 x*) = *abs*(*x*)
by (*simp add*: *vec1-def norm-real*)

lemma *dist-vec1*: *dist*(*vec1 x*) (*vec1 y*) = *abs*(*x* − *y*)
by (*simp only*: *dist-real vec1-component*)
lemma *abs-dest-vec1*: *norm x* = |*dest-vec1 x*|
by (*metis vec1-dest-vec1 norm-vec1*)

lemma *linear-vmul-dest-vec1*:
fixes *f*:: *'a*::*semiring-1^'n* ⇒ *'a^1*
shows *linear f* ⟹ *linear* (λ*x*. *dest-vec1*(*f x*) ∗*s v*)
unfolding *dest-vec1-def*
apply (*rule linear-vmul-component*)
by *auto*

lemma *linear-from-scalars*:
assumes *lf*: *linear* (*f*::*'a*::*comm-ring-1 ^1* ⇒ *'a^'n*)
shows *f* = (λ*x*. *dest-vec1 x* ∗*s column 1* (*matrix f*))
apply (*rule ext*)
apply (*subst matrix-works*[*OF lf*, *symmetric*])
apply (*auto simp add*: *Cart-eq matrix-vector-mult-def dest-vec1-def column-def mult-commute UNIV-1*)
done

lemma *linear-to-scalars*: **assumes** *lf*: *linear* (*f*::*'a*::*comm-ring-1 ^'n*::*finite* ⇒ *'a^1*)
shows *f* = (λ*x*. *vec1*(*row 1* (*matrix f*) · *x*))
apply (*rule ext*)
apply (*subst matrix-works*[*OF lf*, *symmetric*])
apply (*simp add*: *Cart-eq matrix-vector-mult-def vec1-def row-def dot-def mult-commute forall-1*)
done

lemma *dest-vec1-eq-0*: *dest-vec1 x* = *0* ⟷ *x* = *0*
by (*simp add*: *dest-vec1-eq*[*symmetric*])

lemma *setsum-scalars*: **assumes** *fS*: *finite S*
shows *setsum f S* = *vec1* (*setsum* (*dest-vec1 o f*) *S*)
unfolding *vec1-setsum*[*OF fS*] **by** *simp*

lemma *dest-vec1-wlog-le*: (⋀(*x*::*'a*::*linorder ^ 1*) *y*. *P x y* ⟷ *P y x*) ⟹ (⋀*x y*. *dest-vec1 x* <= *dest-vec1 y* ==> *P x y*) ⟹ *P x y*
apply (*cases dest-vec1 x* ≤ *dest-vec1 y*)
apply *simp*
apply (*subgoal-tac dest-vec1 y* ≤ *dest-vec1 x*)
apply (*auto*)
done

Pasting vectors.

lemma *linear-fstcart*: *linear fstcart*
by (*auto simp add*: *linear-def Cart-eq*)

lemma *linear-sndcart*: *linear sndcart*
by (*auto simp add*: *linear-def Cart-eq*)

lemma *fstcart-vec*[*simp*]: *fstcart*(*vec x*) = *vec x*
by (*simp add*: *Cart-eq*)

lemma *fstcart-add*[*simp*]:*fstcart*(*x* + *y*) = *fstcart* (*x*::'*a*::{*plus*,*times*}^('*b* + '*c*)) + *fstcart y*
by (*simp add*: *Cart-eq*)

lemma *fstcart-cmul*[*simp*]:*fstcart*(*c*∗*s x*) = *c*∗*s fstcart* (*x*::'*a*::{*plus*,*times*}^('*b* + '*c*))
by (*simp add*: *Cart-eq*)

lemma *fstcart-neg*[*simp*]:*fstcart*(− *x*) = − *fstcart* (*x*::'*a*::*ring-1*^('*b* + '*c*))
by (*simp add*: *Cart-eq*)

lemma *fstcart-sub*[*simp*]:*fstcart*(*x* − *y*) = *fstcart* (*x*::'*a*::*ring-1*^('*b* + '*c*)) − *fstcart y*
by (*simp add*: *Cart-eq*)

lemma *fstcart-setsum*:
fixes *f*:: '*d* ⇒ '*a*::*semiring-1*^-
assumes *fS*: *finite S*
shows *fstcart* (*setsum f S*) = *setsum* (λ*i*. *fstcart* (*f i*)) *S*
by (*induct rule*: *finite-induct*[*OF fS*], *simp-all add*: *vec-0*[*symmetric*] *del*: *vec-0*)

lemma *sndcart-vec*[*simp*]: *sndcart*(*vec x*) = *vec x*
by (*simp add*: *Cart-eq*)

lemma *sndcart-add*[*simp*]:*sndcart*(*x* + *y*) = *sndcart* (*x*::'*a*::{*plus*,*times*}^('*b* + '*c*)) + *sndcart y*
by (*simp add*: *Cart-eq*)

lemma *sndcart-cmul*[*simp*]:*sndcart*(*c*∗*s x*) = *c*∗*s sndcart* (*x*::'*a*::{*plus*,*times*}^('*b* + '*c*))
by (*simp add*: *Cart-eq*)

lemma *sndcart-neg*[*simp*]:*sndcart*(− *x*) = − *sndcart* (*x*::'*a*::*ring-1*^('*b* + '*c*))
by (*simp add*: *Cart-eq*)

lemma *sndcart-sub*[*simp*]:*sndcart*(*x* − *y*) = *sndcart* (*x*::'*a*::*ring-1*^('*b* + '*c*)) − *sndcart y*
by (*simp add*: *Cart-eq*)

lemma *sndcart-setsum*:
fixes *f*:: *'d* ⇒ *'a*::*semiring-1^*-
assumes *fS*: *finite S*
shows *sndcart* (*setsum f S*) = *setsum* (λ*i*. *sndcart* (*f i*)) *S*
by (*induct rule*: *finite-induct*[*OF fS*], *simp-all add*: *vec-0*[*symmetric*] *del*: *vec-0*)

lemma *pastecart-vec*[*simp*]: *pastecart* (*vec x*) (*vec x*) = *vec x*
by (*simp add*: *pastecart-eq fstcart-pastecart sndcart-pastecart*)

lemma *pastecart-add*[*simp*]:*pastecart* (*x1*::*'a*::{*plus*,*times*}^-) *y1* + *pastecart x2 y2* = *pastecart* (*x1* + *x2*) (*y1* + *y2*)
by (*simp add*: *pastecart-eq fstcart-pastecart sndcart-pastecart*)

lemma *pastecart-cmul*[*simp*]: *pastecart* (*c* ∗*s* (*x1*::*'a*::{*plus*,*times*}^-)) (*c* ∗*s y1*) = *c* ∗*s pastecart x1 y1*
by (*simp add*: *pastecart-eq fstcart-pastecart sndcart-pastecart*)

lemma *pastecart-neg*[*simp*]: *pastecart* (− (*x*::*'a*::*ring-1^*-)) (− *y*) = − *pastecart x y*
unfolding *vector-sneg-minus1 pastecart-cmul* **..**

lemma *pastecart-sub*: *pastecart* (*x1*::*'a*::*ring-1^*-) *y1* − *pastecart x2 y2* = *pastecart* (*x1* − *x2*) (*y1* − *y2*)
by (*simp add*: *diff-def pastecart-neg*[*symmetric*] *del*: *pastecart-neg*)

lemma *pastecart-setsum*:
fixes *f*:: *'d* ⇒ *'a*::*semiring-1^*-
assumes *fS*: *finite S*
shows *pastecart* (*setsum f S*) (*setsum g S*) = *setsum* (λ*i*. *pastecart* (*f i*) (*g i*)) *S*
by (*simp add*: *pastecart-eq fstcart-setsum*[*OF fS*] *sndcart-setsum*[*OF fS*] *fstcart-pastecart sndcart-pastecart*)

lemma *setsum-Plus*:
⟦*finite A*; *finite B*⟧ ⟹
(∑ *x*∈*A* <+> *B*. *g x*) = (∑ *x*∈*A*. *g* (*Inl x*)) + (∑ *x*∈*B*. *g* (*Inr x*))
unfolding *Plus-def*
by (*subst setsum-Un-disjoint*, *auto simp add*: *setsum-reindex*)

lemma *setsum-UNIV-sum*:
fixes *g* :: *'a*::*finite* + *'b*::*finite* ⇒ -
shows (∑ *x*∈*UNIV*. *g x*) = (∑ *x*∈*UNIV*. *g* (*Inl x*)) + (∑ *x*∈*UNIV*. *g* (*Inr x*))
apply (*subst UNIV-Plus-UNIV* [*symmetric*])
apply (*rule setsum-Plus* [*OF finite finite*])
done

lemma *norm-fstcart*: *norm*(*fstcart x*) <= *norm* (*x*::*real* ^(*'n*::*finite* + *'m*::*finite*))
proof−
have *th0*: *norm x* = *norm* (*pastecart* (*fstcart x*) (*sndcart x*))
by (*simp add*: *pastecart-fst-snd*)

have *th1*: *fstcart x · fstcart x ≤ pastecart (fstcart x) (sndcart x) · pastecart (fstcart x) (sndcart x)*
by (*simp add*: *dot-def setsum-UNIV-sum pastecart-def setsum-nonneg*)
then show *?thesis*
unfolding *th0*
unfolding *real-vector-norm-def real-sqrt-le-iff id-def*
by (*simp add*: *dot-def*)
qed

lemma *dist-fstcart*: *dist(fstcart (x::real^-)) (fstcart y) <= dist x y*
by (*metis dist-def fstcart-sub*[*symmetric*] *norm-fstcart*)

lemma *norm-sndcart*: *norm(sndcart x) <= norm (x::real ^('n::finite + 'm::finite))*
proof−
have *th0*: *norm x = norm (pastecart (fstcart x) (sndcart x))*
by (*simp add*: *pastecart-fst-snd*)
have *th1*: *sndcart x · sndcart x ≤ pastecart (fstcart x) (sndcart x) · pastecart (fstcart x) (sndcart x)*
by (*simp add*: *dot-def setsum-UNIV-sum pastecart-def setsum-nonneg*)
then show *?thesis*
unfolding *th0*
unfolding *real-vector-norm-def real-sqrt-le-iff id-def*
by (*simp add*: *dot-def*)
qed

lemma *dist-sndcart*: *dist(sndcart (x::real^-)) (sndcart y) <= dist x y*
by (*metis dist-def sndcart-sub*[*symmetric*] *norm-sndcart*)

lemma *dot-pastecart*: *(pastecart (x1::'a::{times,comm-monoid-add} ^'n::finite) (x2::'a::{times,comm-monoid-*
· (pastecart y1 y2) = x1 · y1 + x2 · y2
by (*simp add*: *dot-def setsum-UNIV-sum pastecart-def*)

lemma *norm-pastecart*: *norm(pastecart x y) <= norm(x :: real ^ 'm::finite) + norm(y::real^'n::finite)*
unfolding *real-vector-norm-def dot-pastecart real-sqrt-le-iff id-def*
apply (*rule power2-le-imp-le*)
apply (*simp add*: *real-sqrt-pow2*[*OF add-nonneg-nonneg*[*OF dot-pos-le*[*of x*] *dot-pos-le*[*of y*]]])
apply (*auto simp add*: *power2-eq-square ring-simps*)
apply (*simp add*: *power2-eq-square*[*symmetric*])
apply (*rule mult-nonneg-nonneg*)
apply (*simp-all add*: *real-sqrt-pow2*[*OF dot-pos-le*])
apply (*rule add-nonneg-nonneg*)
apply (*simp-all add*: *real-sqrt-pow2*[*OF dot-pos-le*])
done

30.19 A generic notion of "hull" (convex, affine, conic hull and closure).

definition *hull* :: *'a set set* ⇒ *'a set* ⇒ *'a set* (**infixl** *hull* 75) **where**
S hull s = *Inter* {*t*. *t* ∈ *S* ∧ *s* ⊆ *t*}

lemma *hull-same*: *s* ∈ *S* ⟹ *S hull s* = *s*
unfolding *hull-def* **by** *auto*

lemma *hull-in*: (⋀*T*. *T* ⊆ *S* ==> *Inter T* ∈ *S*) ==> (*S hull s*) ∈ *S*
unfolding *hull-def subset-iff* **by** *auto*

lemma *hull-eq*: (⋀*T*. *T* ⊆ *S* ==> *Inter T* ∈ *S*) ==> (*S hull s*) = *s* ⟷ *s* ∈ *S*
using *hull-same*[*of s S*] *hull-in*[*of S s*] **by** *metis*

lemma *hull-hull*: *S hull* (*S hull s*) = *S hull s*
unfolding *hull-def* **by** *blast*

lemma *hull-subset*: *s* ⊆ (*S hull s*)
unfolding *hull-def* **by** *blast*

lemma *hull-mono*: *s* ⊆ *t* ==> (*S hull s*) ⊆ (*S hull t*)
unfolding *hull-def* **by** *blast*

lemma *hull-antimono*: *S* ⊆ *T* ==> (*T hull s*) ⊆ (*S hull s*)
unfolding *hull-def* **by** *blast*

lemma *hull-minimal*: *s* ⊆ *t* ⟹ *t* ∈ *S* ==> (*S hull s*) ⊆ *t*
unfolding *hull-def* **by** *blast*

lemma *subset-hull*: *t* ∈ *S* ==> *S hull s* ⊆ *t* ⟷ *s* ⊆ *t*
unfolding *hull-def* **by** *blast*

lemma *hull-unique*: *s* ⊆ *t* ⟹ *t* ∈ *S* ⟹ (⋀*t'*. *s* ⊆ *t'* ⟹ *t'* ∈ *S* ==> *t* ⊆ *t'*)
==> (*S hull s* = *t*)
unfolding *hull-def* **by** *auto*

lemma *hull-induct*: (⋀*x*. *x*∈ *S* ⟹ *P x*) ⟹ *Q* {*x*. *P x*} ⟹ ∀ *x*∈ *Q hull S*. *P x*
using *hull-minimal*[*of S* {*x*. *P x*} *Q*]
by (*auto simp add*: *subset-eq Collect-def mem-def*)

lemma *hull-inc*: *x* ∈ *S* ⟹ *x* ∈ *P hull S* **by** (*metis hull-subset subset-eq*)

lemma *hull-union-subset*: (*S hull s*) ∪ (*S hull t*) ⊆ (*S hull* (*s* ∪ *t*))
unfolding *Un-subset-iff* **by** (*metis hull-mono Un-upper1 Un-upper2*)

lemma *hull-union*: **assumes** *T*: ⋀*T*. *T* ⊆ *S* ==> *Inter T* ∈ *S*
shows *S hull* (*s* ∪ *t*) = *S hull* (*S hull s* ∪ *S hull t*)
apply *rule*

apply (*rule hull-mono*)
unfolding *Un-subset-iff*
apply (*metis hull-subset Un-upper1 Un-upper2 subset-trans*)
apply (*rule hull-minimal*)
apply (*metis hull-union-subset*)
apply (*metis hull-in T*)
done

lemma *hull-redundant-eq*: $a \in (S\ hull\ s) \longleftrightarrow (S\ hull\ (insert\ a\ s) = S\ hull\ s)$
unfolding *hull-def* **by** *blast*

lemma *hull-redundant*: $a \in (S\ hull\ s) ==> (S\ hull\ (insert\ a\ s) = S\ hull\ s)$
by (*metis hull-redundant-eq*)

Archimedian properties and useful consequences.

lemma *real-arch-simple*: $\exists n.\ x <= real\ (n::nat)$
using *reals-Archimedean2*[*of x*] **apply** *auto* **by** (*rule-tac x=Suc n* **in** *exI*, *auto*)
lemmas *real-arch-lt = reals-Archimedean2*

lemmas *real-arch = reals-Archimedean3*

lemma *real-arch-inv*: $0 < e \longleftrightarrow (\exists n::nat.\ n \neq 0 \wedge 0 < inverse\ (real\ n) \wedge inverse\ (real\ n) < e)$
using *reals-Archimedean*
apply (*auto simp add*: *field-simps inverse-positive-iff-positive*)
apply (*subgoal-tac inverse* (*real n*) > *0*)
apply *arith*
apply *simp*
done

lemma *real-pow-lbound*: $0 <= x ==> 1 + real\ n * x <= (1 + x)\ \hat{}\ n$
proof(*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
hence *h*: $1 + real\ n * x \leq (1 + x)\ \hat{}\ n$ **by** *simp*
from *h* **have** *p*: $1 \leq (1 + x)\ \hat{}\ n$ **using** *Suc.prems* **by** *simp*
from *h* **have** $1 + real\ n * x + x \leq (1 + x)\ \hat{}\ n + x$ **by** *simp*
also have $\ldots \leq (1 + x)\ \hat{}\ Suc\ n$ **apply** (*subst diff-le-0-iff-le*[*symmetric*])
apply (*simp add*: *ring-simps*)
using *mult-left-mono*[*OF p Suc.prems*] **by** *simp*
finally show *?case* **by** (*simp add*: *real-of-nat-Suc ring-simps*)
qed

lemma *real-arch-pow*: **assumes** *x*: $1 < (x::real)$ **shows** $\exists n.\ y < x\hat{}n$
proof−
from *x* **have** *x0*: $x - 1 > 0$ **by** *arith*
from *real-arch*[*OF x0*, *rule-format*, *of y*]
obtain *n*::*nat* **where** *n*:$y < real\ n * (x - 1)$ **by** *metis*

from *x0* **have** *x00*: $x - 1 \geq 0$ **by** *arith*
from *real-pow-lbound*[*OF x00*, *of n*] *n*
have $y < x \hat{} n$ **by** *auto*
then show *?thesis* **by** *metis*
qed

lemma *real-arch-pow2*: $\exists n.\ (x::real) < 2 \hat{} n$
using *real-arch-pow*[*of 2 x*] **by** *simp*

lemma *real-arch-pow-inv*: **assumes** *y*: $(y::real) > 0$ **and** *x1*: $x < 1$
shows $\exists n.\ x \hat{} n < y$
proof−
{assume *x0*: $x > 0$
from *x0 x1* **have** *ix*: $1 < 1/x$ **by** (*simp add*: *field-simps*)
from *real-arch-pow*[*OF ix*, *of* $1/y$]
obtain *n* **where** *n*: $1/y < (1/x) \hat{} n$ **by** *blast*
then
have *?thesis* **using** *y x0* **by** (*auto simp add*: *field-simps power-divide*) **}**
moreover
{assume $\neg\ x > 0$ **with** *y x1* **have** *?thesis* **apply** *auto* **by** (*rule exI*[**where** *x*=*1*], *auto*)**}**
ultimately show *?thesis* **by** *metis*
qed

lemma *forall-pos-mono*: $(\bigwedge d\ e::real.\ d < e \Longrightarrow P\ d \Longrightarrow P\ e) \Longrightarrow (\bigwedge n::nat.\ n \neq 0 \Longrightarrow P(inverse(real\ n))) \Longrightarrow (\bigwedge e.\ 0 < e \Longrightarrow P\ e)$
by (*metis real-arch-inv*)

lemma *forall-pos-mono-1*: $(\bigwedge d\ e::real.\ d < e \Longrightarrow P\ d \Longrightarrow P\ e) \Longrightarrow (\bigwedge n.\ P(inverse(real\ (Suc\ n)))) \Longrightarrow 0 < e \Longrightarrow P\ e$
apply (*rule forall-pos-mono*)
apply *auto*
apply (*atomize*)
apply (*erule-tac* x=$n - 1$ **in** *allE*)
apply *auto*
done

lemma *real-archimedian-rdiv-eq-0*: **assumes** *x0*: $x \geq 0$ **and** *c*: $c \geq 0$ **and** *xc*: $\forall (m::nat) > 0.\ real\ m * x \leq c$
shows $x = 0$
proof−
{assume $x \neq 0$ **with** *x0* **have** *xp*: $x > 0$ **by** *arith*
from *real-arch*[*OF xp*, *rule-format*, *of c*] **obtain** *n*::*nat* **where** *n*: $c < real\ n * x$ **by** *blast*
with *xc*[*rule-format*, *of n*] **have** $n = 0$ **by** *arith*
with *n c* **have** *False* **by** *simp***}**
then show *?thesis* **by** *blast*
qed

lemma *real-max-rsup*: *max x y = rsup {x,y}*
proof−
have *f*: *finite {x, y} {x,y} ≠ {}* **by** *simp-all*
from *rsup-finite-le-iff*[*OF f*, *of max x y*] **have** *rsup {x,y} ≤ max x y* **by** *simp*
moreover
have *max x y ≤ rsup {x,y}* **using** *rsup-finite-ge-iff*[*OF f*, *of max x y*]
by (*simp add*: *linorder-linear*)
ultimately show *?thesis* **by** *arith*
qed

lemma *real-min-rinf*: *min x y = rinf {x,y}*
proof−
have *f*: *finite {x, y} {x,y} ≠ {}* **by** *simp-all*
from *rinf-finite-le-iff*[*OF f*, *of min x y*] **have** *rinf {x,y} ≤ min x y*
by (*simp add*: *linorder-linear*)
moreover
have *min x y ≤ rinf {x,y}* **using** *rinf-finite-ge-iff*[*OF f*, *of min x y*]
by *simp*
ultimately show *?thesis* **by** *arith*
qed

lemma *sum-gp-basic*: *((1::'a::{field, recpower}) − x) ∗ setsum (λi. x^i) {0 .. n}*
= (1 − x^(Suc n))
(**is** *?lhs = ?rhs*)
proof−
{assume *x1*: *x = 1* **hence** *?thesis* **by** *simp***}**
moreover
{assume *x1*: *x≠1*
hence *x1'*: *x − 1 ≠ 0 1 − x ≠ 0 x − 1 = − (1 − x) − (1 − x) ≠ 0* **by** *auto*
from *geometric-sum*[*OF x1*, *of Suc n*, *unfolded x1'*]
have *(− (1 − x)) ∗ setsum (λi. x^i) {0 .. n} = − (1 − x^(Suc n))*
unfolding *atLeastLessThanSuc-atLeastAtMost*
using *x1'* **apply** (*auto simp only*: *field-simps*)
apply (*simp add*: *ring-simps*)
done
then have *?thesis* **by** (*simp add*: *ring-simps*) **}**
ultimately show *?thesis* **by** *metis*
qed

lemma *sum-gp-multiplied*: **assumes** *mn*: *m <= n*
shows *((1::'a::{field, recpower}) − x) ∗ setsum (op ^ x) {m..n} = x^m − x^*

Suc n
(**is** *?lhs = ?rhs*)
proof−
let *?S* = {*0*..(*n* − *m*)}
from *mn* **have** *mn'*: *n* − *m* ≥ *0* **by** *arith*
let *?f* = *op* + *m*
have *i*: *inj-on ?f ?S* **unfolding** *inj-on-def* **by** *auto*
have *f*: *?f* ' *?S* = {*m*..*n*}
using *mn* **apply** (*auto simp add*: *image-iff Bex-def*) **by** *arith*
have *th*: *op* ^ *x o op* + *m* = (λ*i*. *x*^*m* ∗ *x*^*i*)
by (*rule ext*, *simp add*: *power-add power-mult*)
from *setsum-reindex*[*OF i*, *of op* ^ *x*, *unfolded f th setsum-right-distrib*[*symmetric*]]
have *?lhs* = *x*^*m* ∗ ((*1* − *x*) ∗ *setsum* (*op* ^ *x*) {*0*..*n* − *m*}) **by** *simp*
then show *?thesis* **unfolding** *sum-gp-basic* **using** *mn*
by (*simp add*: *ring-simps power-add*[*symmetric*])
qed

lemma *sum-gp*: *setsum* (*op* ^ (*x*::'*a*::{*field*, *recpower*})) {*m* .. *n*} =
(*if n* < *m then 0 else if x* = *1 then of-nat* ((*n* + *1*) − *m*)
else (*x*^ *m* − *x*^ (*Suc n*)) / (*1* − *x*))
proof−
{assume *nm*: *n* < *m* **hence** *?thesis* **by** *simp***}**
moreover
{assume ¬ *n* < *m* **hence** *nm*: *m* ≤ *n* **by** *arith*
{assume *x*: *x* = *1* **hence** *?thesis* **by** *simp***}**
moreover
{assume *x*: *x* ≠ *1* **hence** *nz*: *1* − *x* ≠ *0* **by** *simp*
from *sum-gp-multiplied*[*OF nm*, *of x*] *nz* **have** *?thesis* **by** (*simp add*: *field-simps*)**}**
ultimately have *?thesis* **by** *metis*
}
ultimately show *?thesis* **by** *metis*
qed

lemma *sum-gp-offset*: *setsum* (*op* ^ (*x*::'*a*::{*field*,*recpower*})) {*m* .. *m*+*n*} =
(*if x* = *1 then of-nat n* + *1 else x*^*m* ∗ (*1* − *x*^*Suc n*) / (*1* − *x*))
unfolding *sum-gp*[*of x m m* + *n*] *power-Suc*
by (*simp add*: *ring-simps power-add*)

30.20 A bit of linear algebra.

definition *subspace S* ⟷ *0* ∈ *S* ∧ (∀ *x*∈ *S*. ∀ *y* ∈*S*. *x* + *y* ∈ *S*) ∧ (∀ *c*. ∀ *x* ∈*S*. *c* ∗*s x* ∈*S*)
definition *span S* = (*subspace hull S*)
definition *dependent S* ⟷ (∃ *a* ∈ *S*. *a* ∈ *span*(*S* − {*a*}))
abbreviation *independent s* == ~(*dependent s*)

lemma *subspace-UNIV*[*simp*]: *subspace*(*UNIV*) **by** (*simp add*: *subspace-def*)

lemma *subspace-0*: *subspace S* ==> *0* ∈ *S* **by** (*metis subspace-def*)

lemma *subspace-add*: *subspace S* ⟹ *x* ∈ *S* ⟹ *y* ∈ *S* ==> *x* + *y* ∈ *S*
by (*metis subspace-def*)

lemma *subspace-mul*: *subspace S* ⟹ *x* ∈ *S* ⟹ *c* ∗*s x* ∈ *S*
by (*metis subspace-def*)

lemma *subspace-neg*: *subspace S* ⟹ (*x*::'*a*::*ring-1*^'*n*) ∈ *S* ⟹ − *x* ∈ *S*
by (*metis vector-sneg-minus1 subspace-mul*)

lemma *subspace-sub*: *subspace S* ⟹ (*x*::'*a*::*ring-1*^'*n*) ∈ *S* ⟹ *y* ∈ *S* ⟹ *x* − *y* ∈ *S*
by (*metis diff-def subspace-add subspace-neg*)

lemma *subspace-setsum*:
assumes *sA*: *subspace A* **and** *fB*: *finite B*
and *f*: ∀ *x*∈ *B*. *f x* ∈ *A*
shows *setsum f B* ∈ *A*
using *fB f sA*
apply(*induct rule*: *finite-induct*[*OF fB*])
by (*simp add*: *subspace-def sA*, *auto simp add*: *sA subspace-add*)

lemma *subspace-linear-image*:
assumes *lf*: *linear* (*f*::'*a*::*semiring-1*^'*n* ⇒ -) **and** *sS*: *subspace S*
shows *subspace*(*f* ' *S*)
using *lf sS linear-0*[*OF lf*]
unfolding *linear-def subspace-def*
apply (*auto simp add*: *image-iff*)
apply (*rule-tac x*=*x* + *y* **in** *bexI*, *auto*)
apply (*rule-tac x*=*c*∗*s x* **in** *bexI*, *auto*)
done

lemma *subspace-linear-preimage*: *linear* (*f*::'*a*::*semiring-1*^'*n* ⇒ -) ==> *subspace S* ==> *subspace* {*x*. *f x* ∈ *S*}
by (*auto simp add*: *subspace-def linear-def linear-0*[*of f*])

lemma *subspace-trivial*: *subspace* {*0*::'*a*::*semiring-1* ^-}
by (*simp add*: *subspace-def*)

lemma *subspace-inter*: *subspace A* ⟹ *subspace B* ==> *subspace* (*A* ∩ *B*)
by (*simp add*: *subspace-def*)

lemma *span-mono*: *A* ⊆ *B* ==> *span A* ⊆ *span B*
by (*metis span-def hull-mono*)

lemma *subspace-span*: *subspace*(*span S*)
unfolding *span-def*
apply (*rule hull-in*[*unfolded mem-def*])
apply (*simp only*: *subspace-def Inter-iff Int-iff subset-eq*)
apply *auto*
apply (*erule-tac x*=*X* **in** *ballE*)
apply (*simp add*: *mem-def*)
apply *blast*
apply (*erule-tac x*=*X* **in** *ballE*)
apply (*erule-tac x*=*X* **in** *ballE*)
apply (*erule-tac x*=*X* **in** *ballE*)
apply (*clarsimp simp add*: *mem-def*)
apply *simp*
apply *simp*
apply *simp*
apply (*erule-tac x*=*X* **in** *ballE*)
apply (*erule-tac x*=*X* **in** *ballE*)
apply (*simp add*: *mem-def*)
apply *simp*
apply *simp*
done

lemma *span-clauses*:
$a \in S \Longrightarrow a \in span\ S$
$0 \in span\ S$
$x \in span\ S \Longrightarrow y \in span\ S \Longrightarrow x + y \in span\ S$
$x \in span\ S \Longrightarrow c *s x \in span\ S$
by (*metis span-def hull-subset subset-eq subspace-span subspace-def*)+

lemma *span-induct*: **assumes** *SP*: $\bigwedge x.\ x \in S \Longrightarrow P\ x$
and *P*: *subspace P* **and** *x*: $x \in span\ S$ **shows** $P\ x$
proof−
from *SP* **have** *SP'*: $S \subseteq P$ **by** (*simp add*: *mem-def subset-eq*)
from *P* **have** *P'*: $P \in subspace$ **by** (*simp add*: *mem-def*)
from *x hull-minimal*[*OF SP' P'*, *unfolded span-def*[*symmetric*]]
show $P\ x$ **by** (*metis mem-def subset-eq*)
qed

lemma *span-empty*: *span* {} = {(*0*::'*a*::*semiring-0* ^ '*n*)}
apply (*simp add*: *span-def*)
apply (*rule hull-unique*)
apply (*auto simp add*: *mem-def subspace-def*)
unfolding *mem-def*[*of* 0::'*a*^'*n*, *symmetric*]
apply *simp*
done

lemma *independent-empty*: *independent* {}
by (*simp add*: *dependent-def*)

lemma *independent-mono*: *independent A* $\Longrightarrow$ $B \subseteq A$ ==> *independent B*
apply (*clarsimp simp add*: *dependent-def span-mono*)
apply (*subgoal-tac span* $(B - \{a\}) \leq$ *span* $(A - \{a\})$)
apply *force*
apply (*rule span-mono*)
apply *auto*
done

lemma *span-subspace*: $A \subseteq B \Longrightarrow B \leq$ *span A* $\Longrightarrow$ *subspace B* $\Longrightarrow$ *span A* $= B$
by (*metis order-antisym span-def hull-minimal mem-def*)

lemma *span-induct'*: **assumes** *SP*: $\forall x \in S.\ P\ x$
and *P*: *subspace P* **shows** $\forall x \in$ *span S*. *P x*
using *span-induct SP P* **by** *blast*

inductive *span-induct-alt-help* **for** *S*:: *'a*::*semiring-1^'n* $\Rightarrow$ *bool*
where
span-induct-alt-help-0: *span-induct-alt-help S 0*
| *span-induct-alt-help-S*: $x \in S \Longrightarrow$ *span-induct-alt-help S z* $\Longrightarrow$ *span-induct-alt-help S* $(c *s\ x + z)$

lemma *span-induct-alt'*:
assumes *h0*: *h* $(0$::*'a*::*semiring-1^'n*$)$ **and** *hS*: $\bigwedge c\ x\ y.\ x \in S \Longrightarrow h\ y \Longrightarrow h\ (c*s\ x + y)$ **shows** $\forall x \in$ *span S*. *h x*
proof−
{fix *x*:: *'a^'n* **assume** *x*: *span-induct-alt-help S x*
have *h x*
apply (*rule span-induct-alt-help.induct*[*OF x*])
apply (*rule h0*)
apply (*rule hS*, *assumption*, *assumption*)
done}
note *th0* = *this*
{fix *x* **assume** *x*: $x \in$ *span S*

have *span-induct-alt-help S x*
proof(*rule span-induct*[**where** *x*=*x* **and** *S*=*S*])
show $x \in$ *span S* **using** *x* **.**
next
fix *x* **assume** *xS* : $x \in S$
from *span-induct-alt-help-S*[*OF xS span-induct-alt-help-0*, *of 1*]
show *span-induct-alt-help S x* **by** *simp*
next
have *span-induct-alt-help S 0* **by** (*rule span-induct-alt-help-0*)
moreover
{fix *x y* **assume** *h*: *span-induct-alt-help S x span-induct-alt-help S y*
from *h*
have *span-induct-alt-help S* $(x + y)$
apply (*induct rule*: *span-induct-alt-help.induct*)
apply *simp*

```
        unfolding add-assoc
        apply (rule span-induct-alt-help-S)
        apply assumption
        apply simp
        done}
    moreover
    {fix c x assume xt: span-induct-alt-help S x
     then have span-induct-alt-help S (c*s x)
       apply (induct rule: span-induct-alt-help.induct)
       apply (simp add: span-induct-alt-help-0)
       apply (simp add: vector-smult-assoc vector-add-ldistrib)
       apply (rule span-induct-alt-help-S)
       apply assumption
       apply simp
       done
    }
    ultimately show subspace (span-induct-alt-help S)
      unfolding subspace-def mem-def Ball-def by blast
  qed}
 with th0 show ?thesis by blast
qed

lemma span-induct-alt:
 assumes h0: h (0::'a::semiring-1^'n) and hS: ⋀c x y. x ∈ S ⟹ h y ⟹ h (c*s
x + y) and x: x ∈ span S
 shows h x
using span-induct-alt'[of h S] h0 hS x by blast



lemma span-superset: x ∈ S ==> x ∈ span S by (metis span-clauses)

lemma span-0: 0 ∈ span S by (metis subspace-span subspace-0)

lemma span-add: x ∈ span S ⟹ y ∈ span S ==> x + y ∈ span S
 by (metis subspace-add subspace-span)

lemma span-mul: x ∈ span S ==> (c *s x) ∈ span S
 by (metis subspace-span subspace-mul)

lemma span-neg: x ∈ span S ==> − (x::'a::ring-1^'n) ∈ span S
 by (metis subspace-neg subspace-span)

lemma span-sub: (x::'a::ring-1^'n) ∈ span S ⟹ y ∈ span S ==> x − y ∈ span
S
 by (metis subspace-span subspace-sub)

lemma span-setsum: finite A ⟹ ∀ x ∈ A. f x ∈ span S ==> setsum f A ∈ span
S
```

apply (*rule subspace-setsum*)
by (*metis subspace-span subspace-setsum*)+

lemma *span-add-eq*: (*x*::'*a*::*ring-1*^'*n*) ∈ *span S* ⟹ *x* + *y* ∈ *span S* ⟷ *y* ∈ *span S*
apply (*auto simp only*: *span-add span-sub*)
apply (*subgoal-tac* (*x* + *y*) − *x* ∈ *span S*, *simp*)
by (*simp only*: *span-add span-sub*)

lemma *span-linear-image*: **assumes** *lf*: *linear* (*f*::'*a*::*semiring-1* ^ '*n* => -)
shows *span* (*f* ` *S*) = *f* ` (*span S*)
proof−
{fix *x*
assume *x*: *x* ∈ *span* (*f* ` *S*)
have *x* ∈ *f* ` *span S*
apply (*rule span-induct*[**where** *x*=*x* **and** *S* = *f* ` *S*])
apply (*clarsimp simp add*: *image-iff*)
apply (*frule span-superset*)
apply *blast*
apply (*simp only*: *mem-def*)
apply (*rule subspace-linear-image*[*OF lf*])
apply (*rule subspace-span*)
apply (*rule x*)
done}
moreover
{fix *x* **assume** *x*: *x* ∈ *span S*
have *th0*:(λ*a*. *f a* ∈ *span* (*f* ` *S*)) = {*x*. *f x* ∈ *span* (*f* ` *S*)} **apply** (*rule set-ext*)
unfolding *mem-def Collect-def* **..**
have *f x* ∈ *span* (*f* ` *S*)
apply (*rule span-induct*[**where** *S*=*S*])
apply (*rule span-superset*)
apply *simp*
apply (*subst th0*)
apply (*rule subspace-linear-preimage*[*OF lf subspace-span*, *of f* ` *S*])
apply (*rule x*)
done}
ultimately show *?thesis* **by** *blast*
qed

lemma *span-breakdown*:
assumes *bS*: (*b*::'*a*::*ring-1* ^ '*n*) ∈ *S* **and** *aS*: *a* ∈ *span S*
shows ∃*k*. *a* − *k*∗*s b* ∈ *span* (*S* − {*b*}) (**is** *?P a*)
proof−
{fix *x* **assume** *xS*: *x* ∈ *S*
{assume *ab*: *x* = *b*

```
    then have ?P x
      apply simp
      apply (rule exI[where x=1], simp)
      by (rule span-0)}
  moreover
  {assume ab: x ≠ b
    then have ?P x  using xS
      apply −
      apply (rule exI[where x=0])
      apply (rule span-superset)
      by simp}
  ultimately have ?P x by blast}
 moreover have subspace ?P
  unfolding subspace-def
  apply auto
  apply (simp add: mem-def)
  apply (rule exI[where x=0])
  using span-0[of S − {b}]
  apply (simp add: mem-def)
  apply (clarsimp simp add: mem-def)
  apply (rule-tac x=k + ka in exI)
  apply (subgoal-tac x + y − (k + ka) ∗s b = (x − k∗s b) + (y − ka ∗s b))
  apply (simp only: )
  apply (rule span-add[unfolded mem-def])
  apply assumption+
  apply (vector ring-simps)
  apply (clarsimp simp add: mem-def)
  apply (rule-tac x= c∗k in exI)
  apply (subgoal-tac c ∗s x − (c ∗ k) ∗s b = c∗s (x − k∗s b))
  apply (simp only: )
  apply (rule span-mul[unfolded mem-def])
  apply assumption
  by (vector ring-simps)
 ultimately show ?P a using aS span-induct[where S=S and P= ?P] by
metis
qed

lemma span-breakdown-eq:
  (x::'a::ring-1^'n) ∈ span (insert a S) ⟷ (∃k. (x − k ∗s a) ∈ span S) (is ?lhs
⟷ ?rhs)
proof−
  {assume x: x ∈ span (insert a S)
    from x span-breakdown[of a insert a S x]
    have ?rhs apply clarsimp
      apply (rule-tac x= k in exI)
      apply (rule set-rev-mp[of - span (S − {a}) -])
      apply assumption
      apply (rule span-mono)
      apply blast
```

```
    done}
  moreover
  { fix k assume k: x − k ∗s a ∈ span S
    have eq: x = (x − k ∗s a) + k ∗s a by vector
    have (x − k ∗s a) + k ∗s a ∈ span (insert a S)
      apply (rule span-add)
      apply (rule set-rev-mp[of - span S -])
      apply (rule k)
      apply (rule span-mono)
      apply blast
      apply (rule span-mul)
      apply (rule span-superset)
      apply blast
      done
    then have ?lhs using eq by metis}
  ultimately show ?thesis by blast
qed



lemma in-span-insert:
  assumes a: (a::'a::field^'n) ∈ span (insert b S) and na: a ∉ span S
  shows b ∈ span (insert a S)
proof−
  from span-breakdown[of b insert b S a, OF insertI1 a]
  obtain k where k: a − k∗s b ∈ span (S − {b}) by auto
  {assume k0: k = 0
    with k have a ∈ span S
      apply (simp)
      apply (rule set-rev-mp)
      apply assumption
      apply (rule span-mono)
      apply blast
      done
    with na  have ?thesis by blast}
  moreover
  {assume k0: k ≠ 0
    have eq: b = (1/k) ∗s a − ((1/k) ∗s a − b) by vector
    from k0 have eq': (1/k) ∗s (a − k∗s b) = (1/k) ∗s a − b
      by (vector field-simps)
    from k have (1/k) ∗s (a − k∗s b) ∈ span (S − {b})
      by (rule span-mul)
    hence th: (1/k) ∗s a − b ∈ span (S − {b})
      unfolding eq' .

    from k
    have ?thesis
      apply (subst eq)
      apply (rule span-sub)
```

apply (*rule span-mul*)
apply (*rule span-superset*)
apply *blast*
apply (*rule set-rev-mp*)
apply (*rule th*)
apply (*rule span-mono*)
using *na* **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

lemma *in-span-delete*:
assumes *a*: (*a*::'*a*::*field*^'*n*) ∈ *span S*
and *na*: *a* ∉ *span* (*S*−{*b*})
shows *b* ∈ *span* (*insert a* (*S* − {*b*}))
apply (*rule in-span-insert*)
apply (*rule set-rev-mp*)
apply (*rule a*)
apply (*rule span-mono*)
apply *blast*
apply (*rule na*)
done

lemma *span-trans*:
assumes *x*: (*x*::'*a*::*ring-1*^'*n*) ∈ *span S* **and** *y*: *y* ∈ *span* (*insert x S*)
shows *y* ∈ *span S*
proof−
from *span-breakdown*[*of x insert x S y*, *OF insertI1 y*]
obtain *k* **where** *k*: *y* −*k*∗*s x* ∈ *span* (*S* − {*x*}) **by** *auto*
have *eq*: *y* = (*y* − *k* ∗*s x*) + *k* ∗*s x* **by** *vector*
show *?thesis*
apply (*subst eq*)
apply (*rule span-add*)
apply (*rule set-rev-mp*)
apply (*rule k*)
apply (*rule span-mono*)
apply *blast*
apply (*rule span-mul*)
by (*rule x*)
qed

lemma *span-explicit*:
span P = {*y*::'*a*::*semiring-1*^'*n*. ∃ *S u*. *finite S* ∧ *S* ⊆ *P* ∧ *setsum* (λ*v*. *u v* ∗*s v*) *S* = *y*}

```
  (is - = ?E is - = {y. ?h y} is - = {y. ∃S u. ?Q S u y})
proof−
  {fix x assume x: x ∈ ?E
   then obtain S u where fS: finite S and SP: S⊆P and u: setsum (λv. u v ∗s
v) S = x
     by blast
   have x ∈ span P
     unfolding u[symmetric]
     apply (rule span-setsum[OF fS])
     using span-mono[OF SP]
     by (auto intro: span-superset span-mul)}
  moreover
  have ∀x ∈ span P. x ∈ ?E
    unfolding mem-def Collect-def
  proof(rule span-induct-alt′)
    show ?h 0
      apply (rule exI[where x={}]) by simp
  next
    fix c x y
    assume x: x ∈ P and hy: ?h y
    from hy obtain S u where fS: finite S and SP: S⊆P
      and u: setsum (λv. u v ∗s v) S = y by blast
    let ?S = insert x S
    let ?u = λy. if y = x then (if x ∈ S then u y + c else c)
                 else u y
    from fS SP x have th0: finite (insert x S) insert x S ⊆ P by blast+
    {assume xS: x ∈ S
      have S1: S = (S − {x}) ∪ {x}
        and Sss:finite (S − {x}) finite {x} (S −{x}) ∩ {x} = {} using xS fS by
auto
      have setsum (λv. ?u v ∗s v) ?S =(∑ v∈S − {x}. u v ∗s v) + (u x + c) ∗s x
        using xS
        by (simp add: setsum-Un-disjoint[OF Sss, unfolded S1[symmetric]]
          setsum-clauses(2)[OF fS] cong del: if-weak-cong)
      also have ... = (∑ v∈S. u v ∗s v) + c ∗s x
        apply (simp add: setsum-Un-disjoint[OF Sss, unfolded S1[symmetric]])
        by (vector ring-simps)
      also have ... = c∗s x + y
        by (simp add: add-commute u)
      finally have setsum (λv. ?u v ∗s v) ?S = c∗s x + y .
    then have ?Q ?S ?u (c∗s x + y) using th0 by blast}
  moreover
  {assume xS: x ∉ S
    have th00: (∑ v∈S. (if v = x then c else u v) ∗s v) = y
      unfolding u[symmetric]
      apply (rule setsum-cong2)
      using xS by auto
    have ?Q ?S ?u (c∗s x + y) using fS xS th0
      by (simp add: th00 setsum-clauses add-commute cong del: if-weak-cong)}
```

```
  ultimately have ?Q ?S ?u (c*s x + y)
    by (cases x ∈ S, simp, simp)
    then show ?h (c*s x + y)
      apply −
      apply (rule exI[where x=?S])
      apply (rule exI[where x=?u]) by metis
  qed
  ultimately show ?thesis by blast
qed

lemma dependent-explicit:
  dependent P ⟷ (∃S u. finite S ∧ S ⊆ P ∧ (∃(v::'a::{idom,field} ^'n) ∈S. u v
≠ 0 ∧ setsum (λv. u v *s v) S = 0)) (is ?lhs = ?rhs)
proof−
  {assume dP: dependent P
    then obtain a S u where aP: a ∈ P and fS: finite S
      and SP: S ⊆ P − {a} and ua: setsum (λv. u v *s v) S = a
      unfolding dependent-def span-explicit by blast
    let ?S = insert a S
    let ?u = λy. if y = a then − 1 else u y
    let ?v = a
    from aP SP have aS: a ∉ S by blast
    from fS SP aP have th0: finite ?S ?S ⊆ P ?v ∈ ?S ?u ?v ≠ 0 by auto
    have s0: setsum (λv. ?u v *s v) ?S = 0
      using fS aS
     apply (simp add: vector-smult-lneg vector-smult-lid setsum-clauses ring-simps
)
      apply (subst (2) ua[symmetric])
      apply (rule setsum-cong2)
      by auto
    with th0 have ?rhs
      apply −
      apply (rule exI[where x= ?S])
      apply (rule exI[where x= ?u])
      by clarsimp}
  moreover
  {fix S u v assume fS: finite S
      and SP: S ⊆ P and vS: v ∈ S and uv: u v ≠ 0
    and u: setsum (λv. u v *s v) S = 0
    let ?a = v
    let ?S = S − {v}
    let ?u = λi. (− u i) / u v
    have th0: ?a ∈ P finite ?S ?S ⊆ P       using fS SP vS by auto
     have setsum (λv. ?u v *s v) ?S = setsum (λv. (− (inverse (u ?a))) *s (u v
*s v)) S − ?u v *s v
      using fS vS uv
      by (simp add: setsum-diff1 vector-smult-lneg divide-inverse
        vector-smult-assoc field-simps)
    also have ... = ?a
```

```
      unfolding setsum-cmul u
      using uv by (simp add: vector-smult-lneg)
    finally  have setsum (λv. ?u v *s v) ?S = ?a .
    with th0 have ?lhs
      unfolding dependent-def span-explicit
      apply −
      apply (rule bexI[where x= ?a])
      apply simp-all
      apply (rule exI[where x= ?S])
      by auto}
  ultimately show ?thesis by blast
qed


lemma span-finite:
  assumes fS: finite S
  shows span S = {(y::'a::semiring-1^'n). ∃ u. setsum (λv. u v *s v) S = y}
  (is - = ?rhs)
proof−
  {fix y assume y: y ∈ span S
    from y obtain S' u where fS': finite S' and SS': S' ⊆ S and
      u: setsum (λv. u v *s v) S' = y unfolding span-explicit by blast
    let ?u = λx. if x ∈ S' then u x else 0
    from setsum-restrict-set[OF fS, of λv. u v *s v S', symmetric] SS'
    have setsum (λv. ?u v *s v) S = setsum (λv. u v *s v) S'
      unfolding cond-value-iff cond-application-beta
      apply (simp add: cond-value-iff cong del: if-weak-cong)
      apply (rule setsum-cong)
      apply auto
      done
    hence setsum (λv. ?u v *s v) S = y by (metis u)
    hence y ∈ ?rhs by auto}
  moreover
  {fix y u assume u: setsum (λv. u v *s v) S = y
    then have y ∈ span S using fS unfolding span-explicit by auto}
  ultimately show ?thesis by blast
qed


lemma span-stdbasis:span {basis i :: 'a::ring-1^'n::finite | i. i ∈ (UNIV :: 'n set)}
= UNIV
apply (rule set-ext)
apply auto
apply (subst basis-expansion[symmetric])
apply (rule span-setsum)
apply simp
apply auto
```

apply (*rule span-mul*)
apply (*rule span-superset*)
apply (*auto simp add*: *Collect-def mem-def*)
done

lemma *has-size-stdbasis*: {*basis i* ::*real* ^ *'n*::*finite* | *i*. *i* ∈ (*UNIV* :: *'n set*)} *hassize* *CARD*(*'n*) (**is** *?S hassize ?n*)
proof−
have *eq*: *?S* = *basis* ' *UNIV* **by** *blast*
show *?thesis* **unfolding** *eq*
apply (*rule hassize-image-inj*[*OF basis-inj*])
by (*simp add*: *hassize-def*)
qed

lemma *finite-stdbasis*: *finite* {*basis i* ::*real*^*'n*::*finite* |*i*. *i*∈ (*UNIV*:: *'n set*)}
using *has-size-stdbasis*[*unfolded hassize-def*]
..

lemma *card-stdbasis*: *card* {*basis i* ::*real*^*'n*::*finite* |*i*. *i*∈ (*UNIV* :: *'n set*)} = *CARD*(*'n*)
using *has-size-stdbasis*[*unfolded hassize-def*]
..

lemma *independent-stdbasis-lemma*:
assumes *x*: (*x*::*'a*::*semiring-1* ^ *'n*) ∈ *span* (*basis* ' *S*)
and *iS*: *i* ∉ *S*
shows (*x*$*i*) = *0*
proof−
let *?U* = *UNIV* :: *'n set*
let *?B* = *basis* ' *S*
let *?P* = λ(*x*::*'a*^*'n*). ∀ *i*∈ *?U*. *i* ∉ *S* ⟶ *x*$*i* =*0*
{fix *x*::*'a*^*'n* **assume** *xS*: *x*∈ *?B*
from *xS* **have** *?P x* **by** *auto***}**
moreover
have *subspace ?P*
by (*auto simp add*: *subspace-def Collect-def mem-def*)
ultimately show *?thesis*
using *x span-induct*[*of ?B ?P x*] *iS* **by** *blast*
qed

lemma *independent-stdbasis*: *independent* {*basis i* ::*real*^*'n*::*finite* |*i*. *i*∈ (*UNIV* :: *'n set*)}
proof−
let *?I* = *UNIV* :: *'n set*
let *?b* = *basis* :: - ⇒ *real* ^*'n*
let *?B* = *?b* ' *?I*
have *eq*: {*?b i*|*i*. *i* ∈ *?I*} = *?B*
by *auto*
{assume *d*: *dependent ?B*

```
    then obtain k where k: k ∈ ?I ?b k ∈ span (?B − {?b k})
      unfolding dependent-def by auto
    have eq1: ?B − {?b k} = ?B − ?b ‘ {k}  by simp
    have eq2: ?B − {?b k} = ?b ‘ (?I − {k})
      unfolding eq1
      apply (rule inj-on-image-set-diff[symmetric])
      apply (rule basis-inj) using k(1) by auto
    from k(2) have th0: ?b k ∈ span (?b ‘ (?I − {k})) unfolding eq2 .
    from independent-stdbasis-lemma[OF th0, of k, simplified]
    have False by simp}
  then show ?thesis unfolding eq dependent-def ..
qed



lemma independent-insert:
  independent(insert (a::'a::field ^'n) S) ⟷
      (if a ∈ S then independent S
              else independent S ∧ a ∉ span S) (is ?lhs ⟷ ?rhs)
proof−
  {assume aS: a ∈ S
    hence ?thesis using insert-absorb[OF aS] by simp}
  moreover
  {assume aS: a ∉ S
    {assume i: ?lhs
      then have ?rhs using aS
        apply simp
        apply (rule conjI)
        apply (rule independent-mono)
        apply assumption
        apply blast
        by (simp add: dependent-def)}
    moreover
    {assume i: ?rhs
      have ?lhs using i aS
        apply simp
        apply (auto simp add: dependent-def)
        apply (case-tac aa = a, auto)
        apply (subgoal-tac insert a S − {aa} = insert a (S − {aa}))
        apply simp
        apply (subgoal-tac a ∈ span (insert aa (S − {aa})))
        apply (subgoal-tac insert aa (S − {aa}) = S)
        apply simp
        apply blast
        apply (rule in-span-insert)
        apply assumption
        apply blast
        apply blast
        done}
```

ultimately have *?thesis* **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

lemma *mem-delete*: $x \in (A - \{a\}) \longleftrightarrow x \neq a \wedge x \in A$
by *blast*

lemma *span-span*: *span* (*span A*) = *span A*
unfolding *span-def hull-hull* **..**

lemma *span-inc*: $S \subseteq$ *span S*
by (*metis subset-eq span-superset*)

lemma *spanning-subset-independent*:
assumes *BA*: $B \subseteq A$ **and** *iA*: *independent* (*A*::('*a*::*field* ^'*n*) *set*)
and *AsB*: $A \subseteq$ *span B*
shows $A = B$
proof
from *BA* **show** $B \subseteq A$ **.**
next
from *span-mono*[*OF BA*] *span-mono*[*OF AsB*]
have *sAB*: *span A* = *span B* **unfolding** *span-span* **by** *blast*

{**fix** *x* **assume** *x*: $x \in A$
from *iA* **have** *th0*: $x \notin$ *span* $(A - \{x\})$
unfolding *dependent-def* **using** *x* **by** *blast*
from *x* **have** *xsA*: $x \in$ *span A* **by** (*blast intro*: *span-superset*)
have $A - \{x\} \subseteq A$ **by** *blast*
hence *th1*:*span* $(A - \{x\}) \subseteq$ *span A* **by** (*metis span-mono*)
{**assume** *xB*: $x \notin B$
from *xB BA* **have** $B \subseteq A - \{x\}$ **by** *blast*
hence *span B* $\subseteq$ *span* $(A - \{x\})$ **by** (*metis span-mono*)
with *th1 th0 sAB* **have** $x \notin$ *span A* **by** *blast*
with *x* **have** *False* **by** (*metis span-superset*)}
then have $x \in B$ **by** *blast*}
then show $A \subseteq B$ **by** *blast*
qed

lemma *exchange-lemma*:
assumes *f*:*finite* (*t*:: ('*a*::*field*^'*n*) *set*) **and** *i*: *independent s*
and *sp*:*s* $\subseteq$ *span t*
shows $\exists t'.$ (*t'* *hassize card t*) $\wedge$ *s* $\subseteq$ *t'* $\wedge$ *t'* $\subseteq$ *s* $\cup$ *t* $\wedge$ *s* $\subseteq$ *span t'*
using *f i sp*
proof(*induct c*≡*card*(*t* − *s*) *arbitrary*: *s t rule*: *nat-less-induct*)
fix *n*:: *nat* **and** *s t* :: ('*a* ^'*n*) *set*

assume *H*: $\forall m{<}n.\ \forall (x{::}\ ('a\ \hat{}\ 'n)\ set)\ xa.$
finite xa $\longrightarrow$
independent x $\longrightarrow$
$x \subseteq$ *span xa* $\longrightarrow$
$m =$ *card* $(xa - x) \longrightarrow$
$(\exists t'.\ (t'$ *hassize card xa*$) \wedge$
$x \subseteq t' \wedge t' \subseteq x \cup xa \wedge x \subseteq$ *span* $t')$
and *ft*: *finite t* **and** *s*: *independent s* **and** *sp*: $s \subseteq$ *span t*
and *n*: $n =$ *card* $(t - s)$
let *?P* $= \lambda t'.\ (t'$ *hassize card t*$) \wedge s \subseteq t' \wedge t' \subseteq s \cup t \wedge s \subseteq$ *span* t'
let *?ths* $= \exists t'.$ *?P* t'
{assume *st*: $s \subseteq t$
from *st ft span-mono*[*OF st*] **have** *?ths* **apply** − **apply** (*rule exI*[**where** $x{=}t$])
by (*auto simp add*: *hassize-def intro*: *span-superset*)**}**
moreover
{assume *st*: $t \subseteq s$

from *spanning-subset-independent*[*OF st s sp*]
st ft span-mono[*OF st*] **have** *?ths* **apply** − **apply** (*rule exI*[**where** $x{=}t$])
by (*auto simp add*: *hassize-def intro*: *span-superset*)**}**
moreover
{assume *st*: $\neg\ s \subseteq t \neg\ t \subseteq s$
from *st*(*2*) **obtain** *b* **where** *b*: $b \in t\ b \notin s$ **by** *blast*
from *b* **have** $t - \{b\} - s \subset t - s$ **by** *blast*
then have *cardlt*: *card* $(t - \{b\} - s) < n$ **using** *n ft*
by (*auto intro*: *psubset-card-mono*)
from *b ft* **have** *ct0*: *card* $t \neq 0$ **by** *auto*
{assume *stb*: $s \subseteq$ *span*$(t - \{b\})$
from *ft* **have** *ftb*: *finite* $(t - \{b\})$ **by** *auto*
from *H*[*rule-format*, *OF cardlt ftb s stb*]
obtain *u* **where** *u*: *u hassize card* $(t - \{b\})\ s \subseteq u\ u \subseteq s \cup (t - \{b\})\ s \subseteq$ *span u* **by** *blast*
let *?w* $=$ *insert b u*
have *th0*: $s \subseteq$ *insert b u* **using** *u* **by** *blast*
from *u*(*3*) *b* **have** $u \subseteq s \cup t$ **by** *blast*
then have *th1*: *insert b u* $\subseteq s \cup t$ **using** *u b* **by** *blast*
have *bu*: $b \notin u$ **using** *b u* **by** *blast*
from *u*(*1*) **have** *fu*: *finite u* **by** (*simp add*: *hassize-def*)
from *u*(*1*) *ft b* **have** *u hassize* (*card t* − *1*) **by** *auto*
then
have *th2*: *insert b u hassize card t*
using *card-insert-disjoint*[*OF fu bu*] *ct0* **by** (*auto simp add*: *hassize-def*)
from *u*(*4*) **have** $s \subseteq$ *span u* **.**
also have $\ldots \subseteq$ *span* (*insert b u*) **apply** (*rule span-mono*) **by** *blast*
finally have *th3*: $s \subseteq$ *span* (*insert b u*) **.** **from** *th0 th1 th2 th3* **have** *th*: *?P ?w* **by** *blast*
from *th* **have** *?ths* **by** *blast***}**
moreover
{assume *stb*: $\neg\ s \subseteq$ *span*$(t - \{b\})$

```
    from stb obtain a where a: a ∈ s a ∉ span (t − {b}) by blast
    have ab: a ≠ b using a b by blast
    have at: a ∉ t using a ab span-superset[of a t− {b}] by auto
    have mlt: card ((insert a (t − {b})) − s) < n
      using cardlt ft n  a b by auto
    have ft': finite (insert a (t − {b})) using ft by auto
    {fix x assume xs: x ∈ s
      have t: t ⊆ (insert b (insert a (t −{b}))) using b by auto
      from b(1) have b ∈ span t by (simp add: span-superset)
      have bs: b ∈ span (insert a (t − {b}))
        by (metis in-span-delete a sp mem-def subset-eq)
      from xs sp have x ∈ span t by blast
      with span-mono[OF t]
      have x: x ∈ span (insert b (insert a (t − {b}))) ..
      from span-trans[OF bs x] have x ∈ span (insert a (t − {b}))  .}
    then have sp': s ⊆ span (insert a (t − {b})) by blast

    from H[rule-format, OF mlt ft' s sp' refl] obtain u where
      u: u hassize card (insert a (t −{b})) s ⊆ u u ⊆ s ∪ insert a (t −{b})
      s ⊆ span u by blast
    from u a b ft at ct0 have ?P u by (auto simp add: hassize-def)
    then have ?ths by blast }
  ultimately have ?ths by blast
 }
 ultimately
 show ?ths  by blast
qed
```

```
lemma independent-span-bound:
  assumes f: finite t and i: independent (s::('a::field^'n) set) and sp:s ⊆ span t
  shows finite s ∧ card s ≤ card t
  by (metis exchange-lemma[OF f i sp] hassize-def finite-subset card-mono)

lemma finite-Atleast-Atmost[simp]: finite {f x |x. x∈ {(i::'a::finite-intvl-succ) ..
j}}
proof−
  have eq: {f x |x. x∈ {i .. j}} = f ‘ {i .. j} by auto
  show ?thesis unfolding eq
    apply (rule finite-imageI)
    apply (rule finite-intvl)
    done
qed

lemma finite-Atleast-Atmost-nat[simp]: finite {f x |x. x∈ (UNIV::'a::finite set)}
proof−
  have eq: {f x |x. x∈ UNIV} = f ‘ UNIV by auto
  show ?thesis unfolding eq
```

```
    apply (rule finite-imageI)
    apply (rule finite)
    done
qed


lemma independent-bound:
  fixes S:: (real^'n::finite) set
  shows independent S ⟹ finite S ∧ card S <= CARD('n)
  apply (subst card-stdbasis[symmetric])
  apply (rule independent-span-bound)
  apply (rule finite-Atleast-Atmost-nat)
  apply assumption
  unfolding span-stdbasis
  apply (rule subset-UNIV)
  done

lemma dependent-biggerset: (finite (S::(real ^'n::finite) set) ==> card S > CARD('n))
==> dependent S
  by (metis independent-bound not-less)



lemma maximal-independent-subset-extend:
  assumes sv: (S::(real^'n::finite) set) ⊆ V and iS: independent S
  shows ∃B. S ⊆ B ∧ B ⊆ V ∧ independent B ∧ V ⊆ span B
  using sv iS
proof(induct d≡ CARD('n) − card S arbitrary: S rule: nat-less-induct)
  fix n and S:: (real^'n) set
  assume H: ∀m<n. ∀S ⊆ V. independent S ⟶ m = CARD('n) − card S ⟶
            (∃B. S ⊆ B ∧ B ⊆ V ∧ independent B ∧ V ⊆ span B)
    and sv: S ⊆ V and i: independent S and n: n = CARD('n) − card S
  let ?P = λB. S ⊆ B ∧ B ⊆ V ∧ independent B ∧ V ⊆ span B
  let ?ths = ∃x. ?P x
  let ?d = CARD('n)
  {assume V ⊆ span S
    then have ?ths  using sv i by blast }
  moreover
  {assume VS: ¬ V ⊆ span S
    from VS obtain a where a: a ∈ V a ∉ span S by blast
    from a have aS: a ∉ S by (auto simp add: span-superset)
    have th0: insert a S ⊆ V using a sv by blast
    from independent-insert[of a S]  i a
    have th1: independent (insert a S) by auto
    have mlt: ?d − card (insert a S) < n
      using aS a n independent-bound[OF th1]
      by auto

    from H[rule-format, OF mlt th0 th1 refl]
```

obtain B **where** B: *insert a* $S \subseteq B$ $B \subseteq V$ *independent* B $V \subseteq$ *span* B
by *blast*
from B **have** *?P* B **by** *auto*
then have *?ths* **by** *blast*}
ultimately show *?ths* **by** *blast*
qed

lemma *maximal-independent-subset*:
$\exists (B::$ (*real* ^'*n*::*finite*) *set*). $B \subseteq V \wedge$ *independent* $B \wedge V \subseteq$ *span* B
by (*metis maximal-independent-subset-extend*[*of* {}:: (*real* ^'*n*) *set*] *empty-subsetI independent-empty*)

definition *dim* $V = (SOME\ n.\ \exists B.\ B \subseteq V \wedge$ *independent* $B \wedge V \subseteq$ *span* $B \wedge$ (B *hassize* n))

lemma *basis-exists*: $\exists B.\ (B ::$ (*real* ^'*n*::*finite*) *set*) $\subseteq V \wedge$ *independent* $B \wedge V \subseteq$ *span* $B \wedge$ (B *hassize dim* V)
unfolding *dim-def some-eq-ex*[*of* $\lambda n.\ \exists B.\ B \subseteq V \wedge$ *independent* $B \wedge V \subseteq$ *span* $B \wedge$ (B *hassize* n)]
unfolding *hassize-def*
using *maximal-independent-subset*[*of* V] *independent-bound*
by *auto*

lemma *independent-card-le-dim*: (B::(*real* ^'*n*::*finite*) *set*) $\subseteq V \Longrightarrow$ *independent* $B \Longrightarrow$ *finite* $B \wedge$ *card* $B \leq$ *dim* V
by (*metis basis-exists*[*of* V] *independent-span-bound*[**where** *?'a=real*] *hassize-def subset-trans*)

lemma *span-card-ge-dim*: (B::(*real* ^'*n*::*finite*) *set*) $\subseteq V \Longrightarrow V \subseteq$ *span* $B \Longrightarrow$ *finite* $B \Longrightarrow$ *dim* $V \leq$ *card* B
by (*metis basis-exists*[*of* V] *independent-span-bound hassize-def subset-trans*)

lemma *basis-card-eq-dim*:
$B \subseteq (V::$ (*real* ^'*n*::*finite*) *set*) $\Longrightarrow V \subseteq$ *span* $B \Longrightarrow$ *independent* $B \Longrightarrow$ *finite* $B \wedge$ *card* $B =$ *dim* V
by (*metis order-eq-iff independent-card-le-dim span-card-ge-dim independent-mono*)

lemma *dim-unique*: (B::(*real* ^'*n*::*finite*) *set*) $\subseteq V \Longrightarrow V \subseteq$ *span* $B \Longrightarrow$ *independent* $B \Longrightarrow B$ *hassize* $n \Longrightarrow$ *dim* $V = n$
by (*metis basis-card-eq-dim hassize-def*)

lemma *dim-univ*: *dim* (*UNIV* :: (*real*^'*n*::*finite*) *set*) = *CARD*('*n*)
apply (*rule dim-unique*[*of* {*basis i* |*i*. *i*$\in$ (*UNIV* :: '*n set*)}])

by (*auto simp only*: *span-stdbasis has-size-stdbasis independent-stdbasis*)

lemma *dim-subset*:
(*S*:: (*real* ^'*n*::*finite*) *set*) ⊆ *T* ⟹ *dim S* ≤ *dim T*
using *basis-exists*[*of T*] *basis-exists*[*of S*]
by (*metis independent-span-bound*[**where** *?'a* = *real* **and** *?'n* = *'n*] *subset-eq hassize-def*)

lemma *dim-subset-univ*: *dim* (*S*:: (*real*^'*n*::*finite*) *set*) ≤ *CARD*('*n*)
by (*metis dim-subset subset-UNIV dim-univ*)

lemma *card-ge-dim-independent*:
assumes *BV*:(*B*::(*real* ^'*n*::*finite*) *set*) ⊆ *V* **and** *iB*:*independent B* **and** *dVB*:*dim V* ≤ *card B*
shows *V* ⊆ *span B*
proof−
{fix *a* **assume** *aV*: *a* ∈ *V*
{assume *aB*: *a* ∉ *span B*
then have *iaB*: *independent* (*insert a B*) **using** *iB aV BV* **by** (*simp add*: *independent-insert*)
from *aV BV* **have** *th0*: *insert a B* ⊆ *V* **by** *blast*
from *aB* **have** *a* ∉*B* **by** (*auto simp add*: *span-superset*)
with *independent-card-le-dim*[*OF th0 iaB*] *dVB* **have** *False* **by** *auto*}
then have *a* ∈ *span B* **by** *blast*}
then show *?thesis* **by** *blast*
qed

lemma *card-le-dim-spanning*:
assumes *BV*: (*B*:: (*real* ^'*n*::*finite*) *set*) ⊆ *V* **and** *VB*: *V* ⊆ *span B*
and *fB*: *finite B* **and** *dVB*: *dim V* ≥ *card B*
shows *independent B*
proof−
{fix *a* **assume** *a*: *a* ∈ *B a* ∈ *span* (*B* −{*a*})
from *a fB* **have** *c0*: *card B* ≠ *0* **by** *auto*
from *a fB* **have** *cb*: *card* (*B* −{*a*}) = *card B* − *1* **by** *auto*
from *BV a* **have** *th0*: *B* −{*a*} ⊆ *V* **by** *blast*
{fix *x* **assume** *x*: *x* ∈ *V*
from *a* **have** *eq*: *insert a* (*B* −{*a*}) = *B* **by** *blast*
from *x VB* **have** *x'*: *x* ∈ *span B* **by** *blast*
from *span-trans*[*OF a*(*2*), *unfolded eq*, *OF x'*]
have *x* ∈ *span* (*B* −{*a*}) **. }**
then have *th1*: *V* ⊆ *span* (*B* −{*a*}) **by** *blast*
have *th2*: *finite* (*B* −{*a*}) **using** *fB* **by** *auto*
from *span-card-ge-dim*[*OF th0 th1 th2*]
have *c*: *dim V* ≤ *card* (*B* −{*a*}) **.**
from *c c0 dVB cb* **have** *False* **by** *simp*}
then show *?thesis* **unfolding** *dependent-def* **by** *blast*

qed

lemma *card-eq-dim*: (*B*:: (*real* ^'*n*::*finite*) *set*) ⊆ *V* ⟹ *B hassize dim V* ⟹ *independent B* ⟷ *V* ⊆ *span B*
by (*metis hassize-def order-eq-iff card-le-dim-spanning card-ge-dim-independent*)

lemma *independent-bound-general*:
independent (*S*:: (*real*^'*n*::*finite*) *set*) ⟹ *finite S* ∧ *card S* ≤ *dim S*
by (*metis independent-card-le-dim independent-bound subset-refl*)

lemma *dependent-biggerset-general*: (*finite* (*S*:: (*real*^'*n*::*finite*) *set*) ⟹ *card S* > *dim S*) ⟹ *dependent S*
using *independent-bound-general*[*of S*] **by** (*metis linorder-not-le*)

lemma *dim-span*: *dim* (*span* (*S*:: (*real* ^'*n*::*finite*) *set*)) = *dim S*
proof−
have *th0*: *dim S* ≤ *dim* (*span S*)
by (*auto simp add*: *subset-eq intro*: *dim-subset span-superset*)
from *basis-exists*[*of S*]
obtain *B* **where** *B*: *B* ⊆ *S independent B S* ⊆ *span B B hassize dim S* **by** *blast*
from *B* **have** *fB*: *finite B card B* = *dim S* **unfolding** *hassize-def* **by** *blast*+
have *bSS*: *B* ⊆ *span S* **using** *B*(*1*) **by** (*metis subset-eq span-inc*)
have *sssB*: *span S* ⊆ *span B* **using** *span-mono*[*OF B*(*3*)] **by** (*simp add*: *span-span*)
from *span-card-ge-dim*[*OF bSS sssB fB*(*1*)] *th0* **show** *?thesis*
using *fB*(*2*) **by** *arith*
qed

lemma *subset-le-dim*: (*S*:: (*real* ^'*n*::*finite*) *set*) ⊆ *span T* ⟹ *dim S* ≤ *dim T*
by (*metis dim-span dim-subset*)

lemma *span-eq-dim*: *span* (*S*:: (*real* ^'*n*::*finite*) *set*) = *span T* ==> *dim S* = *dim T*
by (*metis dim-span*)

lemma *spans-image*:
assumes *lf*: *linear* (*f*::'*a*::*semiring-1*^'*n* ⇒ -) **and** *VB*: *V* ⊆ *span B*
shows *f* ‘ *V* ⊆ *span* (*f* ‘ *B*)
unfolding *span-linear-image*[*OF lf*]
by (*metis VB image-mono*)

lemma *dim-image-le*:
fixes *f* :: *real*^'*n*::*finite* ⇒ *real*^'*m*::*finite*
assumes *lf*: *linear f* **shows** *dim* (*f* ‘ *S*) ≤ *dim* (*S*:: (*real* ^'*n*::*finite*) *set*)
proof−

from *basis-exists*[*of S*] **obtain** *B* **where**
B: *B* ⊆ *S independent B S* ⊆ *span B B hassize dim S* **by** *blast*
from *B* **have** *fB*: *finite B card B* = *dim S* **unfolding** *hassize-def* **by** *blast*+
have *dim* (*f* ' *S*) ≤ *card* (*f* ' *B*)
apply (*rule span-card-ge-dim*)
using *lf B fB* **by** (*auto simp add*: *span-linear-image spans-image subset-image-iff*)
also have ... ≤ *dim S* **using** *card-image-le*[*OF fB*(*1*)] *fB* **by** *simp*
finally show *?thesis* **.**
qed

lemma *spanning-surjective-image*:
assumes *us*: *UNIV* ⊆ *span* (*S*:: ('*a*::*semiring-1* ^'*n*) *set*)
and *lf*: *linear f* **and** *sf*: *surj f*
shows *UNIV* ⊆ *span* (*f* ' *S*)
proof−
have *UNIV* ⊆ *f* ' *UNIV* **using** *sf* **by** (*auto simp add*: *surj-def*)
also have ... ⊆ *span* (*f* ' *S*) **using** *spans-image*[*OF lf us*] **.**
finally show *?thesis* **.**
qed

lemma *independent-injective-image*:
assumes *iS*: *independent* (*S*::('*a*::*semiring-1*^'*n*) *set*) **and** *lf*: *linear f* **and** *fi*: *inj f*
shows *independent* (*f* ' *S*)
proof−
{fix *a* **assume** *a*: *a* ∈ *S f a* ∈ *span* (*f* ' *S* − {*f a*})
have *eq*: *f* ' *S* − {*f a*} = *f* ' (*S* − {*a*}) **using** *fi*
by (*auto simp add*: *inj-on-def*)
from *a* **have** *f a* ∈ *f* ' *span* (*S* −{*a*})
unfolding *eq span-linear-image*[*OF lf*, *of S* − {*a*}] **by** *blast*
hence *a* ∈ *span* (*S* −{*a*}) **using** *fi* **by** (*auto simp add*: *inj-on-def*)
with *a*(*1*) *iS* **have** *False* **by** (*simp add*: *dependent-def*) **}**
then show *?thesis* **unfolding** *dependent-def* **by** *blast*
qed

definition *pairwise R S* ⟷ (∀ *x* ∈ *S*. ∀ *y*∈ *S*. *x*≠*y* ⟶ *R x y*)

lemma *vector-sub-project-orthogonal*: (*b*::'*a*::*ordered-field*^'*n*::*finite*) · (*x* − ((*b* · *x*) / (*b*·*b*)) ∗*s b*) = *0*
apply (*cases b* = *0*, *simp*)
apply (*simp add*: *dot-rsub dot-rmult*)
unfolding *times-divide-eq-right*[*symmetric*]
by (*simp add*: *field-simps dot-eq-0*)

lemma *basis-orthogonal*:
fixes B :: (*real* ^'*n*::*finite*) *set*
assumes *fB*: *finite B*
shows $\exists\, C.$ *finite* $C \wedge$ *card* $C \leq$ *card* $B \wedge$ *span* $C =$ *span* $B \wedge$ *pairwise orthogonal* C
(**is** $\exists\, C.$ *?P B C*)
proof(*induct rule*: *finite-induct*[*OF fB*])
case *1* **thus** *?case* **apply** (*rule exI*[**where** x={}]) **by** (*auto simp add*: *pairwise-def*)
next
case (*2 a B*)
note *fB* = ‹*finite B*› **and** *aB* = ‹$a \notin B$›
from ‹$\exists\, C.$ *finite* $C \wedge$ *card* $C \leq$ *card* $B \wedge$ *span* $C =$ *span* $B \wedge$ *pairwise orthogonal* C›
obtain C **where** C: *finite* C *card* $C \leq$ *card* B
span $C =$ *span* B *pairwise orthogonal* C **by** *blast*
let *?a* = $a -$ *setsum* $(\lambda x.\ (x \cdot a\ /\ (x \cdot x)) *s\ x)$ C
let *?C* = *insert ?a C*
from $C(1)$ **have** *fC*: *finite ?C* **by** *simp*
from *fB aB* $C(1,2)$ **have** *cC*: *card ?C* $\leq$ *card* (*insert a B*) **by** (*simp add*: *card-insert-if*)
{fix x k
have *th0*: $\bigwedge$(a::'b::*comm-ring*) b c. $a - (b - c) = c + (a - b)$ **by** (*simp add*: *ring-simps*)
have $x - k *s\ (a - (\sum x \in C.\ (x \cdot a\ /\ (x \cdot x)) *s\ x)) \in$ *span* $C \longleftrightarrow x - k *s\ a \in$ *span* C
apply (*simp only*: *vector-ssub-ldistrib th0*)
apply (*rule span-add-eq*)
apply (*rule span-mul*)
apply (*rule span-setsum*[*OF* $C(1)$])
apply *clarify*
apply (*rule span-mul*)
by (*rule span-superset*)**}**
then have *SC*: *span ?C* = *span* (*insert a B*)
unfolding *expand-set-eq span-breakdown-eq* $C(3)$[*symmetric*] **by** *auto*
thm *pairwise-def*
{fix x y **assume** *xC*: $x \in$ *?C* **and** *yC*: $y \in$ *?C* **and** *xy*: $x \neq y$
{assume *xa*: x = *?a* **and** *ya*: y = *?a*
have *orthogonal x y* **using** *xa ya xy* **by** *blast***}**
moreover
{assume *xa*: x = *?a* **and** *ya*: $y \neq$ *?a* $y \in C$
from *ya* **have** *Cy*: C = *insert* y $(C - \{y\})$ **by** *blast*
have *fth*: *finite* $(C - \{y\})$ **using** C **by** *simp*
have *orthogonal x y*
using *xa ya*
unfolding *orthogonal-def xa dot-lsub dot-rsub diff-eq-0-iff-eq*
apply *simp*
apply (*subst Cy*)
using $C(1)$ *fth*

```
      apply (simp only: setsum-clauses)
      thm dot-ladd
        apply (auto simp add: dot-ladd dot-radd dot-lmult dot-rmult dot-eq-0
dot-sym[of y a] dot-lsum[OF fth])
      apply (rule setsum-0')
      apply clarsimp
      apply (rule C(4)[unfolded pairwise-def orthogonal-def, rule-format])
      by auto}
   moreover
   {assume xa: x ≠ ?a x ∈ C and ya: y = ?a
    from xa have Cx: C = insert x (C - {x}) by blast
    have fth: finite (C - {x}) using C by simp
    have orthogonal x y
      using xa ya
      unfolding orthogonal-def ya dot-rsub dot-lsub diff-eq-0-iff-eq
      apply simp
      apply (subst Cx)
      using C(1) fth
      apply (simp only: setsum-clauses)
      apply (subst dot-sym[of x])
     apply (auto simp add: dot-radd dot-rmult dot-eq-0 dot-sym[of x a] dot-rsum[OF
fth])
      apply (rule setsum-0')
      apply clarsimp
      apply (rule C(4)[unfolded pairwise-def orthogonal-def, rule-format])
      by auto}
   moreover
   {assume xa: x ∈ C and ya: y ∈ C
    have orthogonal x y using xa ya xy C(4) unfolding pairwise-def by blast}
   ultimately have orthogonal x y using xC yC by blast}
  then have CPO: pairwise orthogonal ?C unfolding pairwise-def by blast
  from fC cC SC CPO have ?P (insert a B) ?C by blast
  then show ?case by blast
qed

lemma orthogonal-basis-exists:
  fixes V :: (real ^'n::finite) set
  shows ∃B. independent B ∧ B ⊆ span V ∧ V ⊆ span B ∧ (B hassize dim V)
∧ pairwise orthogonal B
proof-
  from basis-exists[of V] obtain B where B: B ⊆ V independent B V ⊆ span B
B hassize dim V by blast
  from B have fB: finite B card B = dim V by (simp-all add: hassize-def)
  from basis-orthogonal[OF fB(1)] obtain C where
    C: finite C card C ≤ card B span C = span B pairwise orthogonal C by blast
  from C B
  have CSV: C ⊆ span V by (metis span-inc span-mono subset-trans)
   from span-mono[OF B(3)]  C have SVC: span V ⊆ span C by (simp add:
span-span)
```

from *card-le-dim-spanning*[*OF CSV SVC C*(*1*)] *C*(*2*,*3*) *fB*
have *iC*: *independent C* **by** (*simp add*: *dim-span*)
from *C fB* **have** *card C* ≤ *dim V* **by** *simp*
moreover have *dim V* ≤ *card C* **using** *span-card-ge-dim*[*OF CSV SVC C*(*1*)]
by (*simp add*: *dim-span*)
ultimately have *CdV*: *C hassize dim V* **unfolding** *hassize-def* **using** *C*(*1*) **by** *simp*
from *C B CSV CdV iC* **show** *?thesis* **by** *auto*
qed

lemma *span-eq*: *span S* = *span T* ⟷ *S* ⊆ *span T* ∧ *T* ⊆ *span S*
by (*metis set-eq-subset span-mono span-span span-inc*)

lemma *span-not-univ-orthogonal*:
assumes *sU*: *span S* ≠ *UNIV*
shows ∃(*a*:: *real* ^*'n*::*finite*). *a* ≠*0* ∧ (∀ *x* ∈ *span S*. *a* ⋅ *x* = *0*)
proof−
from *sU* **obtain** *a* **where** *a*: *a* ∉ *span S* **by** *blast*
from *orthogonal-basis-exists* **obtain** *B* **where**
B: *independent B B* ⊆ *span S S* ⊆ *span B B hassize dim S pairwise orthogonal B*
by *blast*
from *B* **have** *fB*: *finite B card B* = *dim S* **by** (*simp-all add*: *hassize-def*)
from *span-mono*[*OF B*(*2*)] *span-mono*[*OF B*(*3*)]
have *sSB*: *span S* = *span B* **by** (*simp add*: *span-span*)
let *?a* = *a* − *setsum* (λ*b*. (*a*⋅*b* / (*b*⋅*b*)) ∗*s b*) *B*
have *setsum* (λ*b*. (*a*⋅*b* / (*b*⋅*b*)) ∗*s b*) *B* ∈ *span S*
unfolding *sSB*
apply (*rule span-setsum*[*OF fB*(*1*)])
apply *clarsimp*
apply (*rule span-mul*)
by (*rule span-superset*)
with *a* **have** *a0*:*?a* ≠ *0* **by** *auto*
have ∀ *x*∈*span B*. *?a* ⋅ *x* = *0*
proof(*rule span-induct'*)
show *subspace* (λ*x*. *?a* ⋅ *x* = *0*)
by (*auto simp add*: *subspace-def mem-def dot-radd dot-rmult*)
next
{fix *x* **assume** *x*: *x* ∈ *B*
from *x* **have** *B'*: *B* = *insert x* (*B* − {*x*}) **by** *blast*
have *fth*: *finite* (*B* − {*x*}) **using** *fB* **by** *simp*
have *?a* ⋅ *x* = *0*
apply (*subst B'*) **using** *fB fth*
unfolding *setsum-clauses*(*2*)[*OF fth*]
apply *simp*

```
      apply (clarsimp simp add: dot-lsub dot-ladd dot-lmult dot-lsum dot-eq-0)
      apply (rule setsum-0', rule ballI)
      unfolding dot-sym
      by (auto simp add: x field-simps dot-eq-0 intro: B(5)[unfolded pairwise-def
orthogonal-def, rule-format])}
   then show ∀ x ∈ B. ?a · x = 0 by blast
  qed
  with a0 show ?thesis unfolding sSB by (auto intro: exI[where x=?a])
qed

lemma span-not-univ-subset-hyperplane:
  assumes SU: span S ≠ (UNIV ::(real^'n::finite) set)
  shows ∃ a. a ≠0 ∧ span S ⊆ {x. a · x = 0}
  using span-not-univ-orthogonal[OF SU] by auto

lemma lowdim-subset-hyperplane:
  assumes d: dim S < CARD('n::finite)
  shows ∃(a::real ^'n::finite). a ≠ 0 ∧ span S ⊆ {x. a · x = 0}
proof−
  {assume span S = UNIV
    hence dim (span S) = dim (UNIV :: (real ^'n) set) by simp
    hence dim S = CARD('n) by (simp add: dim-span dim-univ)
    with d have False by arith}
  hence th: span S ≠ UNIV by blast
  from span-not-univ-subset-hyperplane[OF th] show ?thesis .
qed



lemma linear-indep-image-lemma:
  assumes lf: linear f and fB: finite B
  and ifB: independent (f ' B)
  and fi: inj-on f B and xsB: x ∈ span B
  and fx: f (x::'a::field^'n) = 0
  shows x = 0
  using fB ifB fi xsB fx
proof(induct arbitrary: x rule: finite-induct[OF fB])
  case 1 thus ?case by (auto simp add: span-empty)
next
  case (2 a b x)
  have fb: finite b using 2.prems by simp
  have th0: f ' b ⊆ f ' (insert a b)
    apply (rule image-mono) by blast
  from independent-mono[ OF 2.prems(2) th0]
  have ifb: independent (f ' b) .
  have fib: inj-on f b
    apply (rule subset-inj-on [OF 2.prems(3)])
    by blast
  from span-breakdown[of a insert a b, simplified, OF 2.prems(4)]
```

```
  obtain k where k: x − k∗s a ∈ span (b −{a}) by blast
  have f (x − k∗s a) ∈ span (f ‘ b)
    unfolding span-linear-image[OF lf]
    apply (rule imageI)
    using k span-mono[of b−{a} b] by blast
  hence f x − k∗s f a ∈ span (f ‘ b)
    by (simp add: linear-sub[OF lf] linear-cmul[OF lf])
  hence th: −k ∗s f a ∈ span (f ‘ b)
    using 2.prems(5) by (simp add: vector-smult-lneg)
  {assume k0: k = 0
    from k0 k have x ∈ span (b −{a}) by simp
    then have x ∈ span b using span-mono[of b−{a} b]
      by blast}
  moreover
  {assume k0: k ≠ 0
    from span-mul[OF th, of − 1/ k] k0
    have th1: f a ∈ span (f ‘ b)
      by (auto simp add: vector-smult-assoc)
    from inj-on-image-set-diff[OF 2.prems(3), of insert a b {a}, symmetric]
    have tha: f ‘ insert a b − f ‘ {a} = f ‘ (insert a b − {a}) by blast
    from 2.prems(2)[unfolded dependent-def bex-simps(10), rule-format, of f a]
    have f a ∉ span (f ‘ b) using tha
      using 2.hyps(2)
      2.prems(3) by auto
    with th1 have False by blast
    then have x ∈ span b by blast}
  ultimately have xsb: x ∈ span b by blast
  from 2.hyps(3)[OF fb ifb fib xsb 2.prems(5)]
  show x = 0 .
qed



lemma linear-independent-extend-lemma:
  assumes fi: finite B and ib: independent B
  shows ∃ g. (∀ x∈ span B. ∀ y∈ span B. g ((x::'a::field^'n) + y) = g x + g y)
          ∧ (∀ x∈ span B. ∀ c. g (c∗s x) = c ∗s g x)
          ∧ (∀ x∈ B. g x = f x)
using ib fi
proof(induct rule: finite-induct[OF fi])
  case 1 thus ?case by (auto simp add: span-empty)
next
  case (2 a b)
  from 2.prems 2.hyps have ibf: independent b finite b
    by (simp-all add: independent-insert)
  from 2.hyps(3)[OF ibf] obtain g where
    g: ∀ x∈span b. ∀ y∈span b. g (x + y) = g x + g y
    ∀ x∈span b. ∀ c. g (c ∗s x) = c ∗s g x ∀ x∈b. g x = f x by blast
  let ?h = λz. SOME k. (z − k ∗s a) ∈ span b
```

{fix z **assume** z: $z \in span\ (insert\ a\ b)$
have *th0*: $z - ?h\ z *s\ a \in span\ b$
apply (*rule someI-ex*)
unfolding *span-breakdown-eq*[*symmetric*]
using z **.**
{fix k **assume** k: $z - k *s\ a \in span\ b$
have *eq*: $z - ?h\ z *s\ a - (z - k *s\ a) = (k - ?h\ z) *s\ a$
by (*simp add*: *ring-simps vector-sadd-rdistrib*[*symmetric*])
from *span-sub*[*OF th0 k*]
have *khz*: $(k - ?h\ z) *s\ a \in span\ b$ **by** (*simp add*: *eq*)
{assume $k \neq ?h\ z$ **hence** *k0*: $k - ?h\ z \neq 0$ **by** *simp*
from *k0 span-mul*[*OF khz*, *of 1* $/(k - ?h\ z)$]
have $a \in span\ b$ **by** (*simp add*: *vector-smult-assoc*)
with *2.prems*(*1*) *2.hyps*(*2*) **have** *False*
by (*auto simp add*: *dependent-def*)**}**
then have $k = ?h\ z$ **by** *blast***}**
with *th0* **have** $z - ?h\ z *s\ a \in span\ b \wedge (\forall k.\ z - k *s\ a \in span\ b \longrightarrow k = ?h\ z)$ **by** *blast***}**
note $h = this$
let $?g = \lambda z.\ ?h\ z *s\ f\ a + g\ (z - ?h\ z *s\ a)$
{fix $x\ y$ **assume** x: $x \in span\ (insert\ a\ b)$ **and** y: $y \in span\ (insert\ a\ b)$
have *tha*: $\bigwedge(x::'a\hat{}'n)\ y\ a\ k\ l.\ (x + y) - (k + l) *s\ a = (x - k *s\ a) + (y - l *s\ a)$
by (*vector ring-simps*)
have *addh*: $?h\ (x + y) = ?h\ x + ?h\ y$
apply (*rule conjunct2*[*OF h*, *rule-format*, *symmetric*])
apply (*rule span-add*[*OF x y*])
unfolding *tha*
by (*metis span-add x y conjunct1*[*OF h*, *rule-format*])
have $?g\ (x + y) = ?g\ x + ?g\ y$
unfolding *addh tha*
g(*1*)[*rule-format*,*OF conjunct1*[*OF h*, *OF x*] *conjunct1*[*OF h*, *OF y*]]
by (*simp add*: *vector-sadd-rdistrib*)**}**
moreover
{fix $x::'a\hat{}'n$ **and** $c::'a$ **assume** x: $x \in span\ (insert\ a\ b)$
have *tha*: $\bigwedge(x::'a\hat{}'n)\ c\ k\ a.\ c *s\ x - (c * k) *s\ a = c *s\ (x - k *s\ a)$
by (*vector ring-simps*)
have *hc*: $?h\ (c *s\ x) = c * ?h\ x$
apply (*rule conjunct2*[*OF h*, *rule-format*, *symmetric*])
apply (*metis span-mul x*)
by (*metis tha span-mul x conjunct1*[*OF h*])
have $?g\ (c *s\ x) = c *s\ ?g\ x$
unfolding *hc tha g*(*2*)[*rule-format*, *OF conjunct1*[*OF h*, *OF x*]]
by (*vector ring-simps*)**}**
moreover
{fix x **assume** x: $x \in (insert\ a\ b)$
{assume *xa*: $x = a$
have *ha1*: $1 = ?h\ a$
apply (*rule conjunct2*[*OF h*, *rule-format*])

```
      apply (metis span-superset insertI1)
      using conjunct1[OF h, OF span-superset, OF insertI1]
      by (auto simp add: span-0)

    from xa ha1[symmetric] have ?g x = f x
      apply simp
      using g(2)[rule-format, OF span-0, of 0]
      by simp}
  moreover
  {assume xb: x ∈ b
    have h0: 0 = ?h x
      apply (rule conjunct2[OF h, rule-format])
      apply (metis  span-superset insertI1 xb x)
      apply simp
      apply (metis span-superset xb)
      done
    have ?g x = f x
      by (simp add: h0[symmetric] g(3)[rule-format, OF xb])}
  ultimately have ?g x = f x using x by blast }
 ultimately show ?case apply − apply (rule exI[where x=?g]) by blast
qed

lemma linear-independent-extend:
 assumes iB: independent (B:: (real ^'n::finite) set)
 shows ∃ g. linear g ∧ (∀ x∈B. g x = f x)
proof−
 from maximal-independent-subset-extend[of B UNIV] iB
 obtain C where C: B ⊆ C independent C ⋀x. x ∈ span C by auto

 from C(2) independent-bound[of C] linear-independent-extend-lemma[of C f]
 obtain g where g: (∀ x∈ span C. ∀ y∈ span C. g (x + y) = g x + g y)
        ∧ (∀ x∈ span C. ∀ c. g (c∗s x) = c ∗s g x)
        ∧ (∀ x∈ C. g x = f x) by blast
 from g show ?thesis unfolding linear-def using C
  apply clarsimp by blast
qed



lemma card-le-inj: assumes fA: finite A and fB: finite B
 and c: card A ≤ card B shows (∃ f. f ' A ⊆ B ∧ inj-on f A)
using fB c
proof(induct arbitrary: B rule: finite-induct[OF fA])
 case 1 thus ?case by simp
next
 case (2 x s t)
 thus ?case
 proof(induct rule: finite-induct[OF 2.prems(1)])
  case 1    then show ?case by simp
```

```
  next
    case (2 y t)
    from 2.prems(1,2,5) 2.hyps(1,2) have cst:card s ≤ card t by simp
    from 2.prems(3) [OF 2.hyps(1) cst] obtain f where
      f: f ‘ s ⊆ t ∧ inj-on f s by blast
    from f 2.prems(2) 2.hyps(2) show ?case
      apply −
      apply (rule exI[where x = λz. if z = x then y else f z])
      by (auto simp add: inj-on-def)
  qed
qed

lemma card-subset-eq: assumes fB: finite B and AB: A ⊆ B and
  c: card A = card B
  shows A = B
proof−
  from fB AB have fA: finite A by (auto intro: finite-subset)
  from fA fB have fBA: finite (B − A) by auto
  have e: A ∩ (B − A) = {} by blast
  have eq: A ∪ (B − A) = B using AB by blast
  from card-Un-disjoint[OF fA fBA e, unfolded eq c]
  have card (B − A) = 0 by arith
  hence B − A = {} unfolding card-eq-0-iff using fA fB by simp
  with AB show A = B by blast
qed

lemma subspace-isomorphism:
  assumes s: subspace (S:: (real ^'n::finite) set)
  and t: subspace (T :: (real ^ 'm::finite) set)
  and d: dim S = dim T
  shows ∃f. linear f ∧ f ‘ S = T ∧ inj-on f S
proof−
  from basis-exists[of S] obtain B where
    B: B ⊆ S independent B S ⊆ span B B hassize dim S by blast
  from basis-exists[of T] obtain C where
    C: C ⊆ T independent C T ⊆ span C C hassize dim T by blast
  from B(4) C(4) card-le-inj[of B C] d obtain f where
    f: f ‘ B ⊆ C inj-on f B unfolding hassize-def by auto
  from linear-independent-extend[OF B(2)] obtain g where
    g: linear g ∀x∈ B. g x = f x by blast
  from B(4) have fB: finite B by (simp add: hassize-def)
  from C(4) have fC: finite C by (simp add: hassize-def)
  from inj-on-iff-eq-card[OF fB, of f] f(2)
  have card (f ‘ B) = card B by simp
  with B(4) C(4) have ceq: card (f ‘ B) = card C using d
    by (simp add: hassize-def)
  have g ‘ B = f ‘ B using g(2)
    by (auto simp add: image-iff)
  also have ... = C using card-subset-eq[OF fC f(1) ceq] .
```

```
  finally have gBC: g ‘ B = C .
  have gi: inj-on g B using f(2) g(2)
    by (auto simp add: inj-on-def)
  note g0 = linear-indep-image-lemma[OF g(1) fB, unfolded gBC, OF C(2) gi]
  {fix x y assume x: x ∈ S and y: y ∈ S and gxy:g x = g y
    from B(3) x y have x': x ∈ span B and y': y ∈ span B by blast+
    from gxy have th0: g (x − y) = 0 by (simp add: linear-sub[OF g(1)])
    have th1: x − y ∈ span B using x' y' by (metis span-sub)
    have x=y using g0[OF th1 th0] by simp }
  then have giS: inj-on g S
    unfolding inj-on-def by blast
  from span-subspace[OF B(1,3) s]
  have g ‘ S = span (g ‘ B) by (simp add: span-linear-image[OF g(1)])
  also have ... = span C unfolding gBC ..
  also have ... = T using span-subspace[OF C(1,3) t] .
  finally have gS: g ‘ S = T .
  from g(1) gS giS show ?thesis by blast
qed



lemma subspace-kernel:
  assumes lf: linear (f::'a::semiring-1 ^'n ⇒ -)
  shows subspace {x. f x = 0}
apply (simp add: subspace-def)
by (simp add: linear-add[OF lf] linear-cmul[OF lf] linear-0[OF lf])

lemma linear-eq-0-span:
  assumes lf: linear f and f0: ∀x∈B. f x = 0
  shows ∀x ∈ span B. f x = (0::'a::semiring-1 ^'n)
proof
  fix x assume x: x ∈ span B
  let ?P = λx. f x = 0
  from subspace-kernel[OF lf] have subspace ?P unfolding Collect-def .
  with x f0 span-induct[of B ?P x] show f x = 0 by blast
qed

lemma linear-eq-0:
  assumes lf: linear f and SB: S ⊆ span B and f0: ∀x∈B. f x = 0
  shows ∀x ∈ S. f x = (0::'a::semiring-1^'n)
  by (metis linear-eq-0-span[OF lf] subset-eq SB f0)

lemma linear-eq:
  assumes lf: linear (f::'a::ring-1^'n ⇒ -) and lg: linear g and S: S ⊆ span B
  and fg: ∀ x∈ B. f x = g x
  shows ∀x∈ S. f x = g x
proof−
  let ?h = λx. f x − g x
  from fg have fg': ∀x∈ B. ?h x = 0 by simp
```

from *linear-eq-0*[*OF linear-compose-sub*[*OF lf lg*] *S fg'*]
show *?thesis* **by** *simp*
qed

lemma *linear-eq-stdbasis*:
assumes *lf*: *linear* (*f*::*'a*::*ring-1^'m*::*finite* ⇒ *'a^'n*::*finite*) **and** *lg*: *linear g*
and *fg*: ∀ *i*. *f* (*basis i*) = *g*(*basis i*)
shows *f* = *g*
proof−
let *?U* = *UNIV* :: *'m set*
let *?I* = {*basis i*:: *'a^'m*|*i*. *i* ∈ *?U*}
{fix *x* **assume** *x*: *x* ∈ (*UNIV* :: (*'a^'m*) *set*)
from *equalityD2*[*OF span-stdbasis*]
have *IU*: (*UNIV* :: (*'a^'m*) *set*) ⊆ *span ?I* **by** *blast*
from *linear-eq*[*OF lf lg IU*] *fg x*
have *f x* = *g x* **unfolding** *Collect-def Ball-def mem-def* **by** *metis***}**
then show *?thesis* **by** (*auto intro*: *ext*)
qed

lemma *bilinear-eq*:
assumes *bf*: *bilinear* (*f*:: *'a*::*ring^'m* ⇒ *'a^'n* ⇒ *'a^'p*)
and *bg*: *bilinear g*
and *SB*: *S* ⊆ *span B* **and** *TC*: *T* ⊆ *span C*
and *fg*: ∀ *x*∈ *B*. ∀ *y*∈ *C*. *f x y* = *g x y*
shows ∀ *x*∈*S*. ∀ *y*∈*T*. *f x y* = *g x y*
proof−
let *?P* = λ*x*. ∀ *y*∈ *span C*. *f x y* = *g x y*
from *bf bg* **have** *sp*: *subspace ?P*
unfolding *bilinear-def linear-def subspace-def bf bg*
by(*auto simp add*: *span-0 mem-def bilinear-lzero*[*OF bf*] *bilinear-lzero*[*OF bg*]
span-add Ball-def intro: *bilinear-ladd*[*OF bf*])

have ∀ *x* ∈ *span B*. ∀ *y*∈ *span C*. *f x y* = *g x y*
apply −
apply (*rule ballI*)
apply (*rule span-induct*[*of B ?P*])
defer
apply (*rule sp*)
apply *assumption*
apply (*clarsimp simp add*: *Ball-def*)
apply (*rule-tac P*=λ*y*. *f xa y* = *g xa y* **and** *S*=*C* **in** *span-induct*)
using *fg*
apply (*auto simp add*: *subspace-def*)
using *bf bg* **unfolding** *bilinear-def linear-def*
by(*auto simp add*: *span-0 mem-def bilinear-rzero*[*OF bf*] *bilinear-rzero*[*OF bg*]
span-add Ball-def intro: *bilinear-ladd*[*OF bf*])
then show *?thesis* **using** *SB TC* **by** (*auto intro*: *ext*)

qed

lemma *bilinear-eq-stdbasis*:
assumes *bf*: *bilinear* (*f*:: *'a*::*ring-1^'m*::*finite* ⇒ *'a^'n*::*finite* ⇒ *'a^'p*)
and *bg*: *bilinear g*
and *fg*: ∀ *i j*. *f* (*basis i*) (*basis j*) = *g* (*basis i*) (*basis j*)
shows *f* = *g*
proof−
from *fg* **have** *th*: ∀ *x* ∈ {*basis i*| *i*. *i*∈ (*UNIV* :: *'m set*)}. ∀ *y*∈ {*basis j* |*j*. *j* ∈ (*UNIV* :: *'n set*)}. *f x y* = *g x y* **by** *blast*
from *bilinear-eq*[*OF bf bg equalityD2*[*OF span-stdbasis*] *equalityD2*[*OF span-stdbasis*] *th*] **show** *?thesis* **by** (*blast intro*: *ext*)
qed

lemma *left-invertible-transp*:
(∃(*B*::*'a^'n^'m*). *B* ** *transp* (*A*::*'a^'n^'m*) = *mat* (*1*::*'a*::*comm-semiring-1*))
⟷ (∃(*B*::*'a^'m^'n*). *A* ** *B* = *mat 1*)
by (*metis matrix-transp-mul transp-mat transp-transp*)

lemma *right-invertible-transp*:
(∃(*B*::*'a^'n^'m*). *transp* (*A*::*'a^'n^'m*) ** *B* = *mat* (*1*::*'a*::*comm-semiring-1*))
⟷ (∃(*B*::*'a^'m^'n*). *B* ** *A* = *mat 1*)
by (*metis matrix-transp-mul transp-mat transp-transp*)

lemma *linear-injective-left-inverse*:
assumes *lf*: *linear* (*f*::*real ^'n*::*finite* ⇒ *real ^'m*::*finite*) **and** *fi*: *inj f*
shows ∃ *g*. *linear g* ∧ *g o f* = *id*
proof−
from *linear-independent-extend*[*OF independent-injective-image*, *OF independent-stdbasis*, *OF lf fi*]
obtain *h*:: *real ^'m* ⇒ *real ^'n* **where** *h*: *linear h* ∀ *x* ∈ *f* ' {*basis i*|*i*. *i* ∈ (*UNIV*::*'n set*)}. *h x* = *inv f x* **by** *blast*
from *h*(*2*)
have *th*: ∀ *i*. (*h* ∘ *f*) (*basis i*) = *id* (*basis i*)
using *inv-o-cancel*[*OF fi*, *unfolded stupid-ext*[*symmetric*] *id-def o-def*]
by *auto*

from *linear-eq-stdbasis*[*OF linear-compose*[*OF lf h*(*1*)] *linear-id th*]
have *h o f* = *id* .
then show *?thesis* **using** *h*(*1*) **by** *blast*
qed

lemma *linear-surjective-right-inverse*:
assumes *lf*: *linear* (*f*:: *real ^'m*::*finite* ⇒ *real ^'n*::*finite*) **and** *sf*: *surj f*
shows ∃ *g*. *linear g* ∧ *f o g* = *id*
proof−
from *linear-independent-extend*[*OF independent-stdbasis*]

```
  obtain h:: real ^'n ⇒ real ^'m where
    h: linear h ∀ x∈ {basis i| i. i∈ (UNIV :: 'n set)}. h x = inv f x by blast
  from h(2)
  have th: ∀ i. (f o h) (basis i) = id (basis i)
    using sf
    apply (auto simp add: surj-iff o-def stupid-ext[symmetric])
    apply (erule-tac x=basis i in allE)
    by auto

  from linear-eq-stdbasis[OF linear-compose[OF h(1) lf] linear-id th]
  have f o h = id .
  then show ?thesis using h(1) by blast
qed

lemma matrix-left-invertible-injective:
(∃ B. (B::real^'m^'n) ** (A::real^'n::finite^'m::finite) = mat 1) ⟷ (∀ x y. A *v
x = A *v y ⟶ x = y)
proof−
  {fix B:: real^'m^'n and x y assume B: B ** A = mat 1 and xy: A *v x =
A*v y
    from xy have B*v (A *v x) = B *v (A*v y) by simp
    hence x = y
      unfolding matrix-vector-mul-assoc B matrix-vector-mul-lid .}
  moreover
  {assume A: ∀ x y. A *v x = A *v y ⟶ x = y
    hence i: inj (op *v A) unfolding inj-on-def by auto
    from linear-injective-left-inverse[OF matrix-vector-mul-linear i]
    obtain g where g: linear g g o op *v A = id by blast
    have matrix g ** A = mat 1
     unfolding matrix-eq matrix-vector-mul-lid matrix-vector-mul-assoc[symmetric]
matrix-works[OF g(1)]
      using g(2) by (simp add: o-def id-def stupid-ext)
    then have ∃ B. (B::real ^'m^'n) ** A = mat 1 by blast}
  ultimately show ?thesis by blast
qed

lemma matrix-left-invertible-ker:
  (∃ B. (B::real ^'m::finite^'n::finite) ** (A::real^'n^'m) = mat 1) ⟷ (∀ x. A *v
x = 0 ⟶ x = 0)
  unfolding matrix-left-invertible-injective
  using linear-injective-0[OF matrix-vector-mul-linear, of A]
  by (simp add: inj-on-def)

lemma matrix-right-invertible-surjective:
(∃ B. (A::real^'n::finite^'m::finite) ** (B::real^'m^'n) = mat 1) ⟷ surj (λx. A
*v x)
proof−
  {fix B :: real ^'m^'n assume AB: A ** B = mat 1
    {fix x :: real ^ 'm
```

```
    have A *v (B *v x) = x
      by (simp add: matrix-vector-mul-lid matrix-vector-mul-assoc AB)}
   hence surj (op *v A) unfolding surj-def by metis }
 moreover
 {assume sf: surj (op *v A)
  from linear-surjective-right-inverse[OF matrix-vector-mul-linear sf]
  obtain g:: real ^'m ⇒ real ^'n where g: linear g op *v A o g = id
    by blast

  have A ** (matrix g) = mat 1
    unfolding matrix-eq  matrix-vector-mul-lid
      matrix-vector-mul-assoc[symmetric] matrix-works[OF g(1)]
    using g(2) unfolding o-def stupid-ext[symmetric] id-def
    .
  hence ∃ B. A ** (B::real^'m^'n) = mat 1 by blast
 }
 ultimately show ?thesis unfolding surj-def by blast
qed

lemma matrix-left-invertible-independent-columns:
 fixes A :: real^'n::finite^'m::finite
 shows (∃ (B::real ^'m^'n). B ** A = mat 1) ⟷ (∀ c. setsum (λi. c i *s column
i A) (UNIV :: 'n set) = 0 ⟶ (∀ i. c i = 0))
  (is ?lhs ⟷ ?rhs)
proof−
 let ?U = UNIV :: 'n set
 {assume k: ∀ x. A *v x = 0 ⟶ x = 0
  {fix c i assume c: setsum (λi. c i *s column i A) ?U = 0
    and i: i ∈ ?U
    let ?x = χ i. c i
    have th0:A *v ?x = 0
      using c
      unfolding matrix-mult-vsum Cart-eq
      by auto
    from k[rule-format, OF th0] i
    have c i = 0 by (vector Cart-eq)}
  hence ?rhs by blast}
 moreover
 {assume H: ?rhs
  {fix x assume x: A *v x = 0
    let ?c = λi. ((x$i )::real)
    from H[rule-format, of ?c, unfolded matrix-mult-vsum[symmetric], OF x]
    have x = 0 by vector}}
 ultimately show ?thesis unfolding matrix-left-invertible-ker by blast
qed

lemma matrix-right-invertible-independent-rows:
 fixes A :: real^'n::finite^'m::finite
 shows (∃ (B::real^'m^'n). A ** B = mat 1) ⟷ (∀ c. setsum (λi. c i *s row i
```

```
A) (UNIV :: 'm set) = 0 ⟶ (∀ i. c i = 0))
  unfolding left-invertible-transp[symmetric]
    matrix-left-invertible-independent-columns
  by (simp add: column-transp)

lemma matrix-right-invertible-span-columns:
   (∃ (B::real ^'n::finite^'m::finite). (A::real ^'m^'n) ** B = mat 1) ⟷ span
(columns A) = UNIV (is ?lhs = ?rhs)
proof−
  let ?U = UNIV :: 'm set
  have fU: finite ?U by simp
  have lhseq: ?lhs ⟷ (∀ y. ∃ (x::real^'m). setsum (λi. (x$i) *s column i A) ?U
= y)
    unfolding matrix-right-invertible-surjective matrix-mult-vsum surj-def
    apply (subst eq-commute) ..
  have rhseq: ?rhs ⟷ (∀ x. x ∈ span (columns A)) by blast
  {assume h: ?lhs
    {fix x:: real ^'n
      from h[unfolded lhseq, rule-format, of x] obtain y:: real ^'m
        where y: setsum (λi. (y$i) *s column i A) ?U = x by blast
      have x ∈ span (columns A)
        unfolding y[symmetric]
        apply (rule span-setsum[OF fU])
        apply clarify
        apply (rule span-mul)
        apply (rule span-superset)
        unfolding columns-def
        by blast}
    then have ?rhs unfolding rhseq by blast}
  moreover
  {assume h:?rhs
    let ?P = λ(y::real ^'n). ∃ (x::real^'m). setsum (λi. (x$i) *s column i A) ?U =
y
    {fix y have ?P y
      proof(rule span-induct-alt[of ?P columns A])
        show ∃ x::real ^ 'm. setsum (λi. (x$i) *s column i A) ?U = 0
          apply (rule exI[where x=0])
          by (simp add: zero-index vector-smult-lzero)
      next
        fix c y1 y2 assume y1: y1 ∈ columns A and y2: ?P y2
        from y1 obtain i where i: i ∈ ?U y1 = column i A
          unfolding columns-def by blast
        from y2 obtain x:: real ^'m where
          x: setsum (λi. (x$i) *s column i A) ?U = y2 by blast
        let ?x = (χ j. if j = i then c + (x$i) else (x$j))::real^'m
        show ?P (c*s y1 + y2)
            proof(rule exI[where x= ?x], vector, auto simp add: i x[symmetric]
cond-value-iff right-distrib cond-application-beta cong del: if-weak-cong)
          fix j
```

```
          have th: ∀ xa ∈ ?U. (if xa = i then (c + (x$i)) * ((column xa A)$j)
         else (x$xa) * ((column xa A$j))) = (if xa = i then c * ((column i A)$j)
else 0) + ((x$xa) * ((column xa A)$j)) using i(1)
            by (simp add: ring-simps)
          have setsum (λxa. if xa = i then (c + (x$i)) * ((column xa A)$j)
         else (x$xa) * ((column xa A$j))) ?U = setsum (λxa. (if xa = i then c *
((column i A)$j) else 0) + ((x$xa) * ((column xa A)$j))) ?U
            apply (rule setsum-cong[OF refl])
            using th by blast
          also have ... = setsum (λxa. if xa = i then c * ((column i A)$j) else
0) ?U + setsum (λxa. ((x$xa) * ((column xa A)$j))) ?U
            by (simp add: setsum-addf)
         also have ... = c * ((column i A)$j) + setsum (λxa. ((x$xa) * ((column
xa A)$j))) ?U
            unfolding setsum-delta[OF fU]
            using i(1) by simp
          finally show setsum (λxa. if xa = i then (c + (x$i)) * ((column xa
A)$j)
         else (x$xa) * ((column xa A$j))) ?U = c * ((column i A)$j) + setsum
(λxa. ((x$xa) * ((column xa A)$j))) ?U .
        qed
       next
        show y ∈ span (columns A) unfolding h by blast
       qed}
   then have ?lhs unfolding lhseq ..}
  ultimately show ?thesis by blast
qed

lemma matrix-left-invertible-span-rows:
  (∃(B::real^'m::finite^'n::finite). B ** (A::real^'n^'m) = mat 1) ⟷ span (rows
A) = UNIV
  unfolding right-invertible-transp[symmetric]
  unfolding columns-transp[symmetric]
  unfolding matrix-right-invertible-span-columns
 ..



lemma linear-injective-imp-surjective:
  assumes lf: linear (f:: real ^'n::finite ⇒ real ^'n) and fi: inj f
  shows surj f
proof−
  let ?U = UNIV :: (real ^'n) set
  from basis-exists[of ?U] obtain B
    where B: B ⊆ ?U independent B ?U ⊆ span B B hassize dim ?U
    by blast
  from B(4) have d: dim ?U = card B by (simp add: hassize-def)
  have th: ?U ⊆ span (f ' B)
    apply (rule card-ge-dim-independent)
```

```
    apply (rule independent-injective-image[OF B(2) lf fi])
    apply (rule order-eq-refl)
    apply (rule sym)
    unfolding d
    apply (rule card-image)
    apply (rule subset-inj-on[OF fi])
    by blast
  from th show ?thesis
    unfolding span-linear-image[OF lf] surj-def
    using B(3) by blast
qed



lemma surjective-iff-injective-gen:
  assumes fS: finite S and fT: finite T and c: card S = card T
  and ST: f ' S ⊆ T
  shows (∀ y ∈ T. ∃ x ∈ S. f x = y) ⟷ inj-on f S (is ?lhs ⟷ ?rhs)
proof−
  {assume h: ?lhs
    {fix x y assume x: x ∈ S and y: y ∈ S and f: f x = f y
      from x fS have S0: card S ≠ 0 by auto
      {assume xy: x ≠ y
        have th: card S ≤ card (f ' (S − {y}))
          unfolding c
          apply (rule card-mono)
          apply (rule finite-imageI)
          using fS apply simp
          using h xy x y f unfolding subset-eq image-iff
          apply auto
          apply (case-tac xa = f x)
          apply (rule bexI[where x=x])
          apply auto
          done
        also have  ... ≤ card (S −{y})
          apply (rule card-image-le)
          using fS by simp
        also have ... ≤ card S − 1 using y fS by simp
        finally have False  using S0 by arith }
      then have x = y by blast}
    then have ?rhs unfolding inj-on-def by blast}
  moreover
  {assume h: ?rhs
    have f ' S = T
      apply (rule card-subset-eq[OF fT ST])
      unfolding card-image[OF h] using c .
    then have ?lhs by blast}
  ultimately show ?thesis by blast
```

qed

lemma *linear-surjective-imp-injective*:
assumes *lf*: *linear* (*f*::*real* ^ *'n*::*finite* => *real* ^ *'n*) **and** *sf*: *surj f*
shows *inj f*
proof−
let *?U* = *UNIV* :: (*real* ^ *'n*) *set*
from *basis-exists*[*of ?U*] **obtain** *B*
where *B*: *B* ⊆ *?U independent B ?U* ⊆ *span B B hassize dim ?U*
by *blast*
{fix *x* **assume** *x*: *x* ∈ *span B* **and** *fx*: *f x* = *0*
from *B*(*4*) **have** *fB*: *finite B* **by** (*simp add*: *hassize-def*)
from *B*(*4*) **have** *d*: *dim ?U* = *card B* **by** (*simp add*: *hassize-def*)
have *fBi*: *independent* (*f* ‘ *B*)
apply (*rule card-le-dim-spanning*[*of f* ‘ *B ?U*])
apply *blast*
using *sf B*(*3*)
unfolding *span-linear-image*[*OF lf*] *surj-def subset-eq image-iff*
apply *blast*
using *fB* **apply** (*blast intro*: *finite-imageI*)
unfolding *d*
apply (*rule card-image-le*)
apply (*rule fB*)
done
have *th0*: *dim ?U* ≤ *card* (*f* ‘ *B*)
apply (*rule span-card-ge-dim*)
apply *blast*
unfolding *span-linear-image*[*OF lf*]
apply (*rule subset-trans*[**where** *B* = *f* ‘ *UNIV*])
using *sf* **unfolding** *surj-def* **apply** *blast*
apply (*rule image-mono*)
apply (*rule B*(*3*))
apply (*metis finite-imageI fB*)
done

moreover have *card* (*f* ‘ *B*) ≤ *card B*
by (*rule card-image-le*, *rule fB*)
ultimately have *th1*: *card B* = *card* (*f* ‘ *B*) **unfolding** *d* **by** *arith*
have *fiB*: *inj-on f B*
unfolding *surjective-iff-injective-gen*[*OF fB finite-imageI*[*OF fB*] *th1 subset-refl*,
symmetric] **by** *blast*
from *linear-indep-image-lemma*[*OF lf fB fBi fiB x*] *fx*
have *x* = *0* **by** *blast***}**
note *th* = *this*
from *th* **show** *?thesis* **unfolding** *linear-injective-0*[*OF lf*]
using *B*(*3*) **by** *blast*
qed

lemma *left-right-inverse-eq*:
assumes *fg*: $f\ o\ g = id$ **and** *gh*: $g\ o\ h = id$
shows $f = h$
proof−
have $f = f\ o\ (g\ o\ h)$ **unfolding** *gh* **by** *simp*
also have $\ldots = (f\ o\ g)\ o\ h$ **by** (*simp add*: *o-assoc*)
finally show $f = h$ **unfolding** *fg* **by** *simp*
qed

lemma *isomorphism-expand*:
$f\ o\ g = id \wedge g\ o\ f = id \longleftrightarrow (\forall x.\ f(g\ x) = x) \wedge (\forall x.\ g(f\ x) = x)$
by (*simp add*: *expand-fun-eq o-def id-def*)

lemma *linear-injective-isomorphism*:
assumes *lf*: *linear* $(f :: real\hat{}'n::finite \Rightarrow real\ \hat{}'n)$ **and** *fi*: *inj f*
shows $\exists f'.\ linear\ f' \wedge (\forall x.\ f'\ (f\ x) = x) \wedge (\forall x.\ f\ (f'\ x) = x)$
unfolding *isomorphism-expand*[*symmetric*]
using *linear-surjective-right-inverse*[*OF lf linear-injective-imp-surjective*[*OF lf fi*]]
linear-injective-left-inverse[*OF lf fi*]
by (*metis left-right-inverse-eq*)

lemma *linear-surjective-isomorphism*:
assumes *lf*: *linear* $(f::real\ \hat{}'n::finite \Rightarrow real\ \hat{}'n)$ **and** *sf*: *surj f*
shows $\exists f'.\ linear\ f' \wedge (\forall x.\ f'\ (f\ x) = x) \wedge (\forall x.\ f\ (f'\ x) = x)$
unfolding *isomorphism-expand*[*symmetric*]
using *linear-surjective-right-inverse*[*OF lf sf*] *linear-injective-left-inverse*[*OF lf linear-surjective-imp-injective*[*O*
lf sf]]
by (*metis left-right-inverse-eq*)

lemma *linear-inverse-left*:
assumes *lf*: *linear* $(f::real\ \hat{}'n::finite \Rightarrow real\ \hat{}'n)$ **and** *lf'*: *linear f'*
shows $f\ o\ f' = id \longleftrightarrow f'\ o\ f = id$
proof−
{fix $f\ f'$:: $real\ \hat{}'n \Rightarrow real\ \hat{}'n$
assume *lf*: *linear f linear f'* **and** *f*: $f\ o\ f' = id$
from *f* **have** *sf*: *surj f*

apply (*auto simp add*: *o-def stupid-ext*[*symmetric*] *id-def surj-def*)
by *metis*
from *linear-surjective-isomorphism*[*OF lf*(*1*) *sf*] *lf f*
have $f'\ o\ f = id$ **unfolding** *stupid-ext*[*symmetric*] *o-def id-def*
by *metis***}**
then show *?thesis* **using** *lf lf'* **by** *metis*
qed

lemma *left-inverse-linear*:
assumes *lf*: *linear* (*f*::*real* ^'*n*::*finite* ⇒ *real* ^'*n*) **and** *gf*: *g o f* = *id*
shows *linear g*
proof−
from *gf* **have** *fi*: *inj f* **apply** (*auto simp add*: *inj-on-def o-def id-def stupid-ext*[*symmetric*])
by *metis*
from *linear-injective-isomorphism*[*OF lf fi*]
obtain *h*:: *real* ^'*n* ⇒ *real* ^'*n* **where**
h: *linear h* ∀ *x*. *h* (*f x*) = *x* ∀ *x*. *f* (*h x*) = *x* **by** *blast*
have *h* = *g* **apply** (*rule ext*) **using** *gf h*(*2*,*3*)
apply (*simp add*: *o-def id-def stupid-ext*[*symmetric*])
by *metis*
with *h*(*1*) **show** *?thesis* **by** *blast*
qed

lemma *right-inverse-linear*:
assumes *lf*: *linear* (*f*:: *real* ^'*n*::*finite* ⇒ *real* ^'*n*) **and** *gf*: *f o g* = *id*
shows *linear g*
proof−
from *gf* **have** *fi*: *surj f* **apply** (*auto simp add*: *surj-def o-def id-def stupid-ext*[*symmetric*])
by *metis*
from *linear-surjective-isomorphism*[*OF lf fi*]
obtain *h*:: *real* ^'*n* ⇒ *real* ^'*n* **where**
h: *linear h* ∀ *x*. *h* (*f x*) = *x* ∀ *x*. *f* (*h x*) = *x* **by** *blast*
have *h* = *g* **apply** (*rule ext*) **using** *gf h*(*2*,*3*)
apply (*simp add*: *o-def id-def stupid-ext*[*symmetric*])
by *metis*
with *h*(*1*) **show** *?thesis* **by** *blast*
qed

lemma *matrix-left-right-inverse*:
fixes *A A'* :: *real* ^'*n*::*finite*^'*n*
shows *A* ** *A'* = *mat 1* ⟷ *A'* ** *A* = *mat 1*
proof−
{**fix** *A A'* :: *real* ^'*n*^'*n* **assume** *AA'*: *A* ** *A'* = *mat 1*
have *sA*: *surj* (*op* ∗*v A*)
unfolding *surj-def*
apply *clarify*
apply (*rule-tac x*=(*A'* ∗*v y*) **in** *exI*)
by (*simp add*: *matrix-vector-mul-assoc AA' matrix-vector-mul-lid*)
from *linear-surjective-isomorphism*[*OF matrix-vector-mul-linear sA*]
obtain *f'* :: *real* ^'*n* ⇒ *real* ^'*n*
where *f'*: *linear f'* ∀ *x*. *f'* (*A* ∗*v x*) = *x* ∀ *x*. *A* ∗*v f' x* = *x* **by** *blast*
have *th*: *matrix f'* ** *A* = *mat 1*
by (*simp add*: *matrix-eq matrix-works*[*OF f'*(*1*)] *matrix-vector-mul-assoc*[*symmetric*]
matrix-vector-mul-lid f'(*2*)[*rule-format*])

hence (*matrix f'* ** *A*) ** *A'* = *mat 1* ** *A'* **by** *simp*
hence *matrix f'* = *A'* **by** (*simp add*: *matrix-mul-assoc*[*symmetric*] *AA' matrix-mul-rid matrix-mul-lid*)
hence *matrix f'* ** *A* = *A'* ** *A* **by** *simp*
hence *A'* ** *A* = *mat 1* **by** (*simp add*: *th*)**}**
then show *?thesis* **by** *blast*
qed

definition *rowvector v* = (χ *i j*. (*v*$*j*))

definition *columnvector v* = (χ *i j*. (*v*$*i*))

lemma *transp-columnvector*:
transp(*columnvector v*) = *rowvector v*
by (*simp add*: *transp-def rowvector-def columnvector-def Cart-eq*)

lemma *transp-rowvector*: *transp*(*rowvector v*) = *columnvector v*
by (*simp add*: *transp-def columnvector-def rowvector-def Cart-eq*)

lemma *dot-rowvector-columnvector*:
columnvector (*A* *v *v*) = *A* ** *columnvector v*
by (*vector columnvector-def matrix-matrix-mult-def matrix-vector-mult-def*)

lemma *dot-matrix-product*: (*x*::*'a*::*semiring-1*^*'n*::*finite*) · *y* = (((*rowvector x* ::*'a*^*'n*^*1*) ** (*columnvector y* :: *'a*^*1*^*'n*))$*1*)$*1*
by (*vector matrix-matrix-mult-def rowvector-def columnvector-def dot-def*)

lemma *dot-matrix-vector-mul*:
fixes *A B* :: *real* ^*'n*::*finite* ^*'n* **and** *x y* :: *real* ^*'n*
shows (*A* *v *x*) · (*B* *v *y*) =
(((*rowvector x* :: *real*^*'n*^*1*) ** ((*transp A* ** *B*) ** (*columnvector y* :: *real* ^*1*^*'n*)))$*1*)$*1*
unfolding *dot-matrix-product transp-columnvector*[*symmetric*]
dot-rowvector-columnvector matrix-transp-mul matrix-mul-assoc **..**

definition *infnorm* (*x*::*real*^*'n*::*finite*) = *rsup* {*abs*(*x*$*i*) |*i*. *i*∈ (*UNIV* :: *'n set*)}

lemma *numseg-dimindex-nonempty*: ∃ *i*. *i* ∈ (*UNIV* :: *'n set*)
by *auto*

lemma *infnorm-set-image*:
{*abs*(*x*$*i*) |*i*. *i*∈ (*UNIV* :: *'n set*)} =
(λ*i*. *abs*(*x*$*i*)) ' (*UNIV* :: *'n set*) **by** *blast*

lemma *infnorm-set-lemma*:

shows *finite* {*abs*((*x*::'*a*::*abs* ^'*n*::*finite*)$*i*) |*i*. *i*∈ (*UNIV* :: '*n set*)}
and {*abs*(*x*$*i*) |*i*. *i*∈ (*UNIV* :: '*n*::*finite set*)} ≠ {}
unfolding *infnorm-set-image*
by (*auto intro*: *finite-imageI*)

lemma *infnorm-pos-le*: *0* ≤ *infnorm* (*x*::*real*^'*n*::*finite*)
unfolding *infnorm-def*
unfolding *rsup-finite-ge-iff*[*OF infnorm-set-lemma*]
unfolding *infnorm-set-image*
by *auto*

lemma *infnorm-triangle*: *infnorm* ((*x*::*real*^'*n*::*finite*) + *y*) ≤ *infnorm x* + *infnorm y*
proof−
have *th*: ⋀*x y* (*z*::*real*). *x* − *y* <= *z* ⟷ *x* − *z* <= *y* **by** *arith*
have *th1*: ⋀*S f*. *f* ' *S* = {*f i*| *i*. *i* ∈ *S*} **by** *blast*
have *th2*: ⋀*x* (*y*::*real*). *abs*(*x* + *y*) − *abs*(*x*) <= *abs*(*y*) **by** *arith*
show *?thesis*
unfolding *infnorm-def*
unfolding *rsup-finite-le-iff*[*OF infnorm-set-lemma*]
apply (*subst diff-le-eq*[*symmetric*])
unfolding *rsup-finite-ge-iff*[*OF infnorm-set-lemma*]
unfolding *infnorm-set-image bex-simps*
apply (*subst th*)
unfolding *th1*
unfolding *rsup-finite-ge-iff*[*OF infnorm-set-lemma*]

unfolding *infnorm-set-image ball-simps bex-simps*
apply *simp*
apply (*metis th2*)
done
qed

lemma *infnorm-eq-0*: *infnorm x* = *0* ⟷ (*x*::*real* ^'*n*::*finite*) = *0*
proof−
have *infnorm x* <= *0* ⟷ *x* = *0*
unfolding *infnorm-def*
unfolding *rsup-finite-le-iff*[*OF infnorm-set-lemma*]
unfolding *infnorm-set-image ball-simps*
by *vector*
then show *?thesis* **using** *infnorm-pos-le*[*of x*] **by** *simp*
qed

lemma *infnorm-0*: *infnorm 0* = *0*
by (*simp add*: *infnorm-eq-0*)

lemma *infnorm-neg*: *infnorm* (− *x*) = *infnorm x*
unfolding *infnorm-def*
apply (*rule cong*[*of rsup rsup*])

apply *blast*
apply (*rule set-ext*)
apply *auto*
done

lemma *infnorm-sub*: *infnorm* (*x* − *y*) = *infnorm* (*y* − *x*)
proof−
have *y* − *x* = − (*x* − *y*) **by** *simp*
then show *?thesis* **by** (*metis infnorm-neg*)
qed

lemma *real-abs-sub-infnorm*: | *infnorm x* − *infnorm y*| ≤ *infnorm* (*x* − *y*)
proof−
have *th*: ⋀(*nx*::*real*) *n ny*. *nx* <= *n* + *ny* ⟹ *ny* <= *n* + *nx* ==> |*nx* − *ny*|
<= *n*
by *arith*
from *infnorm-triangle*[*of x* − *y y*] *infnorm-triangle*[*of x* − *y* −*x*]
have *ths*: *infnorm x* ≤ *infnorm* (*x* − *y*) + *infnorm y*
infnorm y ≤ *infnorm* (*x* − *y*) + *infnorm x*
by (*simp-all add*: *ring-simps infnorm-neg diff-def*[*symmetric*])
from *th*[*OF ths*] **show** *?thesis* **.**
qed

lemma *real-abs-infnorm*: |*infnorm x*| = *infnorm x*
using *infnorm-pos-le*[*of x*] **by** *arith*

lemma *component-le-infnorm*:
shows |*x*$*i*| ≤ *infnorm* (*x*::*real*^'*n*::*finite*)
proof−
let *?U* = *UNIV* :: '*n set*
let *?S* = {|*x*$*i*| |*i*. *i*∈ *?U*}
have *fS*: *finite ?S* **unfolding** *image-Collect*[*symmetric*]
apply (*rule finite-imageI*) **unfolding** *Collect-def mem-def* **by** *simp*
have *S0*: *?S* ≠ {} **by** *blast*
have *th1*: ⋀*S f*. *f* ' *S* = { *f i*| *i*. *i* ∈ *S*} **by** *blast*
from *rsup-finite-in*[*OF fS S0*] *rsup-finite-Ub*[*OF fS S0*]
show *?thesis* **unfolding** *infnorm-def isUb-def setle-def*
unfolding *infnorm-set-image ball-simps* **by** *auto*
qed

lemma *infnorm-mul-lemma*: *infnorm*(*a* ∗*s x*) <= |*a*| ∗ *infnorm x*
apply (*subst infnorm-def*)
unfolding *rsup-finite-le-iff*[*OF infnorm-set-lemma*]
unfolding *infnorm-set-image ball-simps*
apply (*simp add*: *abs-mult*)
apply (*rule allI*)
apply (*cut-tac component-le-infnorm*[*of x*])
apply (*rule mult-mono*)
apply *auto*

done

lemma *infnorm-mul*: *infnorm*(*a* ∗*s* *x*) = *abs a* ∗ *infnorm x*
proof−
{assume *a0*: *a* = *0* **hence** *?thesis* **by** (*simp add*: *infnorm-0*) **}**
moreover
{assume *a0*: *a* ≠ *0*
from *a0* **have** *th*: (*1*/*a*) ∗*s* (*a* ∗*s* *x*) = *x*
by (*simp add*: *vector-smult-assoc*)
from *a0* **have** *ap*: |*a*| > *0* **by** *arith*
from *infnorm-mul-lemma*[*of* *1*/*a* *a* ∗*s* *x*]
have *infnorm x* ≤ *1*/|*a*| ∗ *infnorm* (*a*∗*s* *x*)
unfolding *th* **by** *simp*
with *ap* **have** |*a*| ∗ *infnorm x* ≤ |*a*| ∗ (*1*/|*a*| ∗ *infnorm* (*a* ∗*s* *x*)) **by** (*simp add*: *field-simps*)
then have |*a*| ∗ *infnorm x* ≤ *infnorm* (*a*∗*s* *x*)
using *ap* **by** (*simp add*: *field-simps*)
with *infnorm-mul-lemma*[*of* *a* *x*] **have** *?thesis* **by** *arith* **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *infnorm-pos-lt*: *infnorm x* > *0* ⟷ *x* ≠ *0*
using *infnorm-pos-le*[*of* *x*] *infnorm-eq-0*[*of* *x*] **by** *arith*

lemma *infnorm-le-norm*: *infnorm x* ≤ *norm x*
unfolding *infnorm-def rsup-finite-le-iff*[*OF infnorm-set-lemma*]
unfolding *infnorm-set-image ball-simps*
by (*metis component-le-norm*)
lemma *card-enum*: *card* {*1* .. *n*} = *n* **by** *auto*
lemma *norm-le-infnorm*: *norm*(*x*) <= *sqrt*(*real CARD*('*n*)) ∗ *infnorm*(*x*::*real* ^'*n*::*finite*)
proof−
let *?d* = *CARD*('*n*)
have *real ?d* ≥ *0* **by** *simp*
hence *d2*: (*sqrt* (*real ?d*)) ^*2* = *real ?d*
by (*auto intro*: *real-sqrt-pow2*)
have *th*: *sqrt* (*real ?d*) ∗ *infnorm x* ≥ *0*
by (*simp add*: *zero-le-mult-iff real-sqrt-ge-0-iff infnorm-pos-le*)
have *th1*: *x*•*x* ≤ (*sqrt* (*real ?d*) ∗ *infnorm x*) ^*2*
unfolding *power-mult-distrib d2*
apply (*subst power2-abs*[*symmetric*])
unfolding *real-of-nat-def dot-def power2-eq-square*[*symmetric*]
apply (*subst power2-abs*[*symmetric*])
apply (*rule setsum-bounded*)
apply (*rule power-mono*)
unfolding *abs-of-nonneg*[*OF infnorm-pos-le*]
unfolding *infnorm-def rsup-finite-ge-iff*[*OF infnorm-set-lemma*]
unfolding *infnorm-set-image bex-simps*

apply *blast*
by (*rule abs-ge-zero*)
from *real-le-lsqrt*[*OF dot-pos-le th th1*]
show *?thesis* **unfolding** *real-vector-norm-def id-def* **.**
qed

lemma *norm-cauchy-schwarz-eq*: (*x*::*real* ^ '*n*::*finite*) · *y* = *norm x* ∗ *norm y* ⟷ *norm x* ∗*s y* = *norm y* ∗*s x* (**is** *?lhs* ⟷ *?rhs*)
proof−
{assume *h*: *x* = *0*
hence *?thesis* **by** *simp***}**
moreover
{assume *h*: *y* = *0*
hence *?thesis* **by** *simp***}**
moreover
{assume *x*: *x* ≠ *0* **and** *y*: *y* ≠ *0*
from *dot-eq-0*[*of norm y* ∗*s x* − *norm x* ∗*s y*]
have *?rhs* ⟷ (*norm y* ∗ (*norm y* ∗ *norm x* ∗ *norm x* − *norm x* ∗ (*x* · *y*)) − *norm x* ∗ (*norm y* ∗ (*y* · *x*) − *norm x* ∗ *norm y* ∗ *norm y*) = *0*)
using *x y*
unfolding *dot-rsub dot-lsub dot-lmult dot-rmult*
unfolding *norm-pow-2*[*symmetric*] *power2-eq-square diff-eq-0-iff-eq* **apply** (*simp add*: *dot-sym*)
apply (*simp add*: *ring-simps*)
apply *metis*
done
also have ... ⟷ (*2* ∗ *norm x* ∗ *norm y* ∗ (*norm x* ∗ *norm y* − *x* · *y*) = *0*)
using *x y*
by (*simp add*: *ring-simps dot-sym*)
also have ... ⟷ *?lhs* **using** *x y*
apply *simp*
by *metis*
finally have *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *norm-cauchy-schwarz-abs-eq*:
fixes *x y* :: *real* ^ '*n*::*finite*
shows *abs*(*x* · *y*) = *norm x* ∗ *norm y* ⟷
norm x ∗*s y* = *norm y* ∗*s x* ∨ *norm*(*x*) ∗*s y* = − *norm y* ∗*s x* (**is** *?lhs* ⟷ *?rhs*)
proof−
have *th*: ⋀(*x*::*real*) *a*. *a* ≥ *0* ⟹ *abs x* = *a* ⟷ *x* = *a* ∨ *x* = − *a* **by** *arith*
have *?rhs* ⟷ *norm x* ∗*s y* = *norm y* ∗*s x* ∨ *norm* (− *x*) ∗*s y* = *norm y* ∗*s* (− *x*)
apply *simp* **by** *vector*
also have ... ⟷(*x* · *y* = *norm x* ∗ *norm y* ∨

```
    (−x) · y = norm x * norm y)
   unfolding norm-cauchy-schwarz-eq[symmetric]
   unfolding norm-minus-cancel
    norm-mul by blast
 also have ... ⟷ ?lhs
   unfolding th[OF mult-nonneg-nonneg, OF norm-ge-zero[of x] norm-ge-zero[of
y]] dot-lneg
   by arith
 finally show ?thesis ..
qed

lemma norm-triangle-eq:
 fixes x y :: real ^ 'n::finite
 shows norm(x + y) = norm x + norm y ⟷ norm x *s y = norm y *s x
proof−
 {assume x: x =0 ∨ y =0
  hence ?thesis by (cases x=0, simp-all)}
 moreover
 {assume x: x ≠ 0 and y: y ≠ 0
  hence norm x ≠ 0 norm y ≠ 0
   by simp-all
  hence n: norm x > 0 norm y > 0
   using norm-ge-zero[of x] norm-ge-zero[of y]
   by arith+
  have th: ⋀(a::real) b c. a + b + c ≠ 0 ==> (a = b + c ⟷ a^2 = (b +
c) ^2) by algebra
  have norm(x + y) = norm x + norm y ⟷ norm(x + y) ^ 2 = (norm x +
norm y) ^2
   apply (rule th) using n norm-ge-zero[of x + y]
   by arith
  also have ... ⟷ norm x *s y = norm y *s x
   unfolding norm-cauchy-schwarz-eq[symmetric]
   unfolding norm-pow-2 dot-ladd dot-radd
   by (simp add: norm-pow-2[symmetric] power2-eq-square dot-sym ring-simps)
  finally have ?thesis .}
 ultimately show ?thesis by blast
qed



definition collinear S ⟷ (∃ u. ∀ x ∈ S. ∀ y ∈ S. ∃ c. x − y = c *s u)

lemma collinear-empty: collinear {} by (simp add: collinear-def)

lemma collinear-sing: collinear {(x::'a::ring-1 ^ 'n)}
 apply (simp add: collinear-def)
 apply (rule exI[where x=0])
 by simp
```

lemma *collinear-2*: *collinear* {(*x*::'*a*::*ring-1*^'*n*),*y*}
apply (*simp add*: *collinear-def*)
apply (*rule exI*[**where** *x*=*x* − *y*])
apply *auto*
apply (*rule exI*[**where** *x*=*0*], *simp*)
apply (*rule exI*[**where** *x*=*1*], *simp*)
apply (*rule exI*[**where** *x*=− *1*], *simp add*: *vector-sneg-minus1*[*symmetric*])
apply (*rule exI*[**where** *x*=*0*], *simp*)
done

lemma *collinear-lemma*: *collinear* {(*0*::*real*^'*n*),*x*,*y*} ⟷ *x* = *0* ∨ *y* = *0* ∨ (∃ *c*. *y* = *c* ∗*s* *x*) (**is** *?lhs* ⟷ *?rhs*)
proof−
{assume *x*=*0* ∨ *y* = *0* **hence** *?thesis*
by (*cases x* = *0*, *simp-all add*: *collinear-2 insert-commute*)**}**
moreover
{assume *x*: *x* ≠ *0* **and** *y*: *y* ≠ *0*
{assume *h*: *?lhs*
then obtain *u* **where** *u*: ∀ *x*∈ {*0*,*x*,*y*}. ∀ *y*∈ {*0*,*x*,*y*}. ∃ *c*. *x* − *y* = *c* ∗*s* *u*
unfolding *collinear-def* **by** *blast*
from *u*[*rule-format*, *of x 0*] *u*[*rule-format*, *of y 0*]
obtain *cx* **and** *cy* **where**
cx: *x* = *cx*∗*s* *u* **and** *cy*: *y* = *cy*∗*s* *u*
by *auto*
from *cx x* **have** *cx0*: *cx* ≠ *0* **by** *auto*
from *cy y* **have** *cy0*: *cy* ≠ *0* **by** *auto*
let *?d* = *cy* / *cx*
from *cx cy cx0* **have** *y* = *?d* ∗*s* *x*
by (*simp add*: *vector-smult-assoc*)
hence *?rhs* **using** *x y* **by** *blast***}**
moreover
{assume *h*: *?rhs*
then obtain *c* **where** *c*: *y* = *c*∗*s* *x* **using** *x y* **by** *blast*
have *?lhs* **unfolding** *collinear-def c*
apply (*rule exI*[**where** *x*=*x*])
apply *auto*
apply (*rule exI*[**where** *x*=− *1*], *simp only*: *vector-smult-lneg vector-smult-lid*)
apply (*rule exI*[**where** *x*= −*c*], *simp only*: *vector-smult-lneg*)
apply (*rule exI*[**where** *x*=*1*], *simp*)
apply (*rule exI*[**where** *x*=*1* − *c*], *simp add*: *vector-smult-lneg vector-sub-rdistrib*)
apply (*rule exI*[**where** *x*=*c* − *1*], *simp add*: *vector-smult-lneg vector-sub-rdistrib*)
done}
ultimately have *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *norm-cauchy-schwarz-equal*:
fixes *x y* :: *real* ^ '*n*::*finite*
shows *abs*(*x* ⋅ *y*) = *norm x* ∗ *norm y* ⟷ *collinear* {(*0*::*real*^'*n*),*x*,*y*}

unfolding *norm-cauchy-schwarz-abs-eq*
apply (*cases x=0*, *simp-all add*: *collinear-2*)
apply (*cases y=0*, *simp-all add*: *collinear-2 insert-commute*)
unfolding *collinear-lemma*
apply *simp*
apply (*subgoal-tac norm x* $\neq$ *0*)
apply (*subgoal-tac norm y* $\neq$ *0*)
apply (*rule iffI*)
apply (*cases norm x* $*_s$ *y* = *norm y* $*_s$ *x*)
apply (*rule exI*[**where** *x*=(*1*/*norm x*) * *norm y*])
apply (*drule sym*)
unfolding *vector-smult-assoc*[*symmetric*]
apply (*simp add*: *vector-smult-assoc field-simps*)
apply (*rule exI*[**where** *x*=(*1*/*norm x*) * − *norm y*])
apply *clarify*
apply (*drule sym*)
unfolding *vector-smult-assoc*[*symmetric*]
apply (*simp add*: *vector-smult-assoc field-simps*)
apply (*erule exE*)
apply (*erule ssubst*)
unfolding *vector-smult-assoc*
unfolding *norm-mul*
apply (*subgoal-tac norm x* * *c* = |*c*| * *norm x* ∨ *norm x* * *c* = − |*c*| * *norm x*)
apply (*case-tac c* <= *0*, *simp add*: *ring-simps*)
apply (*simp add*: *ring-simps*)
apply (*case-tac c* <= *0*, *simp add*: *ring-simps*)
apply (*simp add*: *ring-simps*)
apply *simp*
apply *simp*
done

end

31 Permutations: Permutations, both general and specifically on finite sets.

theory *Permutations*
imports *Finite-Cartesian-Product Parity Fact Main*
begin

definition *permutes* (**infixr** *permutes 41*) **where**
(*p permutes S*) ⟷ (∀ *x*. *x* ∉ *S* ⟶ *p x* = *x*) ∧ (∀ *y*. ∃!*x*. *p x* = *y*)

declare *swap-self*[*simp*]
lemma *swapid-sym*: *Fun.swap a b id = Fun.swap b a id*
by (*auto simp add*: *expand-fun-eq swap-def fun-upd-def*)
lemma *swap-id-refl*: *Fun.swap a a id = id* **by** *simp*
lemma *swap-id-sym*: *Fun.swap a b id = Fun.swap b a id*
by (*rule ext*, *simp add*: *swap-def*)
lemma *swap-id-idempotent*[*simp*]: *Fun.swap a b id o Fun.swap a b id = id*
by (*rule ext*, *auto simp add*: *swap-def*)

lemma *inv-unique-comp*: **assumes** *fg*: *f o g = id* **and** *gf*: *g o f = id*
shows *inv f = g*
using *fg gf inv-equality*[*of g f*] **by** (*auto simp add*: *expand-fun-eq*)

lemma *inverse-swap-id*: *inv* (*Fun.swap a b id*) = *Fun.swap a b id*
by (*rule inv-unique-comp*, *simp-all*)

lemma *swap-id-eq*: *Fun.swap a b id x* = (*if x = a then b else if x = b then a else x*)
by (*simp add*: *swap-def*)

lemma *permutes-in-image*: *p permutes S* ⟹ *p x* ∈ *S* ⟷ *x* ∈ *S*
unfolding *permutes-def* **by** *metis*

lemma *permutes-image*: **assumes** *pS*: *p permutes S* **shows** *p* ‘ *S = S*
using *pS*
unfolding *permutes-def*
apply −
apply (*rule set-ext*)
apply (*simp add*: *image-iff*)
apply *metis*
done

lemma *permutes-inj*: *p permutes S* ==> *inj p*
unfolding *permutes-def inj-on-def* **by** *blast*

lemma *permutes-surj*: *p permutes s* ==> *surj p*
unfolding *permutes-def surj-def* **by** *metis*

lemma *permutes-inv-o*: **assumes** *pS*: *p permutes S*
shows *p o inv p = id*
and *inv p o p = id*
using *permutes-inj*[*OF pS*] *permutes-surj*[*OF pS*]
unfolding *inj-iff*[*symmetric*] *surj-iff*[*symmetric*] **by** *blast+*

lemma *permutes-inverses*:
fixes $p :: 'a \Rightarrow 'a$
assumes *pS*: *p permutes S*
shows $p\ (inv\ p\ x) = x$
and $inv\ p\ (p\ x) = x$
using *permutes-inv-o*[*OF pS*, *unfolded expand-fun-eq o-def*] **by** *auto*

lemma *permutes-subset*: *p permutes S* $\Longrightarrow$ $S \subseteq T$ ==> *p permutes T*
unfolding *permutes-def* **by** *blast*

lemma *permutes-empty*[*simp*]: *p permutes* $\{\}$ $\longleftrightarrow$ $p = id$
unfolding *expand-fun-eq permutes-def* **apply** *simp* **by** *metis*

lemma *permutes-sing*[*simp*]: *p permutes* $\{a\}$ $\longleftrightarrow$ $p = id$
unfolding *expand-fun-eq permutes-def* **apply** *simp* **by** *metis*

lemma *permutes-univ*: *p permutes UNIV* $\longleftrightarrow$ $(\forall y.\ \exists!x.\ p\ x = y)$
unfolding *permutes-def* **by** *simp*

lemma *permutes-inv-eq*: *p permutes S* ==> $inv\ p\ y = x$ $\longleftrightarrow$ $p\ x = y$
unfolding *permutes-def inv-def* **apply** *auto*
apply (*erule allE*[**where** *x*=*y*])
apply (*erule allE*[**where** *x*=*y*])
apply (*rule someI-ex*) **apply** *blast*
apply (*rule some1-equality*)
apply *blast*
apply *blast*
done

lemma *permutes-swap-id*: $a \in S$ $\Longrightarrow$ $b \in S$ ==> *Fun.swap a b id permutes S*
unfolding *permutes-def swap-def fun-upd-def* **apply** *auto* **apply** *metis* **done**

lemma *permutes-superset*: *p permutes S* $\Longrightarrow$ $(\forall x \in S - T.\ p\ x = x)$ $\Longrightarrow$ *p permutes T*
apply (*simp add*: *Ball-def permutes-def Diff-iff*) **by** *metis*

lemma *permutes-id*: *id permutes S* **unfolding** *permutes-def* **by** *simp*

lemma *permutes-compose*: *p permutes S* $\Longrightarrow$ *q permutes S* ==> *q o p permutes S*
unfolding *permutes-def o-def* **by** *metis*

lemma *permutes-inv*: **assumes** *pS*: *p permutes S* **shows** *inv p permutes S*

using *pS* **unfolding** *permutes-def permutes-inv-eq*[*OF pS*] **by** *metis*

lemma *permutes-inv-inv*: **assumes** *pS*: *p permutes S* **shows** *inv* (*inv p*) = *p*
unfolding *expand-fun-eq permutes-inv-eq*[*OF pS*] *permutes-inv-eq*[*OF permutes-inv*[*OF pS*]]
by *blast*

lemma *permutes-insert-lemma*:
assumes *pS*: *p permutes* (*insert a S*)
shows *Fun.swap a* (*p a*) *id o p permutes S*
apply (*rule permutes-superset*[**where** *S* = *insert a S*])
apply (*rule permutes-compose*[*OF pS*])
apply (*rule permutes-swap-id*, *simp*)
using *permutes-in-image*[*OF pS*, *of a*] **apply** *simp*
apply (*auto simp add*: *Ball-def Diff-iff swap-def*)
done

lemma *permutes-insert*: {*p*. *p permutes* (*insert a S*)} =
(λ(*b*,*p*). *Fun.swap a b id o p*) ' {(*b*,*p*). *b* ∈ *insert a S* ∧ *p* ∈ {*p*. *p permutes S*}}
proof−

{fix *p*
{assume *pS*: *p permutes insert a S*
let *?b* = *p a*
let *?q* = *Fun.swap a* (*p a*) *id o p*
have *th0*: *p* = *Fun.swap a ?b id o ?q* **unfolding** *expand-fun-eq o-assoc* **by** *simp*
have *th1*: *?b* ∈ *insert a S* **unfolding** *permutes-in-image*[*OF pS*] **by** *simp*
from *permutes-insert-lemma*[*OF pS*] *th0 th1*
have ∃ *b q*. *p* = *Fun.swap a b id o q* ∧ *b* ∈ *insert a S* ∧ *q permutes S* **by** *blast*}
moreover
{fix *b q* **assume** *bq*: *p* = *Fun.swap a b id o q b* ∈ *insert a S q permutes S*
from *permutes-subset*[*OF bq*(*3*), *of insert a S*]
have *qS*: *q permutes insert a S* **by** *auto*
have *aS*: *a* ∈ *insert a S* **by** *simp*
from *bq*(*1*) *permutes-compose*[*OF qS permutes-swap-id*[*OF aS bq*(*2*)]]
have *p permutes insert a S* **by** *simp* **}**
ultimately have *p permutes insert a S* ⟷ (∃ *b q*. *p* = *Fun.swap a b id o q* ∧ *b* ∈ *insert a S* ∧ *q permutes S*) **by** *blast*}
thus *?thesis* **by** *auto*
qed

lemma *hassize-insert*: *a* ∉ *F* ⟹ *insert a F hassize n* ⟹ *F hassize* (*n* − *1*)

```
  by (auto simp add: hassize-def)

lemma hassize-permutations: assumes Sn: S hassize n
  shows {p. p permutes S} hassize (fact n)
proof−
  from Sn have fS:finite S by (simp add: hassize-def)

  have ∀ n. (S hassize n) ⟶ ({p. p permutes S} hassize (fact n))
  proof(rule finite-induct[where F = S])
    from fS show finite S .
  next
    show ∀ n. ({} hassize n) ⟶ ({p. p permutes {}} hassize fact n)
      by (simp add: hassize-def permutes-empty)
  next
    fix x F
    assume fF: finite F and xF: x ∉ F
      and H: ∀ n. (F hassize n) ⟶ ({p. p permutes F} hassize fact n)
    {fix n assume H0: insert x F hassize n
      let ?xF = {p. p permutes insert x F}
      let ?pF = {p. p permutes F}
      let ?pF' = {(b, p). b ∈ insert x F ∧ p ∈ ?pF}
      let ?g = (λ(b, p). Fun.swap x b id ∘ p)
      from permutes-insert[of x F]
      have xfgpF': ?xF = ?g ` ?pF' .
      from hassize-insert[OF xF H0] have Fs: F hassize (n − 1) .
      from H Fs have pFs: ?pF hassize fact (n − 1) by blast
      hence pF'f: finite ?pF' using H0 unfolding hassize-def
        apply (simp only: Collect-split Collect-mem-eq)
        apply (rule finite-cartesian-product)
        apply simp-all
        done

      have ginj: inj-on ?g ?pF'
      proof−
        {
          fix b p c q assume bp: (b,p) ∈ ?pF' and cq: (c,q) ∈ ?pF'
            and eq: ?g (b,p) = ?g (c,q)
            from bp cq have ths: b ∈ insert x F c ∈ insert x F x ∈ insert x F p
permutes F q permutes F by auto
          from ths(4) xF eq have b = ?g (b,p) x unfolding permutes-def
            by (auto simp add: swap-def fun-upd-def expand-fun-eq)
          also have ... = ?g (c,q) x using ths(5) xF eq
            by (auto simp add: swap-def fun-upd-def expand-fun-eq)
          also have ... = cusing ths(5) xF unfolding permutes-def
            by (auto simp add: swap-def fun-upd-def expand-fun-eq)
          finally have bc: b = c .
          hence Fun.swap x b id = Fun.swap x c id by simp
          with eq have Fun.swap x b id o p = Fun.swap x b id o q by simp
            hence Fun.swap x b id o (Fun.swap x b id o p) = Fun.swap x b id o
```

(*Fun.swap x b id o q*) **by** *simp*
        hence $p = q$ **by** (*simp add*: *o-assoc*)
        with *bc* **have** $(b,p) = (c,q)$ **by** *simp* **}**
       thus *?thesis* **unfolding** *inj-on-def* **by** *blast*
     qed
     from *xF H0* **have** *n0*: $n \neq 0$ **by** (*auto simp add*: *hassize-def*)
     hence $\exists m.\ n = Suc\ m$ **by** *arith*
     then obtain *m* **where** *n*[*simp*]: $n = Suc\ m$ **by** *blast*
     from *pFs H0* **have** *xFc*: *card ?xF* = *fact n*
      unfolding *xfgpF'* *card-image*[*OF ginj*] *hassize-def*
      apply (*simp only*: *Collect-split Collect-mem-eq card-cartesian-product*)
      by *simp*
     from *finite-imageI*[*OF pF'f*, *of ?g*] **have** *xFf*: *finite ?xF* **unfolding** *xfgpF'*
by *simp*
     have *?xF hassize fact n*
      using *xFf xFc*
      unfolding *hassize-def xFf* **by** *blast* **}**
   thus $\forall n.$ (*insert x F hassize n*) $\longrightarrow$ ({*p*. *p permutes insert x F*} *hassize fact n*)
    by *blast*
  qed
  with *Sn* **show** *?thesis* **by** *blast*
qed

lemma *finite-permutations*: *finite S* ==> *finite* {*p*. *p permutes S*}
  using *hassize-permutations*[*of S*] **unfolding** *hassize-def* **by** *blast*

lemma (**in** *ab-semigroup-mult*) *fold-image-permute*: **assumes** *fS*: *finite S* **and** *pS*:
p permutes S
  shows *fold-image times f z S* = *fold-image times* (*f o p*) *z S*
   using *fold-image-reindex*[*OF fS subset-inj-on*[*OF permutes-inj*[*OF pS*], *of S*,
simplified], *of f z*]
  unfolding *permutes-image*[*OF pS*] **.**
lemma (**in** *ab-semigroup-add*) *fold-image-permute*: **assumes** *fS*: *finite S* **and** *pS*:
p permutes S
  shows *fold-image plus f z S* = *fold-image plus* (*f o p*) *z S*
proof−
 interpret *ab-semigroup-mult plus* **apply** *unfold-locales* **apply** (*simp add*: *add-assoc*)
    apply (*simp add*: *add-commute*) **done**
 from *fold-image-reindex*[*OF fS subset-inj-on*[*OF permutes-inj*[*OF pS*], *of S*, *simplified*], *of f z*]
  show *?thesis*
  unfolding *permutes-image*[*OF pS*] **.**
qed

lemma *setsum-permute*: **assumes** *pS*: *p permutes S*

shows *setsum f S = setsum (f o p) S*
unfolding *setsum-def* **using** *fold-image-permute[of S p f 0] pS* **by** *clarsimp*

lemma *setsum-permute-natseg*:**assumes** *pS*: *p permutes {m .. n}*
shows *setsum f {m .. n} = setsum (f o p) {m .. n}*
using *setsum-permute[OF pS, of f] pS* **by** *blast*

lemma *setprod-permute*: **assumes** *pS*: *p permutes S*
shows *setprod f S = setprod (f o p) S*
unfolding *setprod-def*
using *ab-semigroup-mult-class.fold-image-permute[of S p f 1] pS* **by** *clarsimp*

lemma *setprod-permute-natseg*:**assumes** *pS*: *p permutes {m .. n}*
shows *setprod f {m .. n} = setprod (f o p) {m .. n}*
using *setprod-permute[OF pS, of f] pS* **by** *blast*

lemma *swap-id-common*: *a ≠ c ⟹ b ≠ c ⟹ Fun.swap a b id o Fun.swap a c id = Fun.swap b c id o Fun.swap a b id* **by** (*simp add*: *expand-fun-eq swap-def*)

lemma *swap-id-common'*: *~(a = b) ⟹ ~(a = c) ⟹ Fun.swap a c id o Fun.swap b c id = Fun.swap b c id o Fun.swap a b id* **by** (*simp add*: *expand-fun-eq swap-def*)

lemma *swap-id-independent*: *~(a = c) ⟹ ~(a = d) ⟹ ~(b = c) ⟹ ~(b = d) ==> Fun.swap a b id o Fun.swap c d id = Fun.swap c d id o Fun.swap a b id*
by (*simp add*: *swap-def expand-fun-eq*)

inductive *swapidseq* :: *nat ⇒ ('a ⇒ 'a) ⇒ bool* **where**
id[*simp*]: *swapidseq 0 id*
| *comp-Suc*: *swapidseq n p ⟹ a ≠ b ⟹ swapidseq (Suc n) (Fun.swap a b id o p)*

declare *id*[*unfolded id-def*, *simp*]
definition *permutation p ⟷ (∃ n. swapidseq n p)*

lemma *permutation-id*[*simp*]: *permutation id***unfolding** *permutation-def*
by (*rule exI*[**where** *x=0*], *simp*)

declare *permutation-id*[*unfolded id-def*, *simp*]

lemma *swapidseq-swap*: *swapidseq* (*if a = b then 0 else 1*) (*Fun.swap a b id*)
apply *clarsimp*
using *comp-Suc*[*of 0 id a b*] **by** *simp*

lemma *permutation-swap-id*: *permutation* (*Fun.swap a b id*)
apply (*cases a=b*, *simp-all*)
unfolding *permutation-def* **using** *swapidseq-swap*[*of a b*] **by** *blast*

lemma *swapidseq-comp-add*: *swapidseq n p* ⟹ *swapidseq m q* ==> *swapidseq* (*n + m*) (*p o q*)
proof (*induct n p arbitrary*: *m q rule*: *swapidseq.induct*)
case (*id m q*) **thus** *?case* **by** *simp*
next
case (*comp-Suc n p a b m q*)
have *th*: *Suc n + m = Suc* (*n + m*) **by** *arith*
show *?case* **unfolding** *th o-assoc*[*symmetric*]
apply (*rule swapidseq.comp-Suc*) **using** *comp-Suc.hyps*(*2*)[*OF comp-Suc.prems*] *comp-Suc.hyps*(*3*) **by** *blast+*
qed

lemma *permutation-compose*: *permutation p* ⟹ *permutation q* ==> *permutation*(*p o q*)
unfolding *permutation-def* **using** *swapidseq-comp-add*[*of - p - q*] **by** *metis*

lemma *swapidseq-endswap*: *swapidseq n p* ⟹ *a* ≠ *b* ==> *swapidseq* (*Suc n*) (*p o Fun.swap a b id*)
apply (*induct n p rule*: *swapidseq.induct*)
using *swapidseq-swap*[*of a b*]
by (*auto simp add*: *o-assoc*[*symmetric*] *intro*: *swapidseq.comp-Suc*)

lemma *swapidseq-inverse-exists*: *swapidseq n p* ==> ∃ *q*. *swapidseq n q* ∧ *p o q = id* ∧ *q o p = id*
proof(*induct n p rule*: *swapidseq.induct*)
case *id* **thus** *?case* **by** (*rule exI*[**where** *x*=*id*], *simp*)
next
case (*comp-Suc n p a b*)
from *comp-Suc.hyps* **obtain** *q* **where** *q*: *swapidseq n q p* ∘ *q = id q* ∘ *p = id* **by** *blast*
let *?q = q o Fun.swap a b id*
note *H = comp-Suc.hyps*
from *swapidseq-swap*[*of a b*] *H*(*3*) **have** *th0*: *swapidseq 1* (*Fun.swap a b id*) **by** *simp*
from *swapidseq-comp-add*[*OF q*(*1*) *th0*] **have** *th1*:*swapidseq* (*Suc n*) *?q* **by** *simp*
have *Fun.swap a b id o p o ?q = Fun.swap a b id o* (*p o q*) *o Fun.swap a b id* **by** (*simp add*: *o-assoc*)
also have ... = *id* **by** (*simp add*: *q*(*2*))
finally have *th2*: *Fun.swap a b id o p o ?q = id* **.**

have *?q* ∘ (*Fun.swap a b id* ∘ *p*) = *q* ∘ (*Fun.swap a b id o Fun.swap a b id*) ∘ *p*
by (*simp only*: *o-assoc*)
hence *?q* ∘ (*Fun.swap a b id* ∘ *p*) = *id* **by** (*simp add*: *q*(*3*))
with *th1 th2* **show** *?case* **by** *blast*
qed

lemma *swapidseq-inverse*: **assumes** *H*: *swapidseq n p* **shows** *swapidseq n* (*inv p*)
using *swapidseq-inverse-exists*[*OF H*] *inv-unique-comp*[*of p*] **by** *auto*

lemma *permutation-inverse*: *permutation p* ==> *permutation* (*inv p*)
using *permutation-def swapidseq-inverse* **by** *blast*

lemma *symmetry-lemma*:(⋀*a b c d*. *P a b c d* ==> *P a b d c*) ⟹
(⋀*a b c d*. *a* ≠ *b* ⟹ *c* ≠ *d* ⟹ (*a* = *c* ∧ *b* = *d* ∨ *a* = *c* ∧ *b* ≠ *d* ∨ *a* ≠ *c* ∧ *b* = *d* ∨ *a* ≠ *c* ∧ *a* ≠ *d* ∧ *b* ≠ *c* ∧ *b* ≠ *d*) ==> *P a b c d*)
==> (⋀*a b c d*. *a* ≠ *b* −−> *c* ≠ *d* ⟶ *P a b c d*) **by** *metis*

lemma *swap-general*: *a* ≠ *b* ⟹ *c* ≠ *d* ⟹ *Fun.swap a b id o Fun.swap c d id* = *id* ∨
(∃*x y z*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *z* ≠ *a* ∧ *x* ≠ *y* ∧ *Fun.swap a b id o Fun.swap c d id* = *Fun.swap x y id o Fun.swap a z id*)
proof−
assume *H*: *a*≠*b c*≠*d*
have *a* ≠ *b* ⟶ *c* ≠ *d* ⟶
(*Fun.swap a b id o Fun.swap c d id* = *id* ∨
(∃*x y z*. *x* ≠ *a* ∧ *y* ≠ *a* ∧ *z* ≠ *a* ∧ *x* ≠ *y* ∧ *Fun.swap a b id o Fun.swap c d id* = *Fun.swap x y id o Fun.swap a z id*))
apply (*rule symmetry-lemma*[**where** *a*=*a* **and** *b*=*b* **and** *c*=*c* **and** *d*=*d*])
apply (*simp-all only*: *swapid-sym*)
apply (*case-tac a* = *c* ∧ *b* = *d*, *clarsimp simp only*: *swapid-sym swap-id-idempotent*)
apply (*case-tac a* = *c* ∧ *b* ≠ *d*)
apply (*rule disjI2*)
apply (*rule-tac x*=*b* **in** *exI*)
apply (*rule-tac x*=*d* **in** *exI*)
apply (*rule-tac x*=*b* **in** *exI*)
apply (*clarsimp simp add*: *expand-fun-eq swap-def*)
apply (*case-tac a* ≠ *c* ∧ *b* = *d*)
apply (*rule disjI2*)
apply (*rule-tac x*=*c* **in** *exI*)
apply (*rule-tac x*=*d* **in** *exI*)
apply (*rule-tac x*=*c* **in** *exI*)
apply (*clarsimp simp add*: *expand-fun-eq swap-def*)
apply (*rule disjI2*)
apply (*rule-tac x*=*c* **in** *exI*)

apply (*rule-tac x=d* **in** *exI*)
apply (*rule-tac x=b* **in** *exI*)
apply (*clarsimp simp add*: *expand-fun-eq swap-def*)
done
with *H* **show** *?thesis* **by** *metis*
qed

lemma *swapidseq-id-iff*[*simp*]: *swapidseq 0 p* ⟷ *p = id*
using *swapidseq.cases*[*of 0 p p = id*]
by *auto*

lemma *swapidseq-cases*: *swapidseq n p* ⟷ (*n=0* ∧ *p = id* ∨ (∃ *a b q m. n = Suc m* ∧ *p = Fun.swap a b id o q* ∧ *swapidseq m q* ∧ *a≠ b*))
apply (*rule iffI*)
apply (*erule swapidseq.cases*[*of n p*])
apply *simp*
apply (*rule disjI2*)
apply (*rule-tac x= a* **in** *exI*)
apply (*rule-tac x= b* **in** *exI*)
apply (*rule-tac x= pa* **in** *exI*)
apply (*rule-tac x= na* **in** *exI*)
apply *simp*
apply *auto*
apply (*rule comp-Suc*, *simp-all*)
done
lemma *fixing-swapidseq-decrease*:
assumes *spn*: *swapidseq n p* **and** *ab*: *a≠b* **and** *pa*: (*Fun.swap a b id o p*) *a = a*
shows *n ≠ 0* ∧ *swapidseq* (*n − 1*) (*Fun.swap a b id o p*)
using *spn ab pa*
proof(*induct n arbitrary*: *p a b*)
case *0* **thus** *?case* **by** (*auto simp add*: *swap-def fun-upd-def*)
next
case (*Suc n p a b*)
from *Suc.prems*(*1*) *swapidseq-cases*[*of Suc n p*] **obtain**
c d q m **where** *cdqm*: *Suc n = Suc m p = Fun.swap c d id o q swapidseq m q c ≠ d n = m*
by *auto*
{assume *H*: *Fun.swap a b id o Fun.swap c d id = id*

have *?case* **apply** (*simp only*: *cdqm o-assoc H*)
by (*simp add*: *cdqm*)**}**
moreover
{ fix *x y z*
assume *H*: *x≠a y≠a z ≠a x ≠y*
Fun.swap a b id o Fun.swap c d id = Fun.swap x y id o Fun.swap a z id
from *H* **have** *az*: *a ≠ z* **by** *simp*

{fix *h* **have** (*Fun.swap x y id o h*) *a = a* ⟷ *h a = a*
using *H* **by** (*simp add*: *swap-def*)**}**

note *th3* = *this*
from *cdqm*(*2*) **have** *Fun.swap a b id o p* = *Fun.swap a b id o* (*Fun.swap c d id o q*) **by** *simp*
hence *Fun.swap a b id o p* = *Fun.swap x y id o* (*Fun.swap a z id o q*) **by** (*simp add*: *o-assoc H*)
hence (*Fun.swap a b id o p*) *a* = (*Fun.swap x y id o* (*Fun.swap a z id o q*)) *a* **by** *simp*
hence (*Fun.swap x y id o* (*Fun.swap a z id o q*)) *a* = *a* **unfolding** *Suc* **by** *metis*
hence *th1*: (*Fun.swap a z id o q*) *a* = *a* **unfolding** *th3* **.**
from *Suc.hyps*[*OF cdqm*(*3*)[*unfolded cdqm*(*5*)[*symmetric*]] *az th1*]
have *th2*: *swapidseq* (*n* − *1*) (*Fun.swap a z id o q*) *n* ≠ *0* **by** *blast*+
have *th*: *Suc n* − *1* = *Suc* (*n* − *1*) **using** *th2*(*2*) **by** *auto*
have *?case* **unfolding** *cdqm*(*2*) *H o-assoc th*
apply (*simp only*: *Suc-not-Zero simp-thms o-assoc*[*symmetric*])
apply (*rule comp-Suc*)
using *th2 H* **apply** *blast*+
done}
ultimately show *?case* **using** *swap-general*[*OF Suc.prems*(*2*) *cdqm*(*4*)] **by** *metis*
qed

lemma *swapidseq-identity-even*:
assumes *swapidseq n* (*id* :: *'a* ⇒ *'a*) **shows** *even n*
using ‹*swapidseq n id*›
proof(*induct n rule*: *nat-less-induct*)
fix *n*
assume *H*: ∀ *m*<*n*. *swapidseq m* (*id*::*'a* ⇒ *'a*) ⟶ *even m swapidseq n* (*id* :: *'a* ⇒ *'a*)
{**assume** *n* = *0* **hence** *even n* **by** *arith*}
moreover
{**fix** *a b* :: *'a* **and** *q m*
assume *h*: *n* = *Suc m* (*id* :: *'a* ⇒ *'a*) = *Fun.swap a b id* ∘ *q swapidseq m q a* ≠ *b*
from *fixing-swapidseq-decrease*[*OF h*(*3*,*4*), *unfolded h*(*2*)[*symmetric*]]
have *m*: *m* ≠ *0 swapidseq* (*m* − *1*) (*id* :: *'a* ⇒ *'a*) **by** *auto*
from *h m* **have** *mn*: *m* − *1* < *n* **by** *arith*
from *H*(*1*)[*rule-format*, *OF mn m*(*2*)] *h*(*1*) *m*(*1*) **have** *even n* **apply** *arith*
done}
ultimately show *even n* **using** *H*(*2*)[*unfolded swapidseq-cases*[*of n id*]] **by** *auto*
qed

definition *evenperm p* = *even* (*SOME n*. *swapidseq n p*)

lemma *swapidseq-even-even*: **assumes**
m: *swapidseq m p* **and** *n*: *swapidseq n p*

shows *even m* ⟷ *even n*
proof−
from *swapidseq-inverse-exists*[*OF n*]
obtain *q* **where** *q*: *swapidseq n q p* ∘ *q* = *id q* ∘ *p* = *id* **by** *blast*

from *swapidseq-identity-even*[*OF swapidseq-comp-add*[*OF m q*(*1*), *unfolded q*]]
show *?thesis* **by** *arith*
qed

lemma *evenperm-unique*: **assumes** *p*: *swapidseq n p* **and** *n*:*even n* = *b*
shows *evenperm p* = *b*
unfolding *n*[*symmetric*] *evenperm-def*
apply (*rule swapidseq-even-even*[**where** *p* = *p*])
apply (*rule someI*[**where** *x* = *n*])
using *p* **by** *blast*+

lemma *evenperm-id*[*simp*]: *evenperm id* = *True*
apply (*rule evenperm-unique*[**where** *n* = *0*]) **by** *simp-all*

lemma *evenperm-swap*: *evenperm* (*Fun.swap a b id*) = (*a* = *b*)
apply (*rule evenperm-unique*[**where** *n*=*if a* = *b then 0 else 1*])
by (*simp-all add*: *swapidseq-swap*)

lemma *evenperm-comp*:
assumes *p*: *permutation p* **and** *q*:*permutation q*
shows *evenperm* (*p o q*) = (*evenperm p* = *evenperm q*)
proof−
from *p q* **obtain**
n m **where** *n*: *swapidseq n p* **and** *m*: *swapidseq m q*
unfolding *permutation-def* **by** *blast*
note *nm* = *swapidseq-comp-add*[*OF n m*]
have *th*: *even* (*n* + *m*) = (*even n* ⟷ *even m*) **by** *arith*
from *evenperm-unique*[*OF n refl*] *evenperm-unique*[*OF m refl*]
evenperm-unique[*OF nm th*]
show *?thesis* **by** *blast*
qed

lemma *evenperm-inv*: **assumes** *p*: *permutation p*
shows *evenperm* (*inv p*) = *evenperm p*
proof−
from *p* **obtain** *n* **where** *n*: *swapidseq n p* **unfolding** *permutation-def* **by** *blast*
from *evenperm-unique*[*OF swapidseq-inverse*[*OF n*] *evenperm-unique*[*OF n refl*, *symmetric*]]
show *?thesis* **.**
qed

lemma *bij-iff*: *bij f* $\longleftrightarrow$ $(\forall x.\ \exists!y.\ f\ y = x)$
unfolding *bij-def inj-on-def surj-def*
apply *auto*
apply *metis*
apply *metis*
done

lemma *permutation-bijective*:
assumes *p*: *permutation p*
shows *bij p*
proof−
from *p* **obtain** *n* **where** *n*: *swapidseq n p* **unfolding** *permutation-def* **by** *blast*
from *swapidseq-inverse-exists*[*OF n*] **obtain** *q* **where**
q: *swapidseq n q p* ∘ *q* = *id q* ∘ *p* = *id* **by** *blast*
thus *?thesis* **unfolding** *bij-iff* **apply** (*auto simp add*: *expand-fun-eq*) **apply** *metis* **done**
qed

lemma *permutation-finite-support*: **assumes** *p*: *permutation p*
shows *finite* {*x*. *p x* ≠ *x*}
proof−
from *p* **obtain** *n* **where** *n*: *swapidseq n p* **unfolding** *permutation-def* **by** *blast*
from *n* **show** *?thesis*
proof(*induct n p rule*: *swapidseq.induct*)
case *id* **thus** *?case* **by** *simp*
next
case (*comp-Suc n p a b*)
let *?S* = *insert a* (*insert b* {*x*. *p x* ≠ *x*})
from *comp-Suc.hyps*(*2*) **have** *fS*: *finite ?S* **by** *simp*
from ‹*a* ≠ *b*› **have** *th*: {*x*. (*Fun.swap a b id o p*) *x* ≠ *x*} ⊆ *?S*
by (*auto simp add*: *swap-def*)
from *finite-subset*[*OF th fS*] **show** *?case* **.**
qed
qed

lemma *bij-inv-eq-iff*: *bij p* ==> *x* = *inv p y* $\longleftrightarrow$ *p x* = *y*
using *surj-f-inv-f*[*of p*] *inv-f-f*[*of f*] **by** (*auto simp add*: *bij-def*)

lemma *bij-swap-comp*:
assumes *bp*: *bij p* **shows** *Fun.swap a b id o p* = *Fun.swap* (*inv p a*) (*inv p b*) *p*
using *surj-f-inv-f*[*OF bij-is-surj*[*OF bp*]]
by (*simp add*: *expand-fun-eq swap-def bij-inv-eq-iff*[*OF bp*])

lemma *bij-swap-ompose-bij*: *bij p* ⟹ *bij* (*Fun.swap a b id o p*)
proof−
assume *H*: *bij p*
show *?thesis*
unfolding *bij-swap-comp*[*OF H*] *bij-swap-iff*
using *H* **.**
qed

lemma *permutation-lemma*:
assumes *fS*: *finite S* **and** *p*: *bij p* **and** *pS*: ∀ *x*. *x*∉ *S* ⟶ *p x* = *x*
shows *permutation p*
using *fS p pS*
proof(*induct S arbitrary*: *p rule*: *finite-induct*)
case (*empty p*) **thus** *?case* **by** *simp*
next
case (*insert a F p*)
let *?r* = *Fun.swap a* (*p a*) *id o p*
let *?q* = *Fun.swap a* (*p a*) *id o ?r*
have *raa*: *?r a* = *a* **by** (*simp add*: *swap-def*)
from *bij-swap-ompose-bij*[*OF insert*(*4*)]
have *br*: *bij ?r* **.**

from *insert raa* **have** *th*: ∀ *x*. *x* ∉ *F* ⟶ *?r x* = *x*
apply (*clarsimp simp add*: *swap-def*)
apply (*erule-tac x*=*x* **in** *allE*)
apply *auto*
unfolding *bij-iff* **apply** *metis*
done
from *insert*(*3*)[*OF br th*]
have *rp*: *permutation ?r* **.**
have *permutation ?q* **by** (*simp add*: *permutation-compose permutation-swap-id rp*)
thus *?case* **by** (*simp add*: *o-assoc*)
qed

lemma *permutation*: *permutation p* ⟷ *bij p* ∧ *finite* {*x*. *p x* ≠ *x*}
(**is** *?lhs* ⟷ *?b* ∧ *?f*)
proof
assume *p*: *?lhs*
from *p permutation-bijective permutation-finite-support* **show** *?b* ∧ *?f* **by** *auto*
next
assume *bf*: *?b* ∧ *?f*
hence *bf*: *?f ?b* **by** *blast*+
from *permutation-lemma*[*OF bf*] **show** *?lhs* **by** *blast*
qed

lemma *permutation-inverse-works*: **assumes** *p*: *permutation p*
shows *inv p o p* = *id p o inv p* = *id*
using *permutation-bijective*[*OF p*] *surj-iff bij-def inj-iff* **by** *auto*

lemma *permutation-inverse-compose*:
assumes *p*: *permutation p* **and** *q*: *permutation q*
shows *inv* (*p o q*) = *inv q o inv p*
proof−
note *ps* = *permutation-inverse-works*[*OF p*]
note *qs* = *permutation-inverse-works*[*OF q*]
have *p o q o* (*inv q o inv p*) = *p o* (*q o inv q*) *o inv p* **by** (*simp add*: *o-assoc*)
also have ... = *id* **by** (*simp add*: *ps qs*)
finally have *th0*: *p o q o* (*inv q o inv p*) = *id* **.**
have *inv q o inv p o* (*p o q*) = *inv q o* (*inv p o p*) *o q* **by** (*simp add*: *o-assoc*)
also have ... = *id* **by** (*simp add*: *ps qs*)
finally have *th1*: *inv q o inv p o* (*p o q*) = *id* **.**
from *inv-unique-comp*[*OF th0 th1*] **show** *?thesis* **.**
qed

lemma *permutation-permutes*: *permutation p* ⟷ (∃ *S*. *finite S* ∧ *p permutes S*)
unfolding *permutation permutes-def bij-iff*[*symmetric*]
apply (*rule iffI*, *clarify*)
apply (*rule exI*[**where** *x*={*x*. *p x* ≠ *x*}])
apply *simp*
apply *clarsimp*
apply (*rule-tac B*=*S* **in** *finite-subset*)
apply *auto*
done

lemma *permutes-induct*: *finite S* ⟹ *P id* ⟹ (⋀ *a b p*. *a* ∈ *S* ⟹ *b* ∈ *S* ⟹ *P p* ⟹ *P p* ⟹ *permutation p* ==> *P* (*Fun.swap a b id o p*))
==> (⋀*p*. *p permutes S* ==> *P p*)
proof(*induct S rule*: *finite-induct*)
case *empty* **thus** *?case* **by** *auto*
next
case (*insert x F p*)
let *?r* = *Fun.swap x* (*p x*) *id o p*
let *?q* = *Fun.swap x* (*p x*) *id o ?r*
have *qp*: *?q* = *p* **by** (*simp add*: *o-assoc*)
from *permutes-insert-lemma*[*OF insert.prems*(*3*)] *insert* **have** *Pr*: *P ?r* **by** *blast*
from *permutes-in-image*[*OF insert.prems*(*3*), *of x*]
have *pxF*: *p x* ∈ *insert x F* **by** *simp*
have *xF*: *x* ∈ *insert x F* **by** *simp*
have *rp*: *permutation ?r*

```
    unfolding permutation-permutes using insert.hyps(1)
      permutes-insert-lemma[OF insert.prems(3)] by blast
  from insert.prems(2)[OF xF pxF Pr Pr rp]
  show ?case  unfolding qp .
qed
```

```
definition sign p = (if evenperm p then (1::int) else −1)

lemma sign-nz: sign p ≠ 0 by (simp add: sign-def)
lemma sign-id: sign id = 1 by (simp add: sign-def)
lemma sign-inverse: permutation p ==> sign (inv p) = sign p
  by (simp add: sign-def evenperm-inv)
lemma sign-compose: permutation p ⟹ permutation q ==> sign (p o q) =
sign(p) * sign(q) by (simp add: sign-def evenperm-comp)
lemma sign-swap-id: sign (Fun.swap a b id) = (if a = b then 1 else −1)
  by (simp add: sign-def evenperm-swap)
lemma sign-idempotent: sign p * sign p = 1 by (simp add: sign-def)
```

```
lemma permutes-natset-le:
  assumes p: p permutes (S::'a::wellorder set) and le: ∀ i ∈ S.  p i <= i shows
p = id
proof−
  {fix n
    have p n = n
      using p le
    proof(induct n arbitrary: S rule: less-induct)
      fix n S assume H: ⋀m S. ⟦m < n; p permutes S; ∀ i∈S. p i ≤ i⟧ ⟹ p m
= m
        p permutes S ∀ i ∈S. p i ≤ i
      {assume n ∉ S
        with H(2) have p n = n unfolding permutes-def by metis}
      moreover
      {assume ns: n ∈ S
        from H(3)  ns have p n < n ∨ p n = n by auto
        moreover{assume h: p n < n
          from H h have p (p n) = p n by metis
         with permutes-inj[OF H(2)] have p n = n unfolding inj-on-def by blast
          with h have False by simp}
        ultimately have p n = n by blast }
      ultimately show p n = n  by blast
    qed}
```

thus *?thesis* **by** (*auto simp add*: *expand-fun-eq*)
qed

lemma *permutes-natset-ge*:
assumes *p*: *p permutes* (*S*::'*a*::*wellorder set*) **and** *le*: ∀ *i* ∈ *S*. *p i* ≥ *i* **shows** *p* = *id*
proof−
{fix *i* **assume** *i*: *i* ∈ *S*
from *i permutes-in-image*[*OF permutes-inv*[*OF p*]] **have** *inv p i* ∈ *S* **by** *simp*
with *le* **have** *p* (*inv p i*) ≥ *inv p i* **by** *blast*
with *permutes-inverses*[*OF p*] **have** *i* ≥ *inv p i* **by** *simp***}**
then have *th*: ∀ *i*∈*S*. *inv p i* ≤ *i* **by** *blast*
from *permutes-natset-le*[*OF permutes-inv*[*OF p*] *th*]
have *inv p* = *inv id* **by** *simp*
then show *?thesis*
apply (*subst permutes-inv-inv*[*OF p*, *symmetric*])
apply (*rule inv-unique-comp*)
apply *simp-all*
done
qed

lemma *image-inverse-permutations*: {*inv p* |*p*. *p permutes S*} = {*p*. *p permutes S*}
apply (*rule set-ext*)
apply *auto*
using *permutes-inv-inv permutes-inv* **apply** *auto*
apply (*rule-tac x*=*inv x* **in** *exI*)
apply *auto*
done

lemma *image-compose-permutations-left*:
assumes *q*: *q permutes S* **shows** {*q o p* | *p*. *p permutes S*} = {*p* . *p permutes S*}
apply (*rule set-ext*)
apply *auto*
apply (*rule permutes-compose*)
using *q* **apply** *auto*
apply (*rule-tac x* = *inv q o x* **in** *exI*)
by (*simp add*: *o-assoc permutes-inv permutes-compose permutes-inv-o*)

lemma *image-compose-permutations-right*:
assumes *q*: *q permutes S*
shows {*p o q* | *p*. *p permutes S*} = {*p* . *p permutes S*}
apply (*rule set-ext*)
apply *auto*
apply (*rule permutes-compose*)
using *q* **apply** *auto*
apply (*rule-tac x* = *x o inv q* **in** *exI*)
by (*simp add*: *o-assoc permutes-inv permutes-compose permutes-inv-o o-assoc*[*symmetric*])

lemma *permutes-in-seg*: *p permutes* {*1 ..n*} ⟹ *i* ∈ {*1..n*} ==> *1* <= *p i* ∧ *p i* <= *n*

apply (*simp add*: *permutes-def*)
apply *metis*
done

term *setsum*
lemma *setsum-permutations-inverse*: *setsum f* {*p*. *p permutes S*} = *setsum* (λ*p*. *f*(*inv p*)) {*p*. *p permutes S*} (**is** *?lhs* = *?rhs*)
proof−
let *?S* = {*p* . *p permutes S*}
have *th0*: *inj-on inv ?S*
proof(*auto simp add*: *inj-on-def*)
fix *q r*
assume *q*: *q permutes S* **and** *r*: *r permutes S* **and** *qr*: *inv q* = *inv r*
hence *inv* (*inv q*) = *inv* (*inv r*) **by** *simp*
with *permutes-inv-inv*[*OF q*] *permutes-inv-inv*[*OF r*]
show *q* = *r* **by** *metis*
qed
have *th1*: *inv* ‘ *?S* = *?S* **using** *image-inverse-permutations* **by** *blast*
have *th2*: *?rhs* = *setsum* (*f o inv*) *?S* **by** (*simp add*: *o-def*)
from *setsum-reindex*[*OF th0*, *of f*] **show** *?thesis* **unfolding** *th1 th2* **.**
qed

lemma *setum-permutations-compose-left*:
assumes *q*: *q permutes S*
shows *setsum f* {*p*. *p permutes S*} =
setsum (λ*p*. *f*(*q o p*)) {*p*. *p permutes S*} (**is** *?lhs* = *?rhs*)
proof−
let *?S* = {*p*. *p permutes S*}
have *th0*: *?rhs* = *setsum* (*f o* (*op o q*)) *?S* **by** (*simp add*: *o-def*)
have *th1*: *inj-on* (*op o q*) *?S*
apply (*auto simp add*: *inj-on-def*)
proof−
fix *p r*
assume *p permutes S* **and** *r*:*r permutes S* **and** *rp*: *q* ∘ *p* = *q* ∘ *r*
hence *inv q o q o p* = *inv q o q o r* **by** (*simp add*: *o-assoc*[*symmetric*])
with *permutes-inj*[*OF q*, *unfolded inj-iff*]

show *p* = *r* **by** *simp*
qed
have *th3*: (*op o q*) ‘ *?S* = *?S* **using** *image-compose-permutations-left*[*OF q*] **by** *auto*
from *setsum-reindex*[*OF th1*, *of f*]
show *?thesis* **unfolding** *th0 th1 th3* **.**
qed

lemma *sum-permutations-compose-right*:
assumes *q*: *q permutes S*
shows *setsum f* {*p*. *p permutes S*} =
setsum (λ*p*. *f*(*p o q*)) {*p*. *p permutes S*} (**is** *?lhs* = *?rhs*)
proof−
let *?S* = {*p*. *p permutes S*}
have *th0*: *?rhs* = *setsum* (*f o* (λ*p*. *p o q*)) *?S* **by** (*simp add*: *o-def*)
have *th1*: *inj-on* (λ*p*. *p o q*) *?S*
apply (*auto simp add*: *inj-on-def*)
proof−
fix *p r*
assume *p permutes S* **and** *r*:*r permutes S* **and** *rp*: *p o q* = *r o q*
hence *p o* (*q o inv q*) = *r o* (*q o inv q*) **by** (*simp add*: *o-assoc*)
with *permutes-surj*[*OF q*, *unfolded surj-iff*]

show *p* = *r* **by** *simp*
qed
have *th3*: (λ*p*. *p o q*) ' *?S* = *?S* **using** *image-compose-permutations-right*[*OF q*]
by *auto*
from *setsum-reindex*[*OF th1*, *of f*]
show *?thesis* **unfolding** *th0 th1 th3* **.**
qed

lemma *setsum-over-permutations-insert*:
assumes *fS*: *finite S* **and** *aS*: *a* ∉ *S*
shows *setsum f* {*p*. *p permutes* (*insert a S*)} = *setsum* (λ*b*. *setsum* (λ*q*. *f* (*Fun.swap a b id o q*)) {*p*. *p permutes S*}) (*insert a S*)
proof−
have *th0*: ⋀*f a b*. (λ(*b*,*p*). *f* (*Fun.swap a b id o p*)) = *f o* (λ(*b*,*p*). *Fun.swap a b id o p*)
by (*simp add*: *expand-fun-eq*)
have *th1*: ⋀*P Q*. *P* × *Q* = {(*a*,*b*). *a* ∈ *P* ∧ *b* ∈ *Q*} **by** *blast*
have *th2*: ⋀*P Q*. *P* ⟹ (*P* ⟹ *Q*) ⟹ *P* ∧ *Q* **by** *blast*
show *?thesis*
unfolding *permutes-insert*
unfolding *setsum-cartesian-product*
unfolding *th1*[*symmetric*]
unfolding *th0*
proof(*rule setsum-reindex*)
let *?f* = (λ(*b*, *y*). *Fun.swap a b id* ∘ *y*)
let *?P* = {*p*. *p permutes S*}
{**fix** *b c p q* **assume** *b*: *b* ∈ *insert a S* **and** *c*: *c* ∈ *insert a S*
and *p*: *p permutes S* **and** *q*: *q permutes S*
and *eq*: *Fun.swap a b id o p* = *Fun.swap a c id o q*
from *p q aS* **have** *pa*: *p a* = *a* **and** *qa*: *q a* = *a*

unfolding *permutes-def* **by** *metis+*
from *eq* **have** (*Fun.swap a b id o p*) *a* = (*Fun.swap a c id o q*) *a* **by** *simp*
hence *bc*: *b* = *c*
apply (*simp add*: *permutes-def pa qa o-def fun-upd-def swap-def id-def cong del*: *if-weak-cong*)
apply (*cases a* = *b*, *auto*)
by (*cases b* = *c*, *auto*)
from *eq*[*unfolded bc*] **have** (λ*p*. *Fun.swap a c id o p*) (*Fun.swap a c id o p*) = (λ*p*. *Fun.swap a c id o p*) (*Fun.swap a c id o q*) **by** *simp*
hence *p* = *q* **unfolding** *o-assoc swap-id-idempotent*
by (*simp add*: *o-def*)
with *bc* **have** *b* = *c* ∧ *p* = *q* **by** *blast*
}

then show *inj-on ?f* (*insert a S* × *?P*)
unfolding *inj-on-def*
apply *clarify* **by** *metis*
qed
qed

end

32 Determinants: Traces, Determinant of square matrices and some properties

theory *Determinants*
imports *Euclidean-Space Permutations*
begin

32.1 First some facts about products

lemma *setprod-insert-eq*: *finite A* ⟹ *setprod f* (*insert a A*) = (*if a* ∈ *A then setprod f A else f a* ∗ *setprod f A*)
apply *clarsimp*
by(*subgoal-tac insert a A* = *A*, *auto*)

lemma *setprod-add-split*:
assumes *mn*: (*m*::*nat*) <= *n* + *1*
shows *setprod f* {*m*.. *n*+*p*} = *setprod f* {*m* .. *n*} ∗ *setprod f* {*n*+*1*..*n*+*p*}
proof−
let *?A* = {*m* .. *n*+*p*}
let *?B* = {*m* .. *n*}
let *?C* = {*n*+*1*..*n*+*p*}
from *mn* **have** *un*: *?B* ∪ *?C* = *?A* **by** *auto*
from *mn* **have** *dj*: *?B* ∩ *?C* = {} **by** *auto*
have *f*: *finite ?B finite ?C* **by** *simp-all*
from *setprod-Un-disjoint*[*OF f dj*, *of f*, *unfolded un*] **show** *?thesis* **.**

qed

lemma *setprod-offset*: *setprod f* {(*m*::*nat*) + *p* .. *n* + *p*} = *setprod* (λ*i*. *f* (*i* + *p*)) {*m*..*n*}
apply (*rule setprod-reindex-cong*[**where** *f*=*op* + *p*])
apply (*auto simp add*: *image-iff Bex-def inj-on-def*)
apply *arith*
apply (*rule ext*)
apply (*simp add*: *add-commute*)
done

lemma *setprod-singleton*: *setprod f* {*x*} = *f x* **by** *simp*

lemma *setprod-singleton-nat-seg*: *setprod f* {*n*..*n*} = *f* (*n*::'*a*::*order*) **by** *simp*

lemma *setprod-numseg*: *setprod f* {*m*..*0*} = (*if m*=*0 then f 0 else 1*)
setprod f {*m* .. *Suc n*} = (*if m* ≤ *Suc n then f* (*Suc n*) ∗ *setprod f* {*m*..*n*}
else setprod f {*m*..*n*})
by (*auto simp add*: *atLeastAtMostSuc-conv*)

lemma *setprod-le*: **assumes** *fS*: *finite S* **and** *fg*: ∀ *x*∈*S*. *f x* ≥ *0* ∧ *f x* ≤ (*g x* :: '*a*::*ordered-idom*)
shows *setprod f S* ≤ *setprod g S*
using *fS fg*
apply(*induct S*)
apply *simp*
apply *auto*
apply (*rule mult-mono*)
apply (*auto intro*: *setprod-nonneg*)
done

lemma *setprod-inversef*: *finite A* ==> *setprod* (*inverse* ∘ *f*) *A* = (*inverse* (*setprod f A*) :: '*a*:: {*division-by-zero*, *field*})
apply (*erule finite-induct*)
apply (*simp*)
apply *simp*
done

lemma *setprod-le-1*: **assumes** *fS*: *finite S* **and** *f*: ∀ *x*∈*S*. *f x* ≥ *0* ∧ *f x* ≤ (*1*::'*a*::*ordered-idom*)
shows *setprod f S* ≤ *1*
using *setprod-le*[*OF fS f*] **unfolding** *setprod-1* **.**

32.2 Trace

definition *trace* :: '*a*::*semiring-1*^'*n*^'*n* ⇒ '*a* **where**
trace A = *setsum* (λ*i*. ((*A*$*i*)$*i*)) (*UNIV*::'*n set*)

lemma *trace-0*: *trace*(*mat 0*) = *0*
by (*simp add*: *trace-def mat-def*)

lemma *trace-I*: *trace*(*mat 1* :: *'a*::*semiring-1^'n^'n*) = *of-nat*(*CARD*('*n*))
by (*simp add*: *trace-def mat-def*)

lemma *trace-add*: *trace* ((*A*::*'a*::*comm-semiring-1^'n^'n*) + *B*) = *trace A* + *trace B*
by (*simp add*: *trace-def setsum-addf*)

lemma *trace-sub*: *trace* ((*A*::*'a*::*comm-ring-1^'n^'n*) − *B*) = *trace A* − *trace B*
by (*simp add*: *trace-def setsum-subtractf*)

lemma *trace-mul-sym*:*trace* ((*A*::*'a*::*comm-semiring-1^'n^'n*) ** *B*) = *trace* (*B***A*)
apply (*simp add*: *trace-def matrix-matrix-mult-def*)
apply (*subst setsum-commute*)
by (*simp add*: *mult-commute*)

definition *det*:: *'a*::*comm-ring-1^'n^'n* ⇒ *'a* **where**
det A = *setsum* (*λp*. *of-int* (*sign p*) ∗ *setprod* (*λi*. *Aip i*) (*UNIV* :: *'n set*)) {*p*. *p permutes* (*UNIV* :: *'n set*)}

lemma *setprod-permute*:
assumes *p*: *p permutes S*
shows *setprod f S* = *setprod* (*f o p*) *S*
proof−
{assume ¬ *finite S* **hence** *?thesis* **by** *simp***}**
moreover
{assume *fS*: *finite S*
then have *?thesis*
apply (*simp add*: *setprod-def cong del*:*strong-setprod-cong*)
apply (*rule ab-semigroup-mult.fold-image-permute*)
apply (*auto simp add*: *p*)
apply *unfold-locales*
done}
ultimately show *?thesis* **by** *blast*
qed

lemma *setproduct-permute-nat-interval*: *p permutes* {*m*::*nat* .. *n*} ==> *setprod f* {*m*..*n*} = *setprod* (*f o p*) {*m*..*n*}
by (*blast intro*!: *setprod-permute*)

lemma *det-transp*: *det* (*transp A*) = *det* (*A*::*'a*::*comm-ring-1* ^ *'n* ^ *'n*::*finite*)
proof−
let *?di* = λ*A i j*. *A*$*i*$*j*
let *?U* = (*UNIV* :: *'n set*)
have *fU*: *finite ?U* **by** *simp*
{**fix** *p* **assume** *p*: *p* ∈ {*p*. *p permutes ?U*}
from *p* **have** *pU*: *p permutes ?U* **by** *blast*
have *sth*: *sign* (*inv p*) = *sign p*
by (*metis sign-inverse fU p mem-def Collect-def permutation-permutes*)
from *permutes-inj*[*OF pU*]
have *pi*: *inj-on p ?U* **by** (*blast intro*: *subset-inj-on*)
from *permutes-image*[*OF pU*]
have *setprod* (λ*i*. *?di* (*transp A*) *i* (*inv p i*)) *?U* = *setprod* (λ*i*. *?di* (*transp A*) *i* (*inv p i*)) (*p* ' *?U*) **by** *simp*
also have ... = *setprod* ((λ*i*. *?di* (*transp A*) *i* (*inv p i*)) *o p*) *?U*
unfolding *setprod-reindex*[*OF pi*] **..**
also have ... = *setprod* (λ*i*. *?di A i* (*p i*)) *?U*
proof−
{**fix** *i* **assume** *i*: *i* ∈ *?U*
from *i permutes-inv-o*[*OF pU*] *permutes-in-image*[*OF pU*]
have ((λ*i*. *?di* (*transp A*) *i* (*inv p i*)) *o p*) *i* = *?di A i* (*p i*)
unfolding *transp-def* **by** (*simp add*: *expand-fun-eq*)}
then show *setprod* ((λ*i*. *?di* (*transp A*) *i* (*inv p i*)) *o p*) *?U* = *setprod* (λ*i*. *?di A i* (*p i*)) *?U* **by** (*auto intro*: *setprod-cong*)
qed
finally have *of-int* (*sign* (*inv p*)) ∗ (*setprod* (λ*i*. *?di* (*transp A*) *i* (*inv p i*)) *?U*) = *of-int* (*sign p*) ∗ (*setprod* (λ*i*. *?di A i* (*p i*)) *?U*) **using** *sth*
by *simp*}
then show *?thesis* **unfolding** *det-def* **apply** (*subst setsum-permutations-inverse*)
apply (*rule setsum-cong2*) **by** *blast*
qed

lemma *det-lowerdiagonal*:
fixes *A* :: *'a*::*comm-ring-1* ^ *'n* ^ *'n*::{*finite*,*wellorder*}
assumes *ld*: ⋀*i j*. *i* < *j* ⟹ *A*$*i*$*j* = *0*
shows *det A* = *setprod* (λ*i*. *A*$*i*$*i*) (*UNIV*:: *'n set*)
proof−
let *?U* = *UNIV*:: *'n set*
let *?PU* = {*p*. *p permutes ?U*}
let *?pp* = λ*p*. *of-int* (*sign p*) ∗ *setprod* (λ*i*. *A*$*i*$*p i*) (*UNIV* :: *'n set*)
have *fU*: *finite ?U* **by** *simp*
from *finite-permutations*[*OF fU*] **have** *fPU*: *finite ?PU* **.**
have *id0*: {*id*} ⊆ *?PU* **by** (*auto simp add*: *permutes-id*)
{**fix** *p* **assume** *p*: *p* ∈ *?PU* −{*id*}

```
    from p have pU: p permutes ?U and pid: p ≠ id by blast+
    from permutes-natset-le[OF pU] pid obtain i where
      i: p i > i by (metis not-le)
    from ld[OF i] have ex:∃ i ∈ ?U. A$i$p i = 0 by blast
    from setprod-zero[OF fU ex] have ?pp p = 0 by simp}
  then have p0: ∀ p ∈ ?PU −{id}. ?pp p = 0  by blast
  from setsum-mono-zero-cong-left[OF fPU id0 p0] show ?thesis
    unfolding det-def by (simp add: sign-id)
qed

lemma det-upperdiagonal:
  fixes A :: 'a::comm-ring-1^'n^'n::{finite,wellorder}
  assumes ld: ⋀i j. i > j ⟹ A$i$j = 0
  shows det A = setprod (λi. A$i$i) (UNIV:: 'n set)
proof−
  let ?U = UNIV:: 'n set
  let ?PU = {p. p permutes ?U}
  let ?pp = (λp. of-int (sign p) * setprod (λi. A$i$p i) (UNIV :: 'n set))
  have fU: finite ?U by simp
  from finite-permutations[OF fU] have fPU: finite ?PU .
  have id0: {id} ⊆ ?PU by (auto simp add: permutes-id)
  {fix p assume p: p ∈ ?PU −{id}
    from p have pU: p permutes ?U and pid: p ≠ id by blast+
    from permutes-natset-ge[OF pU] pid obtain i where
      i: p i < i by (metis not-le)
    from ld[OF i] have ex:∃ i ∈ ?U. A$i$p i = 0 by blast
    from setprod-zero[OF fU ex] have ?pp p = 0 by simp}
  then have p0: ∀ p ∈ ?PU −{id}. ?pp p = 0  by blast
  from   setsum-mono-zero-cong-left[OF fPU id0 p0] show ?thesis
    unfolding det-def by (simp add: sign-id)
qed

lemma det-diagonal:
  fixes A :: 'a::comm-ring-1^'n^'n::finite
  assumes ld: ⋀i j. i ≠ j ⟹ A$i$j = 0
  shows det A = setprod (λi. A$i$i) (UNIV::'n set)
proof−
  let ?U = UNIV:: 'n set
  let ?PU = {p. p permutes ?U}
  let ?pp = λp. of-int (sign p) * setprod (λi. A$i$p i) (UNIV :: 'n set)
  have fU: finite ?U by simp
  from finite-permutations[OF fU] have fPU: finite ?PU .
  have id0: {id} ⊆ ?PU by (auto simp add: permutes-id)
  {fix p assume p: p ∈ ?PU − {id}
    then have p ≠ id by simp
    then obtain i where i: p i ≠ i unfolding expand-fun-eq by auto
    from ld [OF i [symmetric]] have ex:∃ i ∈ ?U. A$i$p i = 0 by blast
    from setprod-zero [OF fU ex] have ?pp p = 0 by simp}
  then have p0: ∀ p ∈ ?PU − {id}. ?pp p = 0  by blast
```

```
  from setsum-mono-zero-cong-left[OF fPU id0 p0] show ?thesis
    unfolding det-def by (simp add: sign-id)
qed

lemma det-I: det (mat 1 :: 'a::comm-ring-1^'n^'n::finite) = 1
proof−
  let ?A = mat 1 :: 'a::comm-ring-1^'n^'n
  let ?U = UNIV :: 'n set
  let ?f = λi j. ?A$i$j
  {fix i assume i: i ∈ ?U
    have ?f i i = 1 using i by (vector mat-def)}
  hence th: setprod (λi. ?f i i) ?U = setprod (λx. 1) ?U
    by (auto intro: setprod-cong)
  {fix i j assume i: i ∈ ?U and j: j ∈ ?U and ij: i ≠ j
    have ?f i j = 0 using i j ij by (vector mat-def) }
  then have det ?A = setprod (λi. ?f i i) ?U using det-diagonal
    by blast
  also have ... = 1 unfolding th setprod-1 ..
  finally show ?thesis .
qed

lemma det-0: det (mat 0 :: 'a::comm-ring-1^'n^'n::finite) = 0
  by (simp add: det-def setprod-zero)

lemma det-permute-rows:
  fixes A :: 'a::comm-ring-1^'n^'n::finite
  assumes p: p permutes (UNIV :: 'n::finite set)
  shows det(χ i. A$p i :: 'a^'n^'n) = of-int (sign p) * det A
  apply (simp add: det-def setsum-right-distrib mult-assoc[symmetric])
  apply (subst sum-permutations-compose-right[OF p])
proof(rule setsum-cong2)
  let ?U = UNIV :: 'n set
  let ?PU = {p. p permutes ?U}
  fix q assume qPU: q ∈ ?PU
  have fU: finite ?U by simp
  from qPU have q: q permutes ?U by blast
  from p q have pp: permutation p and qp: permutation q
    by (metis fU permutation-permutes)+
  from permutes-inv[OF p] have ip: inv p permutes ?U .
    have setprod (λi. A$p i$ (q o p) i) ?U = setprod ((λi. A$p i$(q o p) i) o inv
p) ?U
      by (simp only: setprod-permute[OF ip, symmetric])
    also have ... = setprod (λi. A $ (p o inv p) i $ (q o (p o inv p)) i) ?U
      by (simp only: o-def)
   also have ... = setprod (λi. A$i$q i) ?U by (simp only: o-def permutes-inverses[OF
p])
    finally   have thp: setprod (λi. A$p i$ (q o p) i) ?U = setprod (λi. A$i$q i)
?U
      by blast
```

show *of-int (sign (q o p)) * setprod (λi. A$ p i$ (q o p) i) ?U = of-int (sign p) * of-int (sign q) * setprod (λi. Aiq i) ?U*
by (*simp only*: *thp sign-compose*[*OF qp pp*] *mult-commute of-int-mult*)
qed

lemma *det-permute-columns*:
fixes *A* :: *'a*::*comm-ring-1^'n^'n*::*finite*
assumes *p*: *p permutes* (*UNIV* :: *'n set*)
shows *det*(*χ i j. Ai p j* :: *'a^'n^'n*) = *of-int* (*sign p*) * *det A*
proof−
let *?Ap* = *χ i j. Ai p j* :: *'a^'n^'n*
let *?At* = *transp A*
have *of-int* (*sign p*) * *det A* = *det* (*transp* (*χ i. transp A $ p i*))
unfolding *det-permute-rows*[*OF p*, *of ?At*] *det-transp* **..**
moreover
have *?Ap* = *transp* (*χ i. transp A $ p i*)
by (*simp add*: *transp-def Cart-eq*)
ultimately show *?thesis* **by** *simp*
qed

lemma *det-identical-rows*:
fixes *A* :: *'a*::*ordered-idom^'n^'n*::*finite*
assumes *ij*: *i* ≠ *j*
and *r*: *row i A* = *row j A*
shows *det A* = *0*
proof−
have *tha*: ⋀(*a*::*'a*) *b*. *a* = *b* ==> *b* = − *a* ==> *a* = *0*
by *simp*
have *th1*: *of-int* (−*1*) = − *1* **by** (*metis of-int-1 of-int-minus number-of-Min*)
let *?p* = *Fun.swap i j id*
let *?A* = *χ i. A $?p i*
from *r* **have** *A* = *?A* **by** (*simp add*: *Cart-eq row-def swap-def*)
hence *det A* = *det ?A* **by** *simp*
moreover have *det A* = − *det ?A*
by (*simp add*: *det-permute-rows*[*OF permutes-swap-id*] *sign-swap-id ij th1*)
ultimately show *det A* = *0* **by** (*metis tha*)
qed

lemma *det-identical-columns*:
fixes *A* :: *'a*::*ordered-idom^'n^'n*::*finite*
assumes *ij*: *i* ≠ *j*
and *r*: *column i A* = *column j A*
shows *det A* = *0*
apply (*subst det-transp*[*symmetric*])
apply (*rule det-identical-rows*[*OF ij*])
by (*metis row-transp r*)

lemma *det-zero-row*:
fixes *A* :: *'a*::{*idom*, *ring-char-0*}*^'n^'n*::*finite*

assumes *r*: *row i A = 0*
shows *det A = 0*
using *r*
apply (*simp add*: *row-def det-def Cart-eq*)
apply (*rule setsum-0'*)
apply (*auto simp*: *sign-nz*)
done

lemma *det-zero-column*:
fixes *A* :: *'a*::{*idom*,*ring-char-0*}^*'n*^*'n*::*finite*
assumes *r*: *column i A = 0*
shows *det A = 0*
apply (*subst det-transp*[*symmetric*])
apply (*rule det-zero-row* [*of i*])
by (*metis row-transp r*)

lemma *det-row-add*:
fixes *a b c* :: *'n*::*finite* ⇒ - ^ *'n*
shows *det*((χ *i*. *if i = k then a i + b i else c i*)::*'a*::*comm-ring-1*^*'n*^*'n*) =
det((χ *i*. *if i = k then a i else c i*)::*'a*::*comm-ring-1*^*'n*^*'n*) +
det((χ *i*. *if i = k then b i else c i*)::*'a*::*comm-ring-1*^*'n*^*'n*)
unfolding *det-def Cart-lambda-beta setsum-addf*[*symmetric*]
proof (*rule setsum-cong2*)
let *?U = UNIV* :: *'n set*
let *?pU* = {*p*. *p permutes ?U*}
let *?f* = (λ*i*. *if i = k then a i + b i else c i*)::*'n* ⇒ *'a*::*comm-ring-1*^*'n*
let *?g* = (λ *i*. *if i = k then a i else c i*)::*'n* ⇒ *'a*::*comm-ring-1*^*'n*
let *?h* = (λ *i*. *if i = k then b i else c i*)::*'n* ⇒ *'a*::*comm-ring-1*^*'n*
fix *p* **assume** *p*: *p* ∈ *?pU*
let *?Uk = ?U* − {*k*}
from *p* **have** *pU*: *p permutes ?U* **by** *blast*
have *kU*: *?U = insert k ?Uk* **by** *blast*
{fix *j* **assume** *j*: *j* ∈ *?Uk*
from *j* **have** *?f j* $ *p j = ?g j* $ *p j* **and** *?f j* $ *p j*= *?h j* $ *p j*
by *simp-all***}**
then have *th1*: *setprod* (λ*i*. *?f i* $ *p i*) *?Uk* = *setprod* (λ*i*. *?g i* $ *p i*) *?Uk*
and *th2*: *setprod* (λ*i*. *?f i* $ *p i*) *?Uk* = *setprod* (λ*i*. *?h i* $ *p i*) *?Uk*
apply −
apply (*rule setprod-cong*, *simp-all*)+
done
have *th3*: *finite ?Uk k* ∉ *?Uk* **by** *auto*
have *setprod* (λ*i*. *?f i* $ *p i*) *?U = setprod* (λ*i*. *?f i* $ *p i*) (*insert k ?Uk*)
unfolding *kU*[*symmetric*] **..**
also have ... = *?f k* $ *p k* ∗ *setprod* (λ*i*. *?f i* $ *p i*) *?Uk*
apply (*rule setprod-insert*)
apply *simp*
by *blast*
also have ... = (*a k* $ *p k* ∗ *setprod* (λ*i*. *?f i* $ *p i*) *?Uk*) + (*b k*$ *p k* ∗ *setprod*
(λ*i*. *?f i* $ *p i*) *?Uk*) **by** (*simp add*: *ring-simps*)

also have ... = (*a k* $ *p k* * *setprod* (λ*i*. *?g i* $ *p i*) *?Uk*) + (*b k*$ *p k* * *setprod* (λ*i*. *?h i* $ *p i*) *?Uk*) **by** (*metis th1 th2*)
also have ... = *setprod* (λ*i*. *?g i* $ *p i*) (*insert k ?Uk*) + *setprod* (λ*i*. *?h i* $ *p i*) (*insert k ?Uk*)
unfolding *setprod-insert*[*OF th3*] **by** *simp*
finally have *setprod* (λ*i*. *?f i* $ *p i*) *?U* = *setprod* (λ*i*. *?g i* $ *p i*) *?U* + *setprod* (λ*i*. *?h i* $ *p i*) *?U* **unfolding** *kU*[*symmetric*] **.**
then show *of-int* (*sign p*) * *setprod* (λ*i*. *?f i* $ *p i*) *?U* = *of-int* (*sign p*) * *setprod* (λ*i*. *?g i* $ *p i*) *?U* + *of-int* (*sign p*) * *setprod* (λ*i*. *?h i* $ *p i*) *?U*
by (*simp add*: *ring-simps*)
qed

lemma *det-row-mul*:
fixes *a b* :: *'n*::*finite* ⇒ - ^ *'n*
shows *det*((χ *i*. *if i* = *k then c* *s *a i else b i*)::*'a*::*comm-ring-1*^*'n*^*'n*) =
*c** *det*((χ *i*. *if i* = *k then a i else b i*)::*'a*::*comm-ring-1*^*'n*^*'n*)

unfolding *det-def Cart-lambda-beta setsum-right-distrib*
proof (*rule setsum-cong2*)
let *?U* = *UNIV* :: *'n set*
let *?pU* = {*p*. *p permutes ?U*}
let *?f* = (λ*i*. *if i* = *k then c*s *a i else b i*)::*'n* ⇒ *'a*::*comm-ring-1*^*'n*
let *?g* = (λ *i*. *if i* = *k then a i else b i*)::*'n* ⇒ *'a*::*comm-ring-1*^*'n*
fix *p* **assume** *p*: *p* ∈ *?pU*
let *?Uk* = *?U* − {*k*}
from *p* **have** *pU*: *p permutes ?U* **by** *blast*
have *kU*: *?U* = *insert k ?Uk* **by** *blast*
{fix *j* **assume** *j*: *j* ∈ *?Uk*
from *j* **have** *?f j* $ *p j* = *?g j* $ *p j* **by** *simp***}**
then have *th1*: *setprod* (λ*i*. *?f i* $ *p i*) *?Uk* = *setprod* (λ*i*. *?g i* $ *p i*) *?Uk*
apply −
apply (*rule setprod-cong*, *simp-all*)
done
have *th3*: *finite ?Uk k* ∉ *?Uk* **by** *auto*
have *setprod* (λ*i*. *?f i* $ *p i*) *?U* = *setprod* (λ*i*. *?f i* $ *p i*) (*insert k ?Uk*)
unfolding *kU*[*symmetric*] **..**
also have ... = *?f k* $ *p k* * *setprod* (λ*i*. *?f i* $ *p i*) *?Uk*
apply (*rule setprod-insert*)
apply *simp*
by *blast*
also have ... = (*c*s a k*) $ *p k* * *setprod* (λ*i*. *?f i* $ *p i*) *?Uk* **by** (*simp add*: *ring-simps*)
also have ... = *c** (*a k* $ *p k* * *setprod* (λ*i*. *?g i* $ *p i*) *?Uk*)
unfolding *th1* **by** (*simp add*: *mult-ac*)
also have ... = *c** (*setprod* (λ*i*. *?g i* $ *p i*) (*insert k ?Uk*))
unfolding *setprod-insert*[*OF th3*] **by** *simp*
finally have *setprod* (λ*i*. *?f i* $ *p i*) *?U* = *c** (*setprod* (λ*i*. *?g i* $ *p i*) *?U*)
unfolding *kU*[*symmetric*] **.**
then show *of-int* (*sign p*) * *setprod* (λ*i*. *?f i* $ *p i*) *?U* = *c* * (*of-int* (*sign p*) *

```
setprod (λi. ?g i $ p i) ?U)
  by (simp add: ring-simps)
qed

lemma det-row-0:
  fixes b :: 'n::finite ⇒ - ^ 'n
  shows det((χ i. if i = k then 0 else b i)::'a::comm-ring-1^'n^'n) = 0
using det-row-mul[of k 0 λi. 1 b]
apply (simp)
  unfolding vector-smult-lzero .

lemma det-row-operation:
  fixes A :: 'a::ordered-idom^'n^'n::finite
  assumes ij: i ≠ j
  shows det (χ k. if k = i then row i A + c *s row j A else row k A) = det A
proof−
  let ?Z = (χ k. if k = i then row j A else row k A) :: 'a ^'n^'n
  have th: row i ?Z = row j ?Z by (vector row-def)
  have th2: ((χ k. if k = i then row i A else row k A) :: 'a^'n^'n) = A
    by (vector row-def)
  show ?thesis
    unfolding det-row-add [of i] det-row-mul[of i] det-identical-rows[OF ij th] th2
    by simp
qed

lemma det-row-span:
  fixes A :: 'a:: ordered-idom^'n^'n::finite
  assumes x: x ∈ span {row j A |j. j ≠ i}
  shows det (χ k. if k = i then row i A + x else row k A) = det A
proof−
  let ?U = UNIV :: 'n set
  let ?S = {row j A |j. j ≠ i}
  let ?d = λx. det (χ k. if k = i then x else row k A)
  let ?P = λx. ?d (row i A + x) = det A
  {fix k

    have (if k = i then row i A + 0 else row k A) = row k A by simp}
  then have P0: ?P 0
    apply −
    apply (rule cong[of det, OF refl])
    by (vector row-def)
  moreover
  {fix c z y assume zS: z ∈ ?S and Py: ?P y
    from zS obtain j where j: z = row j A i ≠ j by blast
    let ?w = row i A + y
    have th0: row i A + (c*s z + y) = ?w + c*s z by vector
    have thz: ?d z = 0
      apply (rule det-identical-rows[OF j(2)])
      using j by (vector row-def)
```

```
    have ?d (row i A + (c*s z + y)) = ?d (?w + c*s z) unfolding th0 ..
    then have ?P (c*s z + y) unfolding thz Py det-row-mul[of i] det-row-add[of
i]
      by simp }

  ultimately show ?thesis
    apply -
    apply (rule span-induct-alt[of ?P ?S, OF P0])
    apply blast
    apply (rule x)
    done
qed
```

```
lemma det-dependent-rows:
  fixes A:: 'a::ordered-idom^'n^'n::finite
  assumes d: dependent (rows A)
  shows det A = 0
proof-
  let ?U = UNIV :: 'n set
  from d obtain i where i: row i A ∈ span (rows A - {row i A})
    unfolding dependent-def rows-def by blast
  {fix j k assume jk: j ≠ k
    and c: row j A = row k A
    from det-identical-rows[OF jk c] have ?thesis .}
  moreover
  {assume H: ⋀ i j. i ≠ j ⟹ row i A ≠ row j A
    have th0: - row i A ∈ span {row j A|j. j ≠ i}
      apply (rule span-neg)
      apply (rule set-rev-mp)
      apply (rule i)
      apply (rule span-mono)
      using H i by (auto simp add: rows-def)
    from det-row-span[OF th0]
    have det A = det (χ k. if k = i then 0 *s 1 else row k A)
      unfolding right-minus vector-smult-lzero ..
    with det-row-mul[of i 0::'a λi. 1]
    have det A = 0 by simp}
  ultimately show ?thesis by blast
qed

lemma det-dependent-columns: assumes d: dependent(columns (A::'a::ordered-idom^'n^'n::finite))
shows det A = 0
by (metis d det-dependent-rows rows-transp det-transp)
```

lemma *Cart-lambda-cong*: $(\bigwedge x.\ f\ x = g\ x) \Longrightarrow (Cart\text{-}lambda\ f :: 'a\hat{}'n) = (Cart\text{-}lambda\ g :: 'a\hat{}'n)$
apply (*rule iffD1*[*OF Cart-lambda-unique*]) **by** *vector*

lemma *det-linear-row-setsum*:
assumes *fS*: *finite S*
shows $det\ ((\chi\ i.\ if\ i = k\ then\ setsum\ (a\ i)\ S\ else\ c\ i) :: 'a :: comm\text{-}ring\text{-}1\hat{}'n\hat{}'n :: finite)$
$= setsum\ (\lambda j.\ det\ ((\chi\ i.\ if\ i = k\ then\ a\ \ i\ j\ else\ c\ i) :: 'a\hat{}'n\hat{}'n))\ S$
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **apply** *simp* **unfolding** *setsum-empty det-row-0*[*of k*] **..**
next
case (*2 x F*)
then show *?case* **by** (*simp add*: *det-row-add cong del*: *if-weak-cong*)
qed

lemma *finite-bounded-functions*:
assumes *fS*: *finite S*
shows *finite* $\{f.\ (\forall i \in \{1..\ (k::nat)\}.\ f\ i \in S) \wedge (\forall i.\ i \notin \{1\ ..\ k\} \longrightarrow f\ i = i)\}$
proof(*induct k*)
case *0*
have *th*: $\{f.\ \forall i.\ f\ i = i\} = \{id\}$ **by** (*auto intro*: *ext*)
show *?case* **by** (*auto simp add*: *th*)
next
case (*Suc k*)
let *?f* $= \lambda(y::nat,g)\ i.\ if\ i = Suc\ k\ then\ y\ else\ g\ i$
let *?S* $= ?f\ `\ (S \times \{f.\ (\forall i \in \{1..k\}.\ f\ i \in S) \wedge (\forall i.\ i \notin \{1..k\} \longrightarrow f\ i = i)\})$
have *?S* $= \{f.\ (\forall i \in \{1..\ Suc\ k\}.\ f\ i \in S) \wedge (\forall i.\ i \notin \{1..\ Suc\ k\} \longrightarrow f\ i = i)\}$
apply (*auto simp add*: *image-iff*)
apply (*rule-tac x*=*x* (*Suc k*) **in** *bexI*)
apply (*rule-tac x* $= \lambda i.\ if\ i = Suc\ k\ then\ i\ else\ x\ i$ **in** *exI*)
apply (*auto intro*: *ext*)
done
with *finite-imageI*[*OF finite-cartesian-product*[*OF fS Suc.hyps*(*1*)], *of ?f*]
show *?case* **by** *metis*
qed

lemma *eq-id-iff*[*simp*]: $(\forall x.\ f\ x = x) = (f = id)$ **by** (*auto intro*: *ext*)

lemma *det-linear-rows-setsum-lemma*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
shows $det((\chi\ i.\ if\ i \in T\ then\ setsum\ (a\ i)\ S\ else\ c\ i) :: 'a :: comm\text{-}ring\text{-}1\hat{}'n\hat{}'n :: finite)$
$=$
$$setsum\ (\lambda f.\ det((\chi\ i.\ if\ i \in T\ then\ a\ i\ (f\ i)\ else\ c\ i) :: 'a\hat{}'n\hat{}'n))\ \{f.\ (\forall i \in T.\ f\ i \in S) \wedge (\forall i.\ i \notin T \longrightarrow f\ i = i)\}$$

using *fT*
proof(*induct T arbitrary*: *a c set*: *finite*)
case *empty*
have *th0*: $\bigwedge x\ y.\ (\chi\ i.\ \textit{if}\ i \in \{\}\ \textit{then}\ x\ i\ \textit{else}\ y\ i) = (\chi\ i.\ y\ i)$ **by** *vector*
from *empty.prems* **show** *?case* **unfolding** *th0* **by** *simp*
next
case (*insert z T a c*)
let *?F* = $\lambda T.\ \{f.\ (\forall i \in T.\ f\ i \in S) \wedge (\forall i.\ i \notin T \longrightarrow f\ i = i)\}$
let *?h* = $\lambda(y,g)\ i.\ \textit{if}\ i = z\ \textit{then}\ y\ \textit{else}\ g\ i$
let *?k* = $\lambda h.\ (h(z),(\lambda i.\ \textit{if}\ i = z\ \textit{then}\ i\ \textit{else}\ h\ i))$
let *?s* = $\lambda\ k\ a\ c\ f.\ \textit{det}((\chi\ i.\ \textit{if}\ i \in T\ \textit{then}\ a\ i\ (f\ i)\ \textit{else}\ c\ i)::'a\hat{}'n\hat{}'n)$
let *?c* = $\lambda i.\ \textit{if}\ i = z\ \textit{then}\ a\ i\ j\ \textit{else}\ c\ i$
have *thif*: $\bigwedge a\ b\ c\ d.\ (\textit{if}\ a \vee b\ \textit{then}\ c\ \textit{else}\ d) = (\textit{if}\ a\ \textit{then}\ c\ \textit{else if}\ b\ \textit{then}\ c\ \textit{else}\ d)$ **by** *simp*
have *thif2*: $\bigwedge a\ b\ c\ d\ e.\ (\textit{if}\ a\ \textit{then}\ b\ \textit{else if}\ c\ \textit{then}\ d\ \textit{else}\ e) =$
$(\textit{if}\ c\ \textit{then}\ (\textit{if}\ a\ \textit{then}\ b\ \textit{else}\ d)\ \textit{else}\ (\textit{if}\ a\ \textit{then}\ b\ \textit{else}\ e))$ **by** *simp*
from ‹$z \notin T$› **have** *nz*: $\bigwedge i.\ i \in T \Longrightarrow i = z \longleftrightarrow \textit{False}$ **by** *auto*
have *det* $(\chi\ i.\ \textit{if}\ i \in \textit{insert}\ z\ T\ \textit{then setsum}\ (a\ i)\ S\ \textit{else}\ c\ i) =$
$\textit{det}\ (\chi\ i.\ \textit{if}\ i = z\ \textit{then setsum}\ (a\ i)\ S$
$\textit{else if}\ i \in T\ \textit{then setsum}\ (a\ i)\ S\ \textit{else}\ c\ i)$
unfolding *insert-iff thif* **..**
also have ... = $(\sum j{\in}S.\ \textit{det}\ (\chi\ i.\ \textit{if}\ i \in T\ \textit{then setsum}\ (a\ i)\ S$
$\textit{else if}\ i = z\ \textit{then}\ a\ i\ j\ \textit{else}\ c\ i))$
unfolding *det-linear-row-setsum*[*OF fS*]
apply (*subst thif2*)
using *nz* **by** (*simp cong del*: *if-weak-cong cong add*: *if-cong*)
finally have *tha*:
det $(\chi\ i.\ \textit{if}\ i \in \textit{insert}\ z\ T\ \textit{then setsum}\ (a\ i)\ S\ \textit{else}\ c\ i) =$
$(\sum (j, f){\in}S \times\ ?F\ T.\ \textit{det}\ (\chi\ i.\ \textit{if}\ i \in T\ \textit{then}\ a\ i\ (f\ i)$
$\textit{else if}\ i = z\ \textit{then}\ a\ i\ j$
$\textit{else}\ c\ i))$
unfolding *insert.hyps* **unfolding** *setsum-cartesian-product* **by** *blast*
show *?case* **unfolding** *tha*
apply(*rule setsum-eq-general-reverses*[**where** *h*= *?h* **and** *k*= *?k*],
blast intro: *finite-cartesian-product fS finite*,
blast intro: *finite-cartesian-product fS finite*)
using ‹$z \notin T$›
apply (*auto intro*: *ext*)
apply (*rule cong*[*OF refl*[*of det*]])
by *vector*
qed

lemma *det-linear-rows-setsum*:
assumes *fS*: *finite* (*S*::'*n*::*finite set*)
shows *det* $(\chi\ i.\ \textit{setsum}\ (a\ i)\ S) = \textit{setsum}\ (\lambda f.\ \textit{det}\ (\chi\ i.\ a\ i\ (f\ i) :: 'a{::}\textit{comm-ring-1}\hat{}\ 'n\hat{}'n{::}\textit{finite}))\ \{f.\ \forall i.\ f\ i \in S\}$
proof−
have *th0*: $\bigwedge x\ y.\ ((\chi\ i.\ \textit{if}\ i \in (\textit{UNIV}::'n\ \textit{set})\ \textit{then}\ x\ i\ \textit{else}\ y\ i) :: 'a\hat{}'n\hat{}'n) = (\chi\ i.\ x\ i)$ **by** *vector*

from *det-linear-rows-setsum-lemma*[*OF fS*, *of UNIV* :: *'n set a*, *unfolded th0*, *OF finite*] **show** *?thesis* **by** *simp*
qed

lemma *matrix-mul-setsum-alt*:
fixes *A B* :: *'a*::*comm-ring-1^'n^'n*::*finite*
shows *A ** B* = (*χ i. setsum* (*λk. Aik *s B $ k*) (*UNIV* :: *'n set*))
by (*vector matrix-matrix-mult-def setsum-component*)

lemma *det-rows-mul*:
det((*χ i. c i *s a i*)::*'a*::*comm-ring-1^'n^'n*::*finite*) =
setprod (*λi. c i*) (*UNIV*:: *'n set*) * *det*((*χ i. a i*)::*'a^'n^'n*)
proof (*simp add*: *det-def setsum-right-distrib cong add*: *setprod-cong*, *rule setsum-cong2*)
let *?U* = *UNIV* :: *'n set*
let *?PU* = {*p*. *p permutes ?U*}
fix *p* **assume** *pU*: *p* ∈ *?PU*
let *?s* = *of-int* (*sign p*)
from *pU* **have** *p*: *p permutes ?U* **by** *blast*
have *setprod* (*λi. c i * a i $ p i*) *?U* = *setprod c ?U * setprod* (*λi. a i $ p i*) *?U*
unfolding *setprod-timesf* **..**
then show *?s * (*∏ *xa*∈*?U. c xa * a xa $ p xa*) =
*setprod c ?U * (?s* (*∏ *xa*∈*?U. a xa $ p xa*)) **by** (*simp add*: *ring-simps*)
qed

lemma *det-mul*:
fixes *A B* :: *'a*::*ordered-idom^'n^'n*::*finite*
shows *det* (*A ** B*) = *det A * det B*
proof−
let *?U* = *UNIV* :: *'n set*
let *?F* = {*f*. (∀ *i*∈ *?U. f i* ∈ *?U*) ∧ (∀ *i. i* ∉ *?U* ⟶ *f i* = *i*)}
let *?PU* = {*p*. *p permutes ?U*}
have *fU*: *finite ?U* **by** *simp*
have *fF*: *finite ?F* **by** (*rule finite*)
{fix *p* **assume** *p*: *p permutes ?U*

have *p* ∈ *?F* **unfolding** *mem-Collect-eq permutes-in-image*[*OF p*]
using *p*[*unfolded permutes-def*] **by** *simp*}
then have *PUF*: *?PU* ⊆ *?F* **by** *blast*
{fix *f* **assume** *fPU*: *f* ∈ *?F* − *?PU*
have *fUU*: *f* ' *?U* ⊆ *?U* **using** *fPU* **by** *auto*
from *fPU* **have** *f*: ∀ *i* ∈ *?U. f i* ∈ *?U*
∀ *i. i* ∉ *?U* ⟶ *f i* = *i* ¬(∀ *y*. ∃!*x. f x* = *y*) **unfolding** *permutes-def*
by *auto*

let *?A* = (*χ i. Aif i *s B$f i*) :: *'a^'n^'n*
let *?B* = (*χ i. B$f i*) :: *'a^'n^'n*
{assume *fni*: ¬ *inj-on f ?U*
then obtain *i j* **where** *ij*: *f i* = *f j i* ≠ *j*

unfolding *inj-on-def* **by** *blast*
from *ij*
have *rth*: *row i ?B = row j ?B* **by** (*vector row-def*)
from *det-identical-rows*[*OF ij*(*2*) *rth*]
have *det* (*χ i. Aif i ∗s B$f i*) = *0*
unfolding *det-rows-mul* **by** *simp*}
moreover
{**assume** *fi*: *inj-on f ?U*
from *f fi* **have** *fith*: ⋀*i j. f i = f j ⟹ i = j*
unfolding *inj-on-def* **by** *metis*
note *fs* = *fi*[*unfolded surjective-iff-injective-gen*[*OF fU fU refl fUU*, *symmetric*]]

{**fix** *y*
from *fs f* **have** ∃ *x. f x = y* **by** *blast*
then obtain *x* **where** *x*: *f x = y* **by** *blast*
{**fix** *z* **assume** *z*: *f z = y* **from** *fith x z* **have** *z = x* **by** *metis*}
with *x* **have** ∃!*x. f x = y* **by** *blast*}
with *f*(*3*) **have** *det* (*χ i. Aif i ∗s B$f i*) = *0* **by** *blast*}
ultimately have *det* (*χ i. Aif i ∗s B$f i*) = *0* **by** *blast*}
hence *zth*: ∀ *f*∈ *?F − ?PU. det* (*χ i. Aif i ∗s B$f i*) = *0* **by** *simp*
{**fix** *p* **assume** *pU*: *p* ∈ *?PU*
from *pU* **have** *p*: *p permutes ?U* **by** *blast*
let *?s* = *λp. of-int* (*sign p*)
let *?f* = *λq. ?s p ∗* (∏ *i*∈ *?U. A $ i $ p i*) ∗
(*?s q ∗* (∏ *i*∈ *?U. B $ i $ q i*))
have (*setsum* (*λq. ?s q ∗*
(∏ *i*∈ *?U.* (*χ i. A $ i $ p i ∗s B $ p i* :: *'a^'n^'n*) *$ i $ q i*)) *?PU*) =
(*setsum* (*λq. ?s p ∗* (∏ *i*∈ *?U. A $ i $ p i*) ∗
(*?s q ∗* (∏ *i*∈ *?U. B $ i $ q i*))) *?PU*)
unfolding *sum-permutations-compose-right*[*OF permutes-inv*[*OF p*], *of ?f*]
proof(*rule setsum-cong2*)
fix *q* **assume** *qU*: *q* ∈ *?PU*
hence *q*: *q permutes ?U* **by** *blast*
from *p q* **have** *pp*: *permutation p* **and** *pq*: *permutation q*
unfolding *permutation-permutes* **by** *auto*
have *th00*: *of-int* (*sign p*) ∗ *of-int* (*sign p*) = (*1*::*'a*)
⋀*a. of-int* (*sign p*) ∗ (*of-int* (*sign p*) ∗ *a*) = *a*
unfolding *mult-assoc*[*symmetric*] **unfolding** *of-int-mult*[*symmetric*]
by (*simp-all add*: *sign-idempotent*)
have *ths*: *?s q = ?s p ∗ ?s* (*q o inv p*)
using *pp pq permutation-inverse*[*OF pp*] *sign-inverse*[*OF pp*]
by (*simp add*: *th00 mult-ac sign-idempotent sign-compose*)
have *th001*: *setprod* (*λi. Bi q* (*inv p i*)) *?U = setprod* ((*λi. Bi q* (*inv p i*)) *o p*) *?U*
by (*rule setprod-permute*[*OF p*])
have *thp*: *setprod* (*λi.* (*χ i. Aip i ∗s B$p i* :: *'a^'n^'n*) *$i $ q i*) *?U = setprod* (*λi. Aip i*) *?U ∗ setprod* (*λi. Bi q* (*inv p i*)) *?U*
unfolding *th001 setprod-timesf*[*symmetric*] *o-def permutes-inverses*[*OF p*]
apply (*rule setprod-cong*[*OF refl*])

```
      using permutes-in-image[OF q] by vector
    show ?s q * setprod (λi. (((χ i. A$i$p i *s B$p i) :: 'a^'n^'n)$i$q i)) ?U =
?s p * (setprod (λi. A$i$p i) ?U) * (?s (q o inv p) * setprod (λi. B$i$(q o inv p)
i) ?U)
      using ths thp pp pq permutation-inverse[OF pp] sign-inverse[OF pp]
      by (simp add: sign-nz th00 ring-simps sign-idempotent sign-compose)
   qed
  }
  then have th2: setsum (λf. det (χ i. A$i$f i *s B$f i)) ?PU = det A * det B
   unfolding det-def setsum-product
   by (rule setsum-cong2)
  have det (A**B) = setsum (λf.  det (χ i. A $ i $ f i *s B $ f i)) ?F
   unfolding matrix-mul-setsum-alt det-linear-rows-setsum[OF fU] by simp
  also have ... = setsum (λf. det (χ i. A$i$f i *s B$f i)) ?PU
   using setsum-mono-zero-cong-left[OF fF PUF zth, symmetric]
   unfolding det-rows-mul by auto
  finally show ?thesis unfolding th2 .
qed
```

```
lemma invertible-left-inverse:
  fixes A :: real^'n^'n::finite
  shows invertible A ⟷ (∃(B::real^'n^'n). B** A = mat 1)
  by (metis invertible-def matrix-left-right-inverse)

lemma invertible-righ-inverse:
  fixes A :: real^'n^'n::finite
  shows invertible A ⟷ (∃(B::real^'n^'n). A** B = mat 1)
  by (metis invertible-def matrix-left-right-inverse)

lemma invertible-det-nz:
  fixes A::real ^'n^'n::finite
  shows invertible A ⟷ det A ≠ 0
proof−
  {assume invertible A
   then obtain B :: real ^'n^'n where B: A ** B = mat 1
    unfolding invertible-righ-inverse by blast
   hence det (A ** B) = det (mat 1 :: real ^'n^'n) by simp
   hence det A ≠ 0
    apply (simp add: det-mul det-I) by algebra }
  moreover
  {assume H: ¬ invertible A
   let ?U = UNIV :: 'n set
   have fU: finite ?U by simp
   from H obtain c i where c: setsum (λi. c i *s row i A) ?U = 0
    and iU: i ∈ ?U and ci: c i ≠ 0
```

```
    unfolding invertible-righ-inverse
    unfolding matrix-right-invertible-independent-rows by blast
  have stupid: ⋀(a::real^'n) b. a + b = 0 ⟹ −a = b
    apply (drule-tac f=op + (− a) in cong[OF refl])
    apply (simp only: ab-left-minus add-assoc[symmetric])
    apply simp
    done
  from c ci
  have thr0: − row i A = setsum (λj. (1/ c i) *s c j *s row j A) (?U − {i})
    unfolding setsum-diff1'[OF fU iU] setsum-cmul
    apply −
    apply (rule vector-mul-lcancel-imp[OF ci])
    apply (auto simp add: vector-smult-assoc vector-smult-rneg field-simps)
    unfolding stupid ..
  have thr: − row i A ∈ span {row j A| j. j ≠ i}
    unfolding thr0
    apply (rule span-setsum)
    apply simp
    apply (rule ballI)
    apply (rule span-mul)+
    apply (rule span-superset)
    apply auto
    done
  let ?B = (χ k. if k = i then 0 else row k A) :: real ^'n^'n
  have thrb: row i ?B = 0 using iU by (vector row-def)
  have det A = 0
    unfolding det-row-span[OF thr, symmetric] right-minus
    unfolding det-zero-row[OF thrb] ..}
 ultimately show ?thesis by blast
qed
```

```
lemma cramer-lemma-transp:
  fixes A:: 'a::ordered-idom^'n^'n::finite and x :: 'a ^'n::finite
  shows det ((χ i. if i = k then setsum (λi. x$i *s row i A) (UNIV::'n set)
                        else row i A)::'a^'n^'n) = x$k * det A
  (is ?lhs = ?rhs)
proof−
  let ?U = UNIV :: 'n set
  let ?Uk = ?U − {k}
  have U: ?U = insert k ?Uk by blast
  have fUk: finite ?Uk by simp
  have kUk: k ∉ ?Uk by simp
  have th00: ⋀k s. x$k *s row k A + s = (x$k − 1) *s row k A + row k A + s
    by (vector ring-simps)
  have th001: ⋀f k . (λx. if x = k then f k else f x) = f by (auto intro: ext)
```

have (χ *i. row i A*) = *A* **by** (*vector row-def*)
then have *thd1*: *det* (χ *i. row i A*) = *det A* **by** *simp*
have *thd0*: *det* (χ *i. if i = k then row k A* + ($\sum$ *i* ∈ *?Uk. x* $ *i* ∗*s row i A*) *else row i A*) = *det A*
apply (*rule det-row-span*)
apply (*rule span-setsum*[*OF fUk*])
apply (*rule ballI*)
apply (*rule span-mul*)
apply (*rule span-superset*)
apply *auto*
done
show *?lhs* = *x*$*k* ∗ *det A*
apply (*subst U*)
unfolding *setsum-insert*[*OF fUk kUk*]
apply (*subst th00*)
unfolding *add-assoc*
apply (*subst det-row-add*)
unfolding *thd0*
unfolding *det-row-mul*
unfolding *th001*[*of k* λ*i. row i A*]
unfolding *thd1* **by** (*simp add*: *ring-simps*)
qed

lemma *cramer-lemma*:
fixes *A* :: *'a*::*ordered-idom ^'n^'n*::*finite*
shows *det*((χ *i j. if j = k then* (*A* ∗*v x*)$*i else A*$*i*$*j*):: *'a^'n^'n*) = *x*$*k* ∗ *det A*
proof−
let *?U* = *UNIV* :: *'n set*
have *stupid*: ⋀*c. setsum* (λ*i. c i* ∗*s row i* (*transp A*)) *?U* = *setsum* (λ*i. c i* ∗*s column i A*) *?U*
by (*auto simp add*: *row-transp intro*: *setsum-cong2*)
show *?thesis* **unfolding** *matrix-mult-vsum*
unfolding *cramer-lemma-transp*[*of k x transp A*, *unfolded det-transp*, *symmetric*]
unfolding *stupid*[*of* λ*i. x*$*i*]
apply (*subst det-transp*[*symmetric*])
apply (*rule cong*[*OF refl*[*of det*]]) **by** (*vector transp-def column-def row-def*)
qed

lemma *cramer*:
fixes *A* ::*real^'n^'n*::*finite*
assumes *d0*: *det A* ≠ *0*
shows *A* ∗*v x* = *b* ⟷ *x* = (χ *k. det*(χ *i j. if j*=*k then b*$*i else A*$*i*$*j* :: *real^'n^'n*) / *det A*)
proof−
from *d0* **obtain** *B* **where** *B*: *A* ∗∗ *B* = *mat 1 B* ∗∗ *A* = *mat 1*
unfolding *invertible-det-nz*[*symmetric*] *invertible-def* **by** *blast*
have (*A* ∗∗ *B*) ∗*v b* = *b* **by** (*simp add*: *B matrix-vector-mul-lid*)
hence *A* ∗*v* (*B* ∗*v b*) = *b* **by** (*simp add*: *matrix-vector-mul-assoc*)
then have *xe*: ∃ *x. A*∗*v x* = *b* **by** *blast*

{fix x **assume** x: $A *v x = b$
have $x = (\chi\ k.\ det(\chi\ i\ j.\ if\ j{=}k\ then\ b\$i\ else\ A\$i\$j :: real\hat{}'n\hat{}'n)\ /\ det\ A)$
unfolding $x[symmetric]$
using *d0* **by** (*simp add*: *Cart-eq cramer-lemma field-simps*)**}**
with *xe* **show** *?thesis* **by** *auto*
qed

definition *orthogonal-transformation f* $\longleftrightarrow$ *linear f* $\wedge$ $(\forall\ v\ w.\ f\ v \cdot f\ w = v \cdot w)$

lemma *orthogonal-transformation*: *orthogonal-transformation f* $\longleftrightarrow$ *linear f* $\wedge$ $(\forall (v{::}real\ \hat{}\text{-}).\ norm\ (f\ v) = norm\ v)$
unfolding *orthogonal-transformation-def*
apply *auto*
apply (*erule-tac x=v* **in** *allE*)+
apply (*simp add*: *real-vector-norm-def*)
by (*simp add*: *dot-norm linear-add*[*symmetric*])

definition *orthogonal-matrix* $(Q{::}'a{::}semiring\text{-}1\hat{}'n\hat{}'n) \longleftrightarrow transp\ Q ** Q = mat\ 1 \wedge Q ** transp\ Q = mat\ 1$

lemma *orthogonal-matrix*: *orthogonal-matrix* $(Q{::}\ real\ \hat{}'n\hat{}'n{::}finite) \longleftrightarrow transp\ Q ** Q = mat\ 1$
by (*metis matrix-left-right-inverse orthogonal-matrix-def*)

lemma *orthogonal-matrix-id*: *orthogonal-matrix* $(mat\ 1 :: -\hat{}'n\hat{}'n{::}finite)$
by (*simp add*: *orthogonal-matrix-def transp-mat matrix-mul-lid*)

lemma *orthogonal-matrix-mul*:
fixes $A :: real\ \hat{}'n\hat{}'n{::}finite$
assumes *oA* : *orthogonal-matrix A*
and *oB*: *orthogonal-matrix B*
shows *orthogonal-matrix*$(A ** B)$
using *oA oB*
unfolding *orthogonal-matrix matrix-transp-mul*
apply (*subst matrix-mul-assoc*)
apply (*subst matrix-mul-assoc*[*symmetric*])
by (*simp add*: *matrix-mul-rid*)

lemma *orthogonal-transformation-matrix*:
fixes $f{::}\ real\hat{}'n \Rightarrow real\hat{}'n{::}finite$
shows *orthogonal-transformation f* $\longleftrightarrow$ *linear f* $\wedge$ *orthogonal-matrix*$(matrix\ f)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof−
let *?mf* = *matrix f*
let *?ot* = *orthogonal-transformation f*

let *?U = UNIV :: 'n set*
have *fU*: *finite ?U* **by** *simp*
let *?m1 = mat 1 :: real ^'n^'n*
{assume *ot*: *?ot*
from *ot* **have** *lf*: *linear f* **and** *fd*: $\forall v\ w.\ f\ v \cdot f\ w = v \cdot w$
unfolding *orthogonal-transformation-def orthogonal-matrix* **by** *blast+*
{fix *i j*
let *?A = transp ?mf ** ?mf*
have *th0*: $\bigwedge b\ (x::'a::comm\text{-}ring\text{-}1).\ (if\ b\ then\ 1\ else\ 0)*x = (if\ b\ then\ x\ else\ 0)$
$\bigwedge b\ (x::'a::comm\text{-}ring\text{-}1).\ x*(if\ b\ then\ 1\ else\ 0) = (if\ b\ then\ x\ else\ 0)$
by *simp-all*
from *fd*[*rule-format*, *of basis i basis j*, *unfolded matrix-works*[*OF lf*, *symmetric*] *dot-matrix-vector-mul*]
have *?Aij = ?m1 $ i $ j*
by (*simp add*: *dot-def matrix-matrix-mult-def columnvector-def rowvector-def basis-def th0 setsum-delta*[*OF fU*] *mat-def*)**}**
hence *orthogonal-matrix ?mf* **unfolding** *orthogonal-matrix* **by** *vector*
with *lf* **have** *?rhs* **by** *blast***}**
moreover
{assume *lf*: *linear f* **and** *om*: *orthogonal-matrix ?mf*
from *lf om* **have** *?lhs*
unfolding *orthogonal-matrix-def norm-eq orthogonal-transformation*
unfolding *matrix-works*[*OF lf*, *symmetric*]
apply (*subst dot-matrix-vector-mul*)
by (*simp add*: *dot-matrix-product matrix-mul-lid*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *det-orthogonal-matrix*:
fixes *Q*:: *'a::ordered-idom^'n^'n::finite*
assumes *oQ*: *orthogonal-matrix Q*
shows *det Q = 1 ∨ det Q = − 1*
proof−

have *th*: $\bigwedge x::'a.\ x = 1 \vee x = -1 \longleftrightarrow x*x = 1$ (**is** $\bigwedge x::'a.\ ?ths\ x$)
proof−
fix *x*:: *'a*
have *th0*: *x*x − 1 = (x − 1)*(x + 1)* **by** (*simp add*: *ring-simps*)
have *th1*: $\bigwedge (x::'a)\ y.\ x = -y \longleftrightarrow x + y = 0$
apply (*subst eq-iff-diff-eq-0*) **by** *simp*
have *x*x = 1 ⟷ x*x − 1 = 0* **by** *simp*
also have *... ⟷ x = 1 ∨ x = − 1* **unfolding** *th0 th1* **by** *simp*
finally show *?ths x* **..**
qed
from *oQ* **have** *Q ** transp Q = mat 1* **by** (*metis orthogonal-matrix-def*)
hence *det (Q ** transp Q) = det (mat 1:: 'a^'n^'n)* **by** *simp*
hence *det Q * det Q = 1* **by** (*simp add*: *det-mul det-I det-transp*)
then show *?thesis* **unfolding** *th* **.**

qed

lemma *scaling-linear*:
fixes *f* :: *real ^'n ⇒ real ^'n*::*finite*
assumes *f0*: *f 0 = 0* **and** *fd*: ∀ *x y*. *dist* (*f x*) (*f y*) = *c* ∗ *dist x y*
shows *linear f*
proof−
{**fix** *v w*
{**fix** *x* **note** *fd*[*rule-format*, *of x 0*, *unfolded dist-def f0 diff-0-right*] }
note *th0* = *this*
have *f v* · *f w* = *c^2* ∗ (*v* · *w*)
unfolding *dot-norm-neg dist-def*[*symmetric*]
unfolding *th0 fd*[*rule-format*] **by** (*simp add*: *power2-eq-square field-simps*)}
note *fc* = *this*
show *?thesis* **unfolding** *linear-def vector-eq*
by (*simp add*: *dot-lmult dot-ladd dot-rmult dot-radd fc ring-simps*)
qed

lemma *isometry-linear*:
f (*0*:: *real^'n*) = (*0*:: *real^'n*::*finite*) ⟹ ∀ *x y*. *dist*(*f x*) (*f y*) = *dist x y*
⟹ *linear f*
by (*rule scaling-linear*[**where** *c=1*]) *simp-all*

lemma *orthogonal-transformation-isometry*:
orthogonal-transformation f ⟷ *f*(*0*::*real^'n*) = (*0*::*real^'n*::*finite*) ∧ (∀ *x y*. *dist*(*f x*) (*f y*) = *dist x y*)
unfolding *orthogonal-transformation*
apply (*rule iffI*)
apply *clarify*
apply (*clarsimp simp add*: *linear-0 linear-sub*[*symmetric*] *dist-def*)
apply (*rule conjI*)
apply (*rule isometry-linear*)
apply *simp*
apply *simp*
apply *clarify*
apply (*erule-tac x=v* **in** *allE*)
apply (*erule-tac x=0* **in** *allE*)
by (*simp add*: *dist-def*)

lemma *isometry-sphere-extend*:
fixes f:: *real* ^'*n* ⇒ *real* ^'*n*::*finite*
assumes *f1*: ∀ *x*. *norm x* = *1* ⟶ *norm* (*f x*) = *1*
and *fd1*: ∀ *x y*. *norm x* = *1* ⟶ *norm y* = *1* ⟶ *dist* (*f x*) (*f y*) = *dist x y*
shows ∃ *g*. *orthogonal-transformation g* ∧ (∀ *x*. *norm x* = *1* ⟶ *g x* = *f x*)
proof−
{fix *x y x' y' x0 y0 x0' y0'* :: *real* ^'*n*
assume *H*: *x* = *norm x* ∗*s x0 y* = *norm y* ∗*s y0*
x' = *norm x* ∗*s x0' y'* = *norm y* ∗*s y0'*
norm x0 = *1 norm x0'* = *1 norm y0* = *1 norm y0'* = *1*
norm(*x0'* − *y0'*) = *norm*(*x0* − *y0*)

have *norm*(*x'* − *y'*) = *norm*(*x* − *y*)
apply (*subst H*(*1*))
apply (*subst H*(*2*))
apply (*subst H*(*3*))
apply (*subst H*(*4*))
using *H*(*5*−*9*)
apply (*simp add*: *norm-eq norm-eq-1*)
apply (*simp add*: *dot-lsub dot-rsub dot-lmult dot-rmult*)
apply (*simp add*: *ring-simps*)
by (*simp only*: *right-distrib*[*symmetric*])**}**
note *th0* = *this*
let *?g* = λ*x*. *if x* = *0 then 0 else norm x* ∗*s f* (*inverse* (*norm x*) ∗*s x*)
{fix *x*:: *real* ^'*n* **assume** *nx*: *norm x* = *1*
have *?g x* = *f x* **using** *nx* **by** *auto***}**
hence *thfg*: ∀ *x*. *norm x* = *1* ⟶ *?g x* = *f x* **by** *blast*
have *g0*: *?g 0* = *0* **by** *simp*
{fix *x y* :: *real* ^'*n*
{assume *x* = *0 y* = *0*
then have *dist* (*?g x*) (*?g y*) = *dist x y* **by** *simp* **}**
moreover
{assume *x* = *0 y* ≠ *0*
then have *dist* (*?g x*) (*?g y*) = *dist x y*
apply (*simp add*: *dist-def norm-mul*)
apply (*rule f1*[*rule-format*])
by(*simp add*: *norm-mul field-simps*)**}**
moreover
{assume *x* ≠ *0 y* = *0*
then have *dist* (*?g x*) (*?g y*) = *dist x y*
apply (*simp add*: *dist-def norm-mul*)
apply (*rule f1*[*rule-format*])
by(*simp add*: *norm-mul field-simps*)**}**
moreover
{assume *z*: *x* ≠ *0 y* ≠ *0*
have *th00*: *x* = *norm x* ∗*s inverse* (*norm x*) ∗*s x y* = *norm y* ∗*s inverse* (*norm y*) ∗*s y norm x* ∗*s f* (*inverse* (*norm x*) ∗*s x*) = *norm x* ∗*s f* (*inverse* (*norm x*) ∗*s x*)

```
      norm y *s f (inverse (norm y) *s y) = norm y *s f (inverse (norm y) *s y)
       norm (inverse (norm x) *s x) = 1
       norm (f (inverse (norm x) *s x)) = 1
       norm (inverse (norm y) *s y) = 1
       norm (f (inverse (norm y) *s y)) = 1
       norm (f (inverse (norm x) *s x) - f (inverse (norm y) *s y)) =
       norm (inverse (norm x) *s x - inverse (norm y) *s y)
       using z
     by (auto simp add: vector-smult-assoc field-simps norm-mul intro: f1[rule-format]
fd1[rule-format, unfolded dist-def])
     from z th0[OF th00] have dist (?g x) (?g y) = dist x y
       by (simp add: dist-def)}
   ultimately have dist (?g x) (?g y) = dist x y by blast}
 note thd = this
   show ?thesis
   apply (rule exI[where x= ?g])
   unfolding orthogonal-transformation-isometry
    using  g0 thfg thd by metis
qed
```

```
definition rotation-matrix Q ⟷ orthogonal-matrix Q ∧ det Q = 1
definition rotoinversion-matrix Q ⟷ orthogonal-matrix Q ∧ det Q = - 1

lemma orthogonal-rotation-or-rotoinversion:
  fixes Q :: 'a::ordered-idom^'n^'n::finite
  shows  orthogonal-matrix Q ⟷ rotation-matrix Q ∨ rotoinversion-matrix Q
  by (metis rotoinversion-matrix-def rotation-matrix-def det-orthogonal-matrix)
```

```
lemma setprod-1: setprod f {(1::nat)..1} = f 1 by simp

lemma setprod-2: setprod f {(1::nat)..2} = f 1 * f 2
  by (simp add: nat-number setprod-numseg mult-commute)
lemma setprod-3: setprod f {(1::nat)..3} = f 1 * f 2 * f 3
  by (simp add: nat-number setprod-numseg mult-commute)

lemma det-1: det (A::'a::comm-ring-1^1^1) = A$1$1
  by (simp add: det-def permutes-sing sign-id UNIV-1)

lemma det-2: det (A::'a::comm-ring-1^2^2) = A$1$1 * A$2$2 - A$1$2 * A$2$1
proof-
  have f12: finite {2::2} 1 ∉ {2::2} by auto
  show ?thesis
```

unfolding *det-def UNIV-2*
unfolding *setsum-over-permutations-insert*[*OF f12*]
unfolding *permutes-sing*
apply (*simp add*: *sign-swap-id sign-id swap-id-eq*)
by (*simp add*: *arith-simps*(*31*)[*symmetric*] *of-int-minus of-int-1 del*: *arith-simps*(*31*))
qed

lemma *det-3*: *det* (*A*::'*a*::*comm-ring-1*^*3*^*3*) =
A$1$1 ∗ *A$2$2* ∗ *A$3$3* +
A$1$2 ∗ *A$2$3* ∗ *A$3$1* +
A$1$3 ∗ *A$2$1* ∗ *A$3$2* −
A$1$1 ∗ *A$2$3* ∗ *A$3$2* −
A$1$2 ∗ *A$2$1* ∗ *A$3$3* −
A$1$3 ∗ *A$2$2* ∗ *A$3$1*
proof−
have *f123*: *finite* {*2*::*3*, *3*} *1* ∉ {*2*::*3*, *3*} **by** *auto*
have *f23*: *finite* {*3*::*3*} *2* ∉ {*3*::*3*} **by** *auto*

show *?thesis*
unfolding *det-def UNIV-3*
unfolding *setsum-over-permutations-insert*[*OF f123*]
unfolding *setsum-over-permutations-insert*[*OF f23*]

unfolding *permutes-sing*
apply (*simp add*: *sign-swap-id permutation-swap-id sign-compose sign-id swap-id-eq*)
apply (*simp add*: *arith-simps*(*31*)[*symmetric*] *of-int-minus of-int-1 del*: *arith-simps*(*31*))
by (*simp add*: *ring-simps*)
qed

end

33 Diagonalize: A constructive version of Cantor's first diagonalization argument.

theory *Diagonalize*
imports *Main*
begin

33.1 Summation from *0* to *n*

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

lemma *sum-0*:
sum 0 = *0*
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* $(n + r) + Suc\ (n + r) \leq sum\ n + q$ **unfolding** *sum-Suc* **by** *simp*
with ‹$m = Suc\ (n + r)$› **have** $sum\ (n + r) + m \leq sum\ n + q$ **by** *simp*
have $sum\ n \leq sum\ (n + r)$ **unfolding** *strict-mono-less-eq* [*OF sum-strict-mono*] **by** *simp*
moreover from ‹$q \leq n$› ‹$n < m$› **have** $q < m$ **by** *simp*
ultimately have $sum\ n + q < sum\ (n + r) + m$ **by** *auto*
with ‹$sum\ (n + r) + m \leq sum\ n + q$› **show** *False*
by *auto*
qed

lemma *sum-rest*:
assumes $q \leq n$
shows $sum\ m \leq sum\ n + q \longleftrightarrow m \leq n$
using *assms* **apply** (*auto intro*: *sum-rest-aux*)
apply (*simp add*: *strict-mono-less-eq* [*OF sum-strict-mono*, *symmetric*, *of m n*])
done

33.2 Diagonalization: an injective embedding of two *nat*s to one *nat*

definition *diagonalize* :: $nat \Rightarrow nat \Rightarrow nat$ **where**
$diagonalize\ m\ n = sum\ (m + n) + m$

lemma *diagonalize-inject*:
assumes $diagonalize\ a\ b = diagonalize\ c\ d$
shows $a = c$ **and** $b = d$
proof −
from *assms* **have** *diageq*: $sum\ (a + b) + a = sum\ (c + d) + c$
by (*simp add*: *diagonalize-def*)
have $a + b = c + d \vee a + b \geq Suc\ (c + d) \vee c + d \geq Suc\ (a + b)$ **by** *arith*
then have $a = c \wedge b = d$
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 $a + b \geq Suc\ (c + d)$
with *strict-mono-less-eq* [*OF sum-strict-mono*]
have $sum\ (a + b) \geq sum\ (Suc\ (c + d))$ **by** *simp*
with *diageq* **show** *?thesis* **by** (*simp add*: *sum-Suc*)
next
assume $c + d \geq Suc\ (a + b)$
with *strict-mono-less-eq* [*OF sum-strict-mono*]
have $sum\ (c + d) \geq sum\ (Suc\ (a + b))$ **by** *simp*
with *diageq* **show** *?thesis* **by** (*simp add*: *sum-Suc*)
qed
then show $a = c$ **and** $b = d$ **by** *auto*

qed

33.3 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

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

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

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

this, just include this theory.

34.1 Basic arithmetic

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

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

lemma *zero-nat-code* [*code*, *code inline*]:
0 = (*Numeral0* :: *nat*)
by *simp*
lemmas [*code post*] = *zero-nat-code* [*symmetric*]

lemma *one-nat-code* [*code*, *code inline*]:
1 = (*Numeral1* :: *nat*)
by *simp*
lemmas [*code post*] = *one-nat-code* [*symmetric*]

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* = *Divides.divmod*

lemma [*code*]:
Divides.divmod n m = (*if m* = *0 then* (*0*, *n*) *else divmod-aux n m*)
unfolding *divmod-aux-def divmod-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-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* ⟷ (*of-nat n* :: *int*) < *of-nat m*
by *simp*

34.2 Case analysis

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

lemma [*code*, *code unfold*]:
nat-case = (λ*f g n*. *if n* = *0 then f else g* (*n* − *1*))
by (*auto simp add*: *expand-fun-eq dest*!: *gr0-implies-Suc*)

34.3 Preprocessors

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

lemma *Suc-if-eq'*: (⋀*n*. *f* (*Suc n*) = *h n*) ⟹ *f 0* = *g* ⟹
f n = (*if n* = *0 then g else h* (*n* − *1*))
by (*cases n*) *simp-all*

lemma *Suc-if-eq*: (⋀*n*. *f* (*Suc n*) ≡ *h n*) ⟹ *f 0* ≡ *g* ⟹
f n ≡ *if n* = *0 then g else h* (*n* − *1*)
by (*rule eq-reflection*, *rule Suc-if-eq'*)
(*rule meta-eq-to-obj-eq*, *assumption*,
rule meta-eq-to-obj-eq, *assumption*)

lemma *Suc-clause*: (⋀*n*. *P n* (*Suc n*)) ⟹ *n* ≠ *0* ⟹ *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.

34.4 Target language setup

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

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

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

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

code-include *Haskell Nat* ⟪
newtype Nat = Nat Integer deriving (*Show*, *Eq*);

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

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

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

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

instance Integral Nat where {

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

code-reserved Haskell Nat

code-type nat
  (Haskell Nat.Nat)

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

Natural numerals.

```
lemma [code inline, symmetric, code post]:
  nat (number-of i) = number-nat-inst.number-of-nat i
  — this interacts as desired with number-of ?v = nat (number-of ?v)
  by (simp add: number-nat-inst.number-of-nat)

setup ⟨⟨
  fold (Numeral.add-code @{const-name number-nat-inst.number-of-nat}
    true false) [SML, OCaml, Haskell]
⟩⟩
```

Since natural numbers are implemented using integers in ML, the coercion function *of-nat* of type $nat \Rightarrow int$ 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]:
  of-nat = int by (simp add: int-def)
declare of-nat-int [symmetric, code post]

code-const int
  (SML -)
```

(*OCaml* -)

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

code-const *nat*
(*SML IntInf.max*/ (/*0*,/ -))
(*OCaml Big'-int.max'-big'-int*/ *Big'-int.zero'-big'-int*)

For Haskell, things are slightly different again.

code-const *int* **and** *nat*
(*Haskell toInteger* **and** *fromInteger*)

Conversion from and to indices.

code-const *Code-Index.of-nat*
(*SML IntInf.toInt*)
(*OCaml Big'-int.int'-of'-big'-int*)
(*Haskell fromEnum*)

code-const *Code-Index.nat-of*
(*SML IntInf.fromInt*)
(*OCaml Big'-int.big'-int'-of'-int*)
(*Haskell toEnum*)

Using target language arithmetic operations whenever appropriate

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

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

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

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

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

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

consts-code
0::*nat* (*0*)
1::*nat* (*1*)
Suc ((- +/ *1*))
op + :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* ((- +/ -))
op * :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *nat* ((- */ -))
op $\leq$ :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* ((- <=/ -))
op < :: *nat* $\Rightarrow$ *nat* $\Rightarrow$ *bool* ((- </ -))

Evaluation

lemma [*code*, *code del*]:
(*Code-Eval.term-of* :: *nat* $\Rightarrow$ *term*) = *Code-Eval.term-of* **..**

code-const *Code-Eval.term-of* :: *nat* $\Rightarrow$ *term*
(*SML HOLogic.mk'-number*/ *HOLogic.natT*)

Module names

code-modulename *SML*
Nat Integer
Divides Integer
Ring-and-Field Integer
Efficient-Nat Integer

code-modulename *OCaml*
Nat Integer
Divides Integer
Ring-and-Field Integer
Efficient-Nat Integer

code-modulename *Haskell*
Nat Integer
Divides Integer
Ring-and-Field Integer
Efficient-Nat Integer

hide *const int*

end

35 Enum: Finite types as explicit enumerations

theory *Enum*
imports *Map Main*
begin

35.1 Class *enum*

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

subclass *finite* **proof**
qed (*simp add*: *UNIV-enum*)

lemma *enum-all*: *set enum* = *UNIV* **unfolding** *UNIV-enum* **..**

lemma *in-enum* [*intro*]: $x \in$ *set enum*
  unfolding *enum-all* **by** *auto*

lemma *enum-eq-I*:
  assumes $\bigwedge x.\ x \in$ *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

35.2 Equality and order on functions

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

definition
  eq-class.eq f g $\longleftrightarrow$ $(\forall x \in$ *set enum*. *f x* = *g x*$)$

instance by *default*
  (*simp-all add*: *eq-fun-def enum-all expand-fun-eq*)

end

lemma *order-fun* [*code*]:
  fixes *f g* :: *'a*::*enum* $\Rightarrow$ *'b*::*order*
  shows *f* $\leq$ *g* $\longleftrightarrow$ *list-all* $(\lambda x.\ f\ x \leq g\ x)$ *enum*
    and *f* $<$ *g* $\longleftrightarrow$ *f* $\leq$ *g* $\wedge \neg$ *list-all* $(\lambda x.\ f\ x = g\ x)$ *enum*
  by (*simp-all add*: *list-all-iff enum-all expand-fun-eq le-fun-def order-less-le*)

35.3 Quantifiers

lemma *all-code* [*code*]: $(\forall x.\ P\ x) \longleftrightarrow$ *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*)

35.4 Default instances

primrec *n-lists* :: *nat* $\Rightarrow$ *'a list* $\Rightarrow$ *'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 map-compose* [*symmetric*] *o-def listsum-triv*)

lemma *length-n-lists-elem*: *ys* $\in$ *set* (*n-lists n xs*) $\Longrightarrow$ *length ys* = *n*
by (*induct n arbitrary*: *ys*) *auto*

lemma *set-n-lists*: *set* (*n-lists n xs*) = {*ys*. *length ys* = *n* $\wedge$ *set ys* $\subseteq$ *set xs*}
proof (*rule set-ext*)
fix *ys* :: *'a list*
show *ys* $\in$ *set* (*n-lists n xs*) $\longleftrightarrow$ *ys* $\in$ {*ys*. *length ys* = *n* $\wedge$ *set ys* $\subseteq$ *set xs*}
proof −
have *ys* $\in$ *set* (*n-lists n xs*) $\Longrightarrow$ *length ys* = *n*
by (*induct n arbitrary*: *ys*) *auto*
moreover have $\bigwedge$*x*. *ys* $\in$ *set* (*n-lists n xs*) $\Longrightarrow$ *x* $\in$ *set ys* $\Longrightarrow$ *x* $\in$ *set xs*
by (*induct n arbitrary*: *ys*) *auto*
moreover have *set ys* $\subseteq$ *set xs* $\Longrightarrow$ *ys* $\in$ *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* ($\bigcup$ *ys*$\in$*set* (*n-lists n xs*). (λ*y*. *y* # *ys*) ' *set xs*)

```
    = (∑ ys∈set (n-lists n xs). card ((λy. y # ys) ` set xs))
     by (rule card-UN-disjoint) auto
   moreover have ⋀ys. card ((λy. 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-map:
  fixes f :: 'a::enum ⇒ 'b::enum
  shows map-of (zip xs (map f xs)) = (λx. if x ∈ set xs then Some (f x) else
None)
  by (induct xs) (simp-all add: expand-fun-eq)

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* (λ*ys*. *the* ∘ *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 (λ*ys*. *the* ∘ *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 by *default*
(*simp-all add*: *enum-unit-def UNIV-unit*)

end

instantiation *bool* :: *enum*
begin

definition
enum = [*False*, *True*]

instance by *default*

(*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*)
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 by *default*
(*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-char* [*code*]:
enum = [*Char Nibble0 Nibble0*, *Char Nibble0 Nibble1*, *Char Nibble0 Nibble2*,
Char Nibble0 Nibble3, *Char Nibble0 Nibble4*, *Char Nibble0 Nibble5*,
Char Nibble0 Nibble6, *Char Nibble0 Nibble7*, *Char Nibble0 Nibble8*,
Char Nibble0 Nibble9, *Char Nibble0 NibbleA*, *Char Nibble0 NibbleB*,
Char Nibble0 NibbleC, *Char Nibble0 NibbleD*, *Char Nibble0 NibbleE*,
Char Nibble0 NibbleF, *Char Nibble1 Nibble0*, *Char Nibble1 Nibble1*,
Char Nibble1 Nibble2, *Char Nibble1 Nibble3*, *Char Nibble1 Nibble4*,

Char Nibble1 Nibble5, *Char Nibble1 Nibble6*, *Char Nibble1 Nibble7*,
Char Nibble1 Nibble8, *Char Nibble1 Nibble9*, *Char Nibble1 NibbleA*,
Char Nibble1 NibbleB, *Char Nibble1 NibbleC*, *Char Nibble1 NibbleD*,
Char Nibble1 NibbleE, *Char Nibble1 NibbleF*, *CHR " "*, *CHR "!"*,
Char Nibble2 Nibble2, *CHR "#"*, *CHR "$"*, *CHR "%"*, *CHR "&"*,
Char Nibble2 Nibble7, *CHR "("*, *CHR ")"*, *CHR "*"*, *CHR "+"*, *CHR ","*,
CHR "−", *CHR "."*, *CHR "/"*, *CHR "0"*, *CHR "1"*, *CHR "2"*, *CHR "3"*,
CHR "4", *CHR "5"*, *CHR "6"*, *CHR "7"*, *CHR "8"*, *CHR "9"*, *CHR ":"*,
CHR ";", *CHR "<"*, *CHR "="*, *CHR ">"*, *CHR "?"*, *CHR "@"*, *CHR "A"*,
CHR "B", *CHR "C"*, *CHR "D"*, *CHR "E"*, *CHR "F"*, *CHR "G"*, *CHR "H"*,
CHR "I", *CHR "J"*, *CHR "K"*, *CHR "L"*, *CHR "M"*, *CHR "N"*, *CHR "O"*,
CHR "P", *CHR "Q"*, *CHR "R"*, *CHR "S"*, *CHR "T"*, *CHR "U"*, *CHR "V"*,
CHR "W", *CHR "X"*, *CHR "Y"*, *CHR "Z"*, *CHR "["*, *Char Nibble5 NibbleC*,
CHR "]", *CHR "^"*, *CHR "-"*, *Char Nibble6 Nibble0*, *CHR "a"*, *CHR "b"*,
CHR "c", *CHR "d"*, *CHR "e"*, *CHR "f"*, *CHR "g"*, *CHR "h"*, *CHR "i"*,
CHR "j", *CHR "k"*, *CHR "l"*, *CHR "m"*, *CHR "n"*, *CHR "o"*, *CHR "p"*,
CHR "q", *CHR "r"*, *CHR "s"*, *CHR "t"*, *CHR "u"*, *CHR "v"*, *CHR "w"*,
CHR "x", *CHR "y"*, *CHR "z"*, *CHR "{"*, *CHR "|"*, *CHR "}"*, *CHR "~"*,
Char Nibble7 NibbleF, *Char Nibble8 Nibble0*, *Char Nibble8 Nibble1*,
Char Nibble8 Nibble2, *Char Nibble8 Nibble3*, *Char Nibble8 Nibble4*,
Char Nibble8 Nibble5, *Char Nibble8 Nibble6*, *Char Nibble8 Nibble7*,
Char Nibble8 Nibble8, *Char Nibble8 Nibble9*, *Char Nibble8 NibbleA*,
Char Nibble8 NibbleB, *Char Nibble8 NibbleC*, *Char Nibble8 NibbleD*,
Char Nibble8 NibbleE, *Char Nibble8 NibbleF*, *Char Nibble9 Nibble0*,
Char Nibble9 Nibble1, *Char Nibble9 Nibble2*, *Char Nibble9 Nibble3*,
Char Nibble9 Nibble4, *Char Nibble9 Nibble5*, *Char Nibble9 Nibble6*,
Char Nibble9 Nibble7, *Char Nibble9 Nibble8*, *Char Nibble9 Nibble9*,
Char Nibble9 NibbleA, *Char Nibble9 NibbleB*, *Char Nibble9 NibbleC*,
Char Nibble9 NibbleD, *Char Nibble9 NibbleE*, *Char Nibble9 NibbleF*,
Char NibbleA Nibble0, *Char NibbleA Nibble1*, *Char NibbleA Nibble2*,
Char NibbleA Nibble3, *Char NibbleA Nibble4*, *Char NibbleA Nibble5*,
Char NibbleA Nibble6, *Char NibbleA Nibble7*, *Char NibbleA Nibble8*,
Char NibbleA Nibble9, *Char NibbleA NibbleA*, *Char NibbleA NibbleB*,
Char NibbleA NibbleC, *Char NibbleA NibbleD*, *Char NibbleA NibbleE*,
Char NibbleA NibbleF, *Char NibbleB Nibble0*, *Char NibbleB Nibble1*,
Char NibbleB Nibble2, *Char NibbleB Nibble3*, *Char NibbleB Nibble4*,
Char NibbleB Nibble5, *Char NibbleB Nibble6*, *Char NibbleB Nibble7*,
Char NibbleB Nibble8, *Char NibbleB Nibble9*, *Char NibbleB NibbleA*,
Char NibbleB NibbleB, *Char NibbleB NibbleC*, *Char NibbleB NibbleD*,
Char NibbleB NibbleE, *Char NibbleB NibbleF*, *Char NibbleC Nibble0*,
Char NibbleC Nibble1, *Char NibbleC Nibble2*, *Char NibbleC Nibble3*,
Char NibbleC Nibble4, *Char NibbleC Nibble5*, *Char NibbleC Nibble6*,
Char NibbleC Nibble7, *Char NibbleC Nibble8*, *Char NibbleC Nibble9*,
Char NibbleC NibbleA, *Char NibbleC NibbleB*, *Char NibbleC NibbleC*,
Char NibbleC NibbleD, *Char NibbleC NibbleE*, *Char NibbleC NibbleF*,
Char NibbleD Nibble0, *Char NibbleD Nibble1*, *Char NibbleD Nibble2*,
Char NibbleD Nibble3, *Char NibbleD Nibble4*, *Char NibbleD Nibble5*,
Char NibbleD Nibble6, *Char NibbleD Nibble7*, *Char NibbleD Nibble8*,
Char NibbleD Nibble9, *Char NibbleD NibbleA*, *Char NibbleD NibbleB*,

Char NibbleD NibbleC, *Char NibbleD NibbleD*, *Char NibbleD NibbleE*,
Char NibbleD NibbleF, *Char NibbleE Nibble0*, *Char NibbleE Nibble1*,
Char NibbleE Nibble2, *Char NibbleE Nibble3*, *Char NibbleE Nibble4*,
Char NibbleE Nibble5, *Char NibbleE Nibble6*, *Char NibbleE Nibble7*,
Char NibbleE Nibble8, *Char NibbleE Nibble9*, *Char NibbleE NibbleA*,
Char NibbleE NibbleB, *Char NibbleE NibbleC*, *Char NibbleE NibbleD*,
Char NibbleE NibbleE, *Char NibbleE NibbleF*, *Char NibbleF Nibble0*,
Char NibbleF Nibble1, *Char NibbleF Nibble2*, *Char NibbleF Nibble3*,
Char NibbleF Nibble4, *Char NibbleF Nibble5*, *Char NibbleF Nibble6*,
Char NibbleF Nibble7, *Char NibbleF Nibble8*, *Char NibbleF Nibble9*,
Char NibbleF NibbleA, *Char NibbleF NibbleB*, *Char NibbleF NibbleC*,
Char NibbleF NibbleD, *Char NibbleF NibbleE*, *Char NibbleF NibbleF*]
unfolding *enum-char-def enum-nibble-def* **by** *simp*

instance by *default*
(*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 by *default*
(*auto simp add*: *enum-all enum-option-def*, *case-tac x*, *auto intro*: *inj-onI simp add*: *distinct-map enum-distinct*)

end

end

36 Eval-Witness: Evaluation Oracle with ML witnesses

theory *Eval-Witness*
imports *List Main*
begin

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

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

true, this is effectively a proof of $\exists\, x.\ P\ x$.

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

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

class *ml-equiv*

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

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

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

ML ⟪
```
structure Eval-Witness-Method =
struct

val eval-ref : (unit -> bool) option ref = 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-ML.eval-term (Eval-Witness-Method.eval-ref, Eval-Witness-Method.eval-ref)
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*

36.1 Toy Examples

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

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

lemma ∃ *l*. *length* (*l*::*bool list*) = *3*
apply (*eval-witness* [*true*,*true*,*true*])
done

Multiple witnesses

lemma ∃ *k l*. *length* (*k*::*bool list*) = *length* (*l*::*bool list*)
apply (*eval-witness* [] [])
done

36.2 Discussion

36.2.1 Conflicts

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

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

36.2.2 Haskell

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

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

end

37 Executable-Set: Implementation of finite sets by lists

theory *Executable-Set*
imports *Main*
begin

37.1 Definitional rewrites

definition *subset* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
subset = *op* ≤

declare *subset-def* [*symmetric*, *code unfold*]

lemma [*code*]: *subset A B* ⟷ (∀ *x*∈*A*. *x* ∈ *B*)
unfolding *subset-def subset-eq* **..**

definition *is-empty* :: *'a set* ⇒ *bool* **where**
is-empty A ⟷ *A* = {}

definition *eq-set* :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
[*code del*]: *eq-set* = *op* =

lemma [*code*]: *eq-set A B* ⟷ *A* ⊆ *B* ∧ *B* ⊆ *A*
unfolding *eq-set-def* **by** *auto*

lemma [*code*]:
a ∈ *A* ⟷ (∃ *x*∈*A*. *x* = *a*)
unfolding *bex-triv-one-point1* **..**

definition *filter-set* :: (*'a* ⇒ *bool*) ⇒ *'a set* ⇒ *'a set* **where**
filter-set P xs = {*x*∈*xs*. *P x*}

declare *filter-set-def*[*symmetric*, *code unfold*]

37.2 Operations on lists

37.2.1 Basic definitions

definition
flip :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'b* ⇒ *'a* ⇒ *'c* **where**
flip f a b = *f b a*

definition
member :: *'a list* ⇒ *'a* ⇒ *bool* **where**
member xs x ⟷ *x* ∈ *set xs*

definition

```
  insertl :: 'a ⇒ 'a list ⇒ 'a list where
  insertl x xs = (if member xs x then xs else x#xs)

lemma [code target: List]: member [] y ⟷ False
  and [code target: List]: member (x#xs) y ⟷ y = x ∨ member xs y
  unfolding member-def by (induct xs) simp-all

fun
  drop-first :: ('a ⇒ bool) ⇒ 'a list ⇒ 'a list where
  drop-first f [] = []
| drop-first f (x#xs) = (if f x then xs else x # drop-first f xs)
declare drop-first.simps [code del]
declare drop-first.simps [code target: List]

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

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

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

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

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

lemma foldr-remove1-empty [simp]:
```

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

37.2.2 Derived definitions

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

lemmas *unionl-eq* = *unionl.simps*(*2*)

function *intersect* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
intersect [] *ys* = []
| *intersect xs* [] = []
| *intersect xs ys* = *filter* (*member xs*) *ys*
by *pat-completeness auto*
termination by *lexicographic-order*

lemmas *intersect-eq* = *intersect.simps*(*3*)

function *subtract* :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
subtract [] *ys* = *ys*
| *subtract xs* [] = []
| *subtract xs ys* = *foldr remove1 xs ys*
by *pat-completeness auto*
termination by *lexicographic-order*

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

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

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

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

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

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

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

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

37.3 Isomorphism proofs

lemma *iso-member*:
member xs x ⟷ *x* ∈ *set xs*
unfolding *member-def mem-iff* **..**

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

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

lemma *iso-union*:
set (*unionl xs ys*) = *set xs* ∪ *set ys*
unfolding *unionl-eq*
by (*induct xs arbitrary*: *ys*) (*simp-all add*: *iso-insert*)

lemma *iso-intersect*:
set (*intersect xs ys*) = *set xs* ∩ *set ys*
unfolding *intersect-eq Int-def* **by** (*simp add*: *Int-def iso-member*) *auto*

definition
subtract' :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
subtract' = *flip subtract*

lemma *iso-subtract*:
fixes *ys*
assumes *distnct*: *distinct ys*
shows *set* (*subtract' ys xs*) = *set ys* − *set xs*
and *distinct* (*subtract' ys xs*)
unfolding *subtract'-def flip-def subtract-eq*

using *distnct* **by** (*induct xs arbitrary*: *ys*) *auto*

lemma *iso-map-distinct*:
set (*map-distinct f xs*) = *image f* (*set xs*)
unfolding *map-distinct-eq* **by** (*induct xs*) (*simp-all add*: *iso-insert*)

lemma *iso-unions*:
set (*unions xss*) = ⋃ *set* (*map set xss*)
unfolding *unions-eq*
proof (*induct xss*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons xs xss*) **thus** *?case* **by** (*induct xs*) (*simp-all add*: *iso-insert*)
qed

lemma *iso-intersects*:
set (*intersects* (*xs*#*xss*)) = ⋂ *set* (*map set* (*xs*#*xss*))
by (*induct xss*) (*simp-all add*: *Int-def iso-member*, *auto*)

lemma *iso-UNION*:
set (*map-union xs f*) = *UNION* (*set xs*) (*set o f*)
unfolding *map-union-def iso-unions* **by** *simp*

lemma *iso-INTER*:
set (*map-inter* (*x*#*xs*) *f*) = *INTER* (*set* (*x*#*xs*)) (*set o f*)
unfolding *map-inter-def iso-intersects* **by** (*induct xs*) (*simp-all add*: *iso-member*, *auto*)

definition
Blall :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blall = *flip list-all*
definition
Blex :: *'a list* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Blex = *flip list-ex*

lemma *iso-Ball*:
Blall xs f = *Ball* (*set xs*) *f*
unfolding *Blall-def flip-def* **by** (*induct xs*) *simp-all*

lemma *iso-Bex*:
Blex xs f = *Bex* (*set xs*) *f*
unfolding *Blex-def flip-def* **by** (*induct xs*) *simp-all*

lemma *iso-filter*:
set (*filter P xs*) = *filter-set P* (*set xs*)
unfolding *filter-set-def* **by** (*induct xs*) *auto*

37.4 code generator setup

ML ⟪
nonfix inter;
nonfix union;
nonfix subset;
⟫

37.4.1 const serializations

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

end

38 Float: Floating-Point Numbers

theory *Float*
imports *Complex-Main*
begin

definition
pow2 :: *int* ⇒ *real* **where**
[*simp*]: *pow2 a* = (*if* (*0* <= *a*) *then* (*2^(nat a)*) *else* (*inverse* (*2^(nat* (−*a*)))))

datatype *float* = *Float int int*

fun *Ifloat* :: *float* ⇒ *real* **where**
Ifloat (*Float a b*) = *real a* ∗ *pow2 b*

instantiation *float* :: *zero* **begin**
definition *zero-float* **where** *0* = *Float 0 0*
instance ..
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

fun *mantissa* :: *float* ⇒ *int* **where**
mantissa (Float a b) = a

fun *scale* :: *float* ⇒ *int* **where**
scale (Float a b) = b

lemma *Ifloat-neg-exp*: *e < 0 ⟹ Ifloat (Float m e) = real m * inverse (2^nat (−e))* **by** *auto*
lemma *Ifloat-nge0-exp*: *¬ 0 ≤ e ⟹ Ifloat (Float m e) = real m * inverse (2^nat (−e))* **by** *auto*
lemma *Ifloat-ge0-exp*: *0 ≤ e ⟹ Ifloat (Float m e) = real m * (2^nat e)* **by** *auto*

lemma *Float-num*[*simp*]: **shows**
Ifloat (Float 1 0) = 1 **and** *Ifloat (Float 1 1) = 2* **and** *Ifloat (Float 1 2) = 4* **and**
Ifloat (Float 1 −1) = 1/2 **and** *Ifloat (Float 1 −2) = 1/4* **and** *Ifloat (Float 1 −3) = 1/8* **and**
Ifloat (Float −1 0) = −1 **and** *Ifloat (Float (number-of n) 0) = number-of n*
by *auto*

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]: Ifloat (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]: Ifloat f = Ifloat (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: Ifloat (Float a b) = Ifloat (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*: *Ifloat* (*Float a b*) = *Ifloat* (*Float a' b'*)
shows $a = a' \wedge b = b'$
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 *Ifloat-eq*: *Ifloat f* = *Ifloat 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 *Ifloat* (*normfloat f*) = *Ifloat* (*normfloat g*)
by (*simp add*: *Ifloat-eq*)
then have *float-eq*: *Ifloat* (*Float a b*) = *Ifloat* (*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** $a = a' \wedge b = b'$ **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** $a = a' \wedge b = b'$ **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
fun uminus-float where [simp del]: uminus-float (Float m e) = Float (−m) e
instance ..
end

instantiation float :: minus begin
fun 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

fun float-pprt :: float ⇒ float where
float-pprt (Float a e) = (if 0 <= a then (Float a e) else 0)

fun float-nprt :: float ⇒ float where
float-nprt (Float a e) = (if 0 <= a then 0 else (Float a e))

instantiation float :: ord begin
definition le-float-def: z ≤ w ≡ Ifloat z ≤ Ifloat w
definition less-float-def: z < w ≡ Ifloat z < Ifloat w
instance ..
end

lemma Ifloat-add[simp]: Ifloat (a + b) = Ifloat a + Ifloat b
  by (cases a, cases b, simp add: algebra-simps plus-float.simps,
      auto simp add: pow2-int[symmetric] pow2-add[symmetric])

lemma Ifloat-minus[simp]: Ifloat (− a) = − Ifloat a
  by (cases a, simp add: uminus-float.simps)

lemma Ifloat-sub[simp]: Ifloat (a − b) = Ifloat a − Ifloat b
```

by (*cases a*, *cases b*, *simp add*: *minus-float.simps*)

lemma *Ifloat-mult*[*simp*]: *Ifloat* (*a*∗*b*) = *Ifloat a* ∗ *Ifloat b*
by (*cases a*, *cases b*, *simp add*: *times-float.simps pow2-add*)

lemma *Ifloat-0*[*simp*]: *Ifloat 0* = *0*
by (*auto simp add*: *zero-float-def float-zero*)

lemma *Ifloat-1*[*simp*]: *Ifloat 1* = *1*
by (*auto simp add*: *one-float-def*)

lemma *zero-le-float*:
(*0* <= *Ifloat* (*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*:
(*Ifloat* (*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 *Ifloat.simps*[*simp del*]

lemma *Ifloat-pprt*[*simp*]: *Ifloat* (*float-pprt a*) = *pprt* (*Ifloat a*)
by (*cases a*, *auto simp add*: *float-pprt.simps zero-le-float float-le-zero float-zero*)

lemma *Ifloat-nprt*[*simp*]: *Ifloat* (*float-nprt a*) = *nprt* (*Ifloat 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*) ≠ *1*
by (*simp add*: *zero-float-def one-float-def*)
qed

lemma *float-le-simp*: ((*x*::*float*) ≤ *y*) = (*0* ≤ *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 *Ifloat-min*: *Ifloat* (*min x y*) = *min* (*Ifloat x*) (*Ifloat y*) **unfolding** *min-def le-float-def* **by** *auto*
lemma *Ifloat-max*: *Ifloat* (*max a b*) = *max* (*Ifloat a*) (*Ifloat b*) **unfolding** *max-def le-float-def* **by** *auto*

instantiation *float* :: *power* **begin**
fun *power-float* **where** [*simp del*]: (*Float m e*) ^ *n* = *Float* (*m* ^ *n*) (*e* ∗ *int n*)
instance ..
end

instance *float* :: *recpower*
proof (*intro-classes*)
fix *a* :: *float* **show** *a* ^ *0* = *1* **by** (*cases a*, *auto simp add*: *power-float.simps one-float-def*)
next
fix *a* :: *float* **and** *n* :: *nat* **show** *a* ^ (*Suc n*) = *a* * *a* ^ *n*
by (*cases a*, *auto simp add*: *power-float.simps times-float.simps algebra-simps*)
qed

lemma *float-power*: *Ifloat* (*x* ^ *n*) = (*Ifloat x*) ^ *n*
proof (*cases x*)
case (*Float m e*)

have *pow2 e* ^ *n* = *pow2* (*e* * *int n*)
proof (*cases e* >= *0*)
case *True* **hence** *e-nat*: *e* = *int* (*nat e*) **by** *auto*
hence *pow2 e* ^ *n* = (*2* ^ *nat e*) ^ *n* **using** *pow2-int*[*of nat e*] **by** *auto*
thus *?thesis* **unfolding** *power-mult*[*symmetric*] **unfolding** *pow2-int*[*symmetric*] *int-mult e-nat*[*symmetric*] **.**
next
case *False* **hence** *e-minus*: −*e* = *int* (*nat* (−*e*)) **by** *auto*
hence *pow2* (−*e*) ^ *n* = (*2* ^ *nat* (−*e*)) ^ *n* **using** *pow2-int*[*of nat* (−*e*)] **by** *auto*
hence *pow2* (−*e*) ^ *n* = *pow2* ((−*e*) * *int n*) **unfolding** *power-mult*[*symmetric*] **unfolding** *pow2-int*[*symmetric*] *int-mult e-minus*[*symmetric*] *zmult-zminus* **.**
thus *?thesis* **unfolding** *pow2-neg*[*of* −*e*] *pow2-neg*[*of* −*e* * *int n*] **unfolding** *zmult-zminus zminus-zminus nonzero-power-inverse*[*OF pow2-neq-zero*, *symmetric*]
using *nonzero-inverse-eq-imp-eq*[*OF - pow2-neq-zero pow2-neq-zero*] **by** *auto*
qed
thus *?thesis* **by** (*auto simp add*: *Float power-mult-distrib Ifloat.simps power-float.simps*)
qed

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*: $0 < Float\ m\ e \Longrightarrow 0 < m$
unfolding *less-float-def Ifloat.simps Ifloat-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 $0 \leq real\ m$ **and** $1 \leq real\ m$ **by** *auto*

show $e < 0$
proof (*rule ccontr*)
assume $\neg\ e < 0$ **hence** $0 \leq e$ **by** *auto*
hence $1 \leq pow2\ e$ **unfolding** *pow2-def* **by** *auto*
from *mult-mono*[*OF* ‹$1 \leq real\ m$› *this* ‹$0 \leq real\ m$›]
have $1 \leq Float\ m\ e$ **by** (*simp add*: *le-float-def Ifloat.simps*)
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** $m < 2\hat{}(nat\ (-e))$
proof −
have $e < 0$ **using** *float-pos-less1-e-neg assms* **by** *auto*
have $\bigwedge x.\ (0::real) < 2\hat{}x$ **by** *auto*
have $real\ m < 2\hat{}(nat\ (-e))$ **using** ‹$Float\ m\ e < 1$›
unfolding *less-float-def Ifloat-neg-exp*[*OF* ‹$e < 0$›] *Ifloat-1*
real-mult-less-iff1[*of - - 1*, *OF* ‹$0 < 2\hat{}(nat\ (-e))$›, *symmetric*]
real-mult-assoc **by** *auto*
thus *?thesis* **unfolding** *real-of-int-less-iff*[*symmetric*] **by** *auto*
qed

function *bitlen* :: *int* ⇒ *int* **where**
$bitlen\ 0 = 0$ |
$bitlen\ -1 = 1$ |
$0 < x \Longrightarrow bitlen\ x = 1 + (bitlen\ (x\ div\ 2))$ |
$x < -1 \Longrightarrow bitlen\ x = 1 + (bitlen\ (x\ div\ 2))$
apply (*case-tac* $x = 0 \vee x = -1 \vee x < -1 \vee x > 0$)
apply *auto*
done
termination by (*relation measure* (*nat o abs*), *auto*)

lemma *bitlen-ge0*: $0 \leq bitlen\ x$ **by** (*induct x rule*: *bitlen.induct*, *auto*)
lemma *bitlen-ge1*: $x \neq 0 \Longrightarrow 1 \leq bitlen\ x$ **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* ≤ *Float m e*
shows *0* ≤ *e* + (*bitlen m* − *1*)
proof (*cases 0* ≤ *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* ≠ *0* **by** *auto*
case *True* **with** *bitlen-ge1*[*OF* ‹*m* ≠ *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* ≠ *0* **and** *1* < (*2*::*int*) **by** *auto*
case *False* **let** *?S* = *2^*(*nat* (−*e*))
have *1* ≤ *real m* ∗ *inverse ?S* **using** *assms* **unfolding** *le-float-def Ifloat-nge0-exp*[*OF False*] **by** *auto*
hence *1* ∗ *?S* ≤ *real m* ∗ *inverse ?S* ∗ *?S* **by** (*rule mult-right-mono*, *auto*)
hence *?S* ≤ *real m* **unfolding** *mult-assoc* **by** *auto*
hence *?S* ≤ *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** *Ifloat* (*Float m* (− (*bitlen m* − *1*))) = *real m* / *2^nat* (*bitlen m* − *1*)
proof (*cases* − (*bitlen m* − *1*) = *0*)
case *True* **show** *?thesis* **unfolding** *Ifloat.simps pow2-def* **using** *True* **by** *auto*
next
case *False* **hence** *P*: ¬ *0* ≤ − (*bitlen m* − *1*) **using** *bitlen-ge1*[*OF* ‹$m \neq 0$›] **by** *auto*
show *?thesis* **unfolding** *Ifloat-nge0-exp*[*OF P*] *real-divide-def* **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* $0 < 2 * b \vee 2 * b < -1$)
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* $0 < 2 * b + 1 \vee 2 * b + 1 < -1$)
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 *Ifloat* (*lapprox-posrat prec x y*) ≤ *real x* / *real y*
proof −
let *?l* = *nat* (*int prec* + *bitlen y* − *bitlen x*)

have *real* (*x* ∗ *2*^*?l div y*) ∗ *inverse* (*2*^*?l*) ≤ (*real* (*x* ∗ *2*^*?l*) / *real y*) ∗ *inverse* (*2*^*?l*)
by (*rule mult-right-mono*, *fact real-of-int-div4*, *simp*)
also have ... ≤ (*real x* / *real y*) ∗ *2*^*?l* ∗ *inverse* (*2*^*?l*) **by** *auto*
finally have *real* (*x* ∗ *2*^*?l div y*) ∗ *inverse* (*2*^*?l*) ≤ *real x* / *real y* **unfolding** *real-mult-assoc* **by** *auto*
thus *?thesis* **unfolding** *lapprox-posrat-def Let-def normfloat Ifloat.simps*
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*) ≤ *real* (*x* ∗ *c div y*) ∗ *inverse* (*real c*)
proof −
have *c* ∗ (*x div y*) + *0* ≤ *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* ≤ *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*) ≤ *real* (*x* * *c div y*) * *inverse* (*real c*)
using ‹*0* < *c*› **by** *auto*
thus *?thesis* **unfolding** *real-mult-assoc* **using** ‹*0* < *c*› **by** *auto*
qed

lemma *lapprox-posrat-bottom*: **assumes** *0* < *y*
shows *real* (*x div y*) ≤ *Ifloat* (*lapprox-posrat n x y*)
proof −
have *pow*: ⋀*x*. (*0*::*int*) < *2*^*x* **by** *auto*
show *?thesis*
unfolding *lapprox-posrat-def Let-def Ifloat-add normfloat Ifloat.simps pow2-minus pow2-int*
using *real-of-int-div-mult*[*OF* ‹*0* < *y*› *pow*] **by** *auto*
qed

lemma *lapprox-posrat-nonneg*: **assumes** *0* ≤ *x* **and** *0* < *y*
shows *0* ≤ *Ifloat* (*lapprox-posrat n x y*)
proof −
show *?thesis*
unfolding *lapprox-posrat-def Let-def Ifloat-add normfloat Ifloat.simps 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* ≤ *Ifloat* (*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 Ifloat.simps 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 real-divide-def ..
   finally show ?thesis unfolding rapprox-posrat-def Let-def normfloat Ifloat.simps
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 Ifloat (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 Ifloat.simps 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** ‹$x \leq y$› **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*[*symmetric*] *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*[*symmetric*] *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 Ifloat.simps 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 *Ifloat* (*rapprox-posrat n x y*) *< 1*
proof (*cases x = 0*)
case *True* **thus** *?thesis* **unfolding** *rapprox-posrat-def True Let-def normfloat Ifloat.simps* **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 Ifloat.simps 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^nat* (*bitlen x* − *1*) ≤ *y* * *x*
using *bitlen-bounds*[*OF* ‹*0* < *x*›, *THEN conjunct1*] ‹*0* < *y*›[*THEN less-imp-le*] **by** (*rule mult-left-mono*)
also have ... ≤ *2^nat* (*bitlen y*) * *x* **using** *bitlen-bounds*[*OF* ‹*0* < *y*›, *THEN conjunct2*, *THEN less-imp-le*] ‹*0* ≤ *x*› **by** (*rule mult-right-mono*)
also have ... ≤ *x* * *2^nat* (*int* (*n* − *1*) + *bitlen y*) **unfolding** *mult-commute*[*of x*] **by** (*rule mult-right-mono*, *auto simp add*: ‹*0* ≤ *x*›)
finally have *real y* * *2^nat* (*bitlen x* − *1*) * *inverse* (*2^nat* (*bitlen x* − *1*)) ≤ *real x* * *2^nat* (*int* (*n* − *1*) + *bitlen y*) * *inverse* (*2^nat* (*bitlen x* − *1*))
unfolding *real-of-int-le-iff*[*symmetric*] **by** *auto*
hence *real y* ≤ *real x* * (*2^nat* (*int* (*n* − *1*) + *bitlen y*) / (*2^nat* (*bitlen x* − *1*)))
unfolding *real-mult-assoc real-divide-def* **by** *auto*
also have ... = *real x* * (*2^*(*nat* (*int* (*n* − *1*) + *bitlen y*) − *nat* (*bitlen x* − *1*))) **using** *power-diff*[*of 2*::*real*, *OF* - *len-less*] **by** *auto*
finally have *y* ≤ *x* * *2^?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*) ≤ (*real x / real y*) * *2^?l* **unfolding** *real-of-int-mult times-divide-eq-left real-of-int-power*[*symmetric*] *real-number-of* **.**
also have ... < *1/2 * 2^?l* **using** ‹*real x / real y < 1/2*› **by** (*rule mult-strict-right-mono*, *auto*)
finally have *?X div y * 2 < 2^?l* **unfolding** *real-of-int-less-iff*[*of - 2^?l*, *symmetric*] **by** *auto*
hence *?X div y + 1 < 2^?l* **using** ‹*0 < ?X div y*› **by** *auto*
hence *real* (*?X div y + 1*) * *inverse* (*2^?l*) < *2^?l * inverse* (*2^?l*)
unfolding *real-of-int-less-iff*[*of - 2^?l*, *symmetric*] *real-of-int-power*[*symmetric*] *real-number-of*
by (*rule mult-strict-right-mono*, *auto*)
hence *real* (*?X div y + 1*) * *inverse* (*2^?l*) < *1* **by** *auto*
thus *?thesis* **unfolding** *rapprox-posrat-def Let-def normfloat Ifloat.simps 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*: ⋀*y prec x*. ⟦*y = 0*; *ps = (prec, x, 0)*⟧ ⟹ *P*
and *A*: ⋀*x y prec*. ⟦*0 ≤ x*; *0 < y*; *ps = (prec, x, y)*⟧ ⟹ *P*
and *B*: ⋀*x y prec*. ⟦*x < 0*; *0 < y*; *ps = (prec, x, y)*⟧ ⟹ *P*
and *C*: ⋀*x y prec*. ⟦*x < 0*; *y < 0*; *ps = (prec, x, y)*⟧ ⟹ *P*
and *D*: ⋀*x y prec*. ⟦*0 ≤ x*; *y < 0*; *ps = (prec, x, y)*⟧ ⟹ *P*
shows *P*
proof −
obtain *prec x y* **where** [*simp*]: *ps = (prec, x, y)* **by** (*cases ps*, *auto*)
from *Y* **have** *y = 0* ⟹ *P* **by** *auto*
moreover { assume *0 < y* **have** *P* **proof** (*cases 0 ≤ 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 ≤ 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 ∨ 0 < y ∨ y < 0*, *auto*)
qed

function *lapprox-rat* :: *nat ⇒ int ⇒ int ⇒ float*
where
y = 0 ⟹ *lapprox-rat prec x y = 0*
| *0 ≤ x* ⟹ *0 < y* ⟹ *lapprox-rat prec x y = lapprox-posrat prec x y*
| *x < 0* ⟹ *0 < y* ⟹ *lapprox-rat prec x y = − (rapprox-posrat prec (−x) y)*
| *x < 0* ⟹ *y < 0* ⟹ *lapprox-rat prec x y = lapprox-posrat prec (−x) (−y)*
| *0 ≤ x* ⟹ *y < 0* ⟹ *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 ≤ 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*: *Ifloat (lapprox-rat prec x y) ≤ real x / real 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*: *lapprox-posrat*)
apply (*case-tac x < 0 ∧ 0 < y*)
apply *simp*
apply (*subst minus-le-iff*)
apply (*rule h*[*OF rapprox-posrat*])
apply (*simp-all*)
apply (*case-tac x < 0 ∧ y < 0*)
apply *simp*
apply (*rule h*[*OF - lapprox-posrat*])
apply (*simp-all*)
apply (*case-tac 0 ≤ x ∧ 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 ≤ x* **and** *0 < y*
shows *real (x div y) ≤ Ifloat (lapprox-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 ⟹ rapprox-rat prec x y = 0
| *0 ≤ x ⟹ 0 < y ⟹ rapprox-rat prec x y = rapprox-posrat prec x y*
| *x < 0 ⟹ 0 < y ⟹ rapprox-rat prec x y = − (lapprox-posrat prec (−x) y)*
| *x < 0 ⟹ y < 0 ⟹ rapprox-rat prec x y = rapprox-posrat prec (−x) (−y)*
| *0 ≤ x ⟹ y < 0 ⟹ rapprox-rat prec x y = − (lapprox-posrat prec x (−y))*
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 ≤ Ifloat (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 ≤ x* **and** *0 < y* **and** *x ≤ y*
shows *Ifloat (rapprox-rat n x y) ≤ 1*
unfolding *rapprox-rat.simps*(*2*)[*OF ‹0 ≤ x› ‹0 < y›*] **using** *rapprox-posrat-le1*[*OF assms*] **.**

lemma *rapprox-rat-neg*: **assumes** *x < 0* **and** *0 < y*
shows *Ifloat (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 ≤ x* **and** *y < 0*
shows *Ifloat (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 ≤ 0* **and** *0 < y*
shows *Ifloat (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*: *Ifloat* (*float-divl prec x y*) ≤ *Ifloat x* / *Ifloat 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 *Ifloat* (*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 *Ifloat* (*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*) ∗ *Ifloat* (*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 Ifloat.simps*)
qed

lemma *float-divl-lower-bound*: **assumes** $0 \leq x$ **and** $0 < y$ **shows** $0 \leq$ *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 $0 \leq xm$ **using** ‹$0 \leq x$›[*unfolded x-eq le-float-def Ifloat.simps Ifloat-0 zero-le-mult-iff*]
by *auto*
have $0 < ym$ **using** ‹$0 < y$›[*unfolded y-eq less-float-def Ifloat.simps Ifloat-0 zero-less-mult-iff*] **by** *auto*

have $\bigwedge n.\ 0 \leq$ *Ifloat* (*Float 1 n*) **unfolding** *Ifloat.simps* **using** *zero-le-pow2* **by** *auto*
moreover have $0 \leq$ *Ifloat* (*lapprox-rat prec xm ym*) **by** (*rule order-trans*[*OF - lapprox-rat-bottom*[*OF* ‹$0 \leq xm$› ‹$0 < ym$›]], *auto simp add*: ‹$0 \leq xm$› *pos-imp-zdiv-nonneg-iff*[*OF* ‹$0 < ym$›])
ultimately show $0 \leq$ *float-divl prec x y*
unfolding *x-eq y-eq float-divl.simps Let-def le-float-def Ifloat-0* **by** (*auto intro*!: *mult-nonneg-nonneg*)
qed

lemma *float-divl-pos-less1-bound*: **assumes** $0 < x$ **and** $x < 1$ **and** $0 <$ *prec* **shows** $1 \leq$ *float-divl prec 1 x*
proof (*cases x*)
case (*Float m e*)
from ‹$0 < x$› ‹$x < 1$›[*unfolded Float* **by** *auto*
let *?b* = *nat* (*bitlen m*) **and** *?e* = *nat* ($-e$)
have $1 \leq m$ **and** $m \neq 0$ **using** ‹$0 < m$› **by** *auto*
with *bitlen-bounds*[*OF* ‹$0 < m$›] **have** $m < 2\hat{}?b$ **and** (2::*int*) $\leq 2\hat{}?b$ **by** *auto*
hence $1 \leq$ *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*: $\bigwedge x.$ (2::*real*)$\hat{}x \neq 0$ **by** *auto*

from *float-less1-mantissa-bound* ‹$0 < x$› ‹$x < 1$› *Float*
have $m < 2\hat{}?e$ **by** *auto*
with *bitlen-bounds*[*OF* ‹$0 < m$›, *THEN conjunct1*]
have (2::*int*)$\hat{}$*nat* (*bitlen m* − 1) $< 2\hat{}?e$ **by** (*rule order-le-less-trans*)
from *power-less-imp-less-exp*[*OF - this*]
have *bitlen m* $\leq -$ *e* **by** *auto*
hence (2::*real*)$\hat{}?b \leq 2\hat{}?e$ **by** *auto*
hence (2::*real*)$\hat{}?b *$ *inverse* ($2\hat{}?b$) $\leq 2\hat{}?e *$ *inverse* ($2\hat{}?b$) **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 real-mult-assoc*[*symmetric*] *right-inverse*[*OF pow-not0*] *real-mult-1*], *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 Ifloat-mult normfloat Ifloat.simps 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*: *Ifloat x* / *Ifloat y* ≤ *Ifloat* (*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 *Ifloat* (*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 *Ifloat* (*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 Ifloat.simps*)
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 / Ifloat x* **using** ‹*0 < x*› **and** ‹*x < 1*› **unfolding** *less-float-def* **by** *auto*
also have ... ≤ *Ifloat* (*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 ≤ 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 ≤ 0* **using** ‹*x ≤ 0*›[*unfolded x-eq le-float-def Ifloat.simps Ifloat-0 mult-le-0-iff*] **by** *auto*
have *0 < ym* **using** ‹*0 < y*›[*unfolded y-eq less-float-def Ifloat.simps Ifloat-0 zero-less-mult-iff*] **by** *auto*

have ⋀*n*. *0 ≤ Ifloat* (*Float 1 n*) **unfolding** *Ifloat.simps* **using** *zero-le-pow2* **by** *auto*
moreover have *Ifloat* (*rapprox-rat prec xm ym*) ≤ *0* **using** *rapprox-rat-nonpos-pos*[*OF* ‹*xm ≤ 0*› ‹*0 < ym*›] .
ultimately show *float-divr prec x y ≤ 0*
unfolding *x-eq y-eq float-divr.simps Let-def le-float-def Ifloat-0 Ifloat-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 Ifloat.simps Ifloat-0 zero-le-mult-iff*]
by *auto*
have $ym < 0$ **using** ‹$y < 0$›[*unfolded y-eq less-float-def Ifloat.simps Ifloat-0 mult-less-0-iff*] **by** *auto*
hence $0 < -\ ym$ **by** *auto*

have $\bigwedge n.\ 0 \leq Ifloat\ (Float\ 1\ n)$ **unfolding** *Ifloat.simps* **using** *zero-le-pow2* **by** *auto*
moreover have *Ifloat* (*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 Ifloat-0 Ifloat-mult* **by** (*auto intro*!: *mult-nonneg-nonpos*)
qed

fun *round-down* :: *nat* ⇒ *float* ⇒ *float* **where**
round-down prec (*Float m e*) = (*let d* = *bitlen m* − *int prec in*
if 0 < *d then let P* = 2^*nat d* ; *n* = *m div P in Float n* (*e* + *d*)
else Float m e)

fun *round-up* :: *nat* ⇒ *float* ⇒ *float* **where**
round-up prec (*Float m e*) = (*let d* = *bitlen m* − *int prec in*
if 0 < *d then let P* = 2^*nat d* ; *n* = *m div P* ; *r* = *m mod P in Float* (*n* + (*if r* = 0 *then* 0 *else* 1)) (*e* + *d*)
else Float m e)

lemma *round-up*: *Ifloat x* ≤ *Ifloat* (*round-up prec x*)
proof (*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* **unfolding** *pow2-int*[*symmetric*] *power-real-number-of*[*symmetric*]
by *auto*
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 *Ifloat* (*Float m e*) = *Ifloat* (*Float* (*m div ?p*) (*e + ?d*)) **unfolding** *Ifloat.simps 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 *Ifloat* (*Float m e*) *≤ Ifloat* (*Float* (*m div ?p + 1*) (*e + ?d*)) **unfolding** *Ifloat.simps 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*: *Ifloat* (*round-down prec x*) *≤ Ifloat 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* **unfolding** *pow2-int*[*symmetric*] *power-real-number-of*[*symmetric*] **by** *auto*
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 *Ifloat* (*Float* (*m div ?p*) (*e + ?d*)) *≤ Ifloat* (*Float m e*) **unfolding** *Ifloat.simps 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: Ifloat (lb-mult prec x y) ≤ Ifloat (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 Ifloat (lb-mult prec x y) ≤ Ifloat (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[symmetric] 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]
Ifloat.simps pow2-add real-mult-commute real-mult-assoc by auto
  qed
  also have ... = Ifloat (x * y) unfolding normfloat ..
  finally show ?thesis .
qed
```

lemma *ub-mult*: *Ifloat* $(x * y) \leq$ *Ifloat* (*ub-mult prec x y*)
proof (*cases normfloat* $(x * y)$)
case (*Float m e*)
hence *odd m* $\vee$ $(m = 0 \wedge e = 0)$ **by** (*rule normfloat-imp-odd-or-zero*)
let *?l* = *bitlen m* − *int prec*
have *Ifloat* $(x * y)$ = *Ifloat* (*normfloat* $(x * y)$) **unfolding** *normfloat* **..**
also have ... $\leq$ *Ifloat* (*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* $\leq$ *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)*) $\leq$ *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 ... $\leq$ (*real* (*m div 2^(nat ?l)*) + *1*) ∗ *2^(nat ?l)* **unfolding** *real-add-mult-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*] *Ifloat.simps pow2-add real-mult-commute real-mult-assoc* **by** *auto*
qed
finally show *?thesis* **.**
qed

fun *float-abs* :: *float* ⇒ *float* **where**
float-abs (*Float m e*) = *Float* |*m*| *e*

instantiation *float* :: *abs* **begin**
definition *abs-float-def*: |*x*| = *float-abs x*
instance ..
end

lemma *Ifloat-abs*: *Ifloat* |*x*| = |*Ifloat 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 Ifloat.simps* **by** *auto*
qed

fun *floor-fl* :: *float* ⇒ *float* **where**
floor-fl (*Float m e*) = (*if 0* $\leq$ *e then Float m e*

else Float (m div (2 ^ (nat (−e)))) 0)

lemma *floor-fl*: *Ifloat (floor-fl x) ≤ Ifloat 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 *Ifloat (Float (m div (2 ^ (nat (−e)))) 0) = real (m div 2 ^ (nat (−e)))*
unfolding *Ifloat.simps* **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** *power-real-number-of*[*symmetric*] *real-divide-def* **..**
also have *... = Ifloat (Float m e)* **unfolding** *Ifloat.simps 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*]

fun *ceiling-fl* :: *float ⇒ float* **where**
ceiling-fl (Float m e) = (if 0 ≤ e then Float m e
else Float (m div (2 ^ (nat (−e))) + 1) 0)

lemma *ceiling-fl*: *Ifloat x ≤ Ifloat (ceiling-fl x)*
proof (*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 *Ifloat (Float m e) = real m * inverse (2 ^ (nat (−e)))* **unfolding** *Ifloat.simps me-eq pow2-int pow2-neg*[*of e*] **..**
also have *... = real m / real ((2::int) ^ (nat (−e)))* **unfolding** *power-real-number-of*[*symmetric*] *real-divide-def* **..**
also have *... ≤ 1 + real (m div 2 ^ (nat (−e)))* **using** *real-of-int-div3*[*unfolded diff-le-eq*] **.**
also have *... = Ifloat (Float (m div (2 ^ (nat (−e))) + 1) 0)* **unfolding** *Ifloat.simps* **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 ≤ Ifloat x and real k ∗ y ≤ Ifloat x (is
?k ∗ y ≤ ?x)
  assumes 0 < Ifloat lb Ifloat lb ≤ y (is ?lb ≤ y) y ≤ Ifloat ub (is y ≤ ?ub)
  shows Ifloat (lb-mod prec x ub lb) ≤ ?x − ?k ∗ y
proof −
  have ?lb ≤ ?ub by (auto!)
  have 0 ≤ ?lb and ?lb ≠ 0 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 ... ≤ Ifloat (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 Ifloat-sub Ifloat-mult by auto
qed

lemma ub-mod: fixes k :: int assumes 0 ≤ Ifloat x and Ifloat x ≤ real k ∗ y (is
?x ≤ ?k ∗ y)
  assumes 0 < Ifloat lb Ifloat lb ≤ y (is ?lb ≤ y) y ≤ Ifloat ub (is y ≤ ?ub)
  shows ?x − ?k ∗ y ≤ Ifloat (ub-mod prec x ub lb)
proof −
  have ?lb ≤ ?ub by (auto!)
  hence 0 ≤ ?lb and 0 ≤ ?ub and ?ub ≠ 0 by (auto!)
   have Ifloat (floor-fl (float-divl prec x ub)) ∗ ?lb ≤ ?x / ?ub ∗ ?lb by (metis
mult-right-mono order-trans ‹0 ≤ ?lb› ‹?lb ≤ ?ub› float-divl floor-fl)
 also have ... ≤ ?x by (metis mult-left-mono[OF ‹?lb ≤ ?ub› ‹0 ≤ ?x›] divide-right-mono[OF
- ‹0 ≤ ?ub› ] times-divide-eq-left nonzero-mult-divide-cancel-right[OF ‹?ub ≠ 0›])
  also have ... ≤ ?k ∗ y using assms by auto
  finally show ?thesis unfolding ub-mod-def Ifloat-sub Ifloat-mult by auto
qed

lemma le-float-def': f ≤ g = (case f − g of Float a b ⇒ a ≤ 0)
proof −
  have le-transfer: (f ≤ g) = (Ifloat (f − g) ≤ 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'*: $f \leq g = (Ifloat\ (Float\ a\ b) \leq 0)$ **by** *simp*
show *?thesis* **by** (*simp add*: *le-transfer' f-diff-g float-le-zero*)
qed

lemma *float-less-zero*:
$(Ifloat\ (Float\ a\ b) < 0) = (a < 0)$
apply (*auto simp add*: *mult-less-0-iff Ifloat.simps*)
done

lemma *less-float-def'*: $f < g = (case\ f - g\ of\ Float\ a\ b \Rightarrow a < 0)$
proof −
have *less-transfer*: $(f < g) = (Ifloat\ (f - g) < 0)$ **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'*: $f < g = (Ifloat\ (Float\ a\ b) < 0)$ **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 *Main Fact Parity*
begin

39.1 The type of formal power series

typedef (**open**) $'a\ fps = \{f :: nat \Rightarrow 'a.\ True\}$
morphisms *fps-nth Abs-fps*
by *simp*

notation *fps-nth* (**infixl** $ 75)

lemma *expand-fps-eq*: $p = q \longleftrightarrow (\forall n.\ p\ \$\ n = q\ \$\ n)$
by (*simp add*: *fps-nth-inject* [*symmetric*] *expand-fun-eq*)

lemma *fps-ext*: $(\bigwedge n.\ p\ \$\ n = q\ \$\ n) \Longrightarrow p = q$
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 (λn. 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 (λn. 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 + = (λf g. Abs-fps (λn. 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 − = (λf g. Abs-fps (λn. 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 = (λ*f*. *Abs-fps* (λ*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 ∗ = (λ*f g*. *Abs-fps* (λ*n*. ∑ *i*=*0*..*n*. *f* $ *i* ∗ *g* $ (*n* − *i*)))

instance ..
end

lemma *fps-mult-nth*: (*f* ∗ *g*) $ *n* = (∑ *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* ∗ (*if b then y else 0*) = (*if b then x* ∗ *y else 0*)
by *simp*

lemma *cond-value-iff*: *f* (*if b then x else y*) = (*if b then f x else f y*)
by *auto*
lemma *cond-application-beta*: (*if b then f else g*) *x* = (*if b then f x 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 ⇒ nat ⇒ nat ⇒ '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 $(\sum i{=}0..n.\ f\ i\ (n - i)) = (\sum i{=}0..n.\ f\ (n - i)\ i)$
proof (*rule setsum-reindex-cong*)
  show *inj-on* $(\lambda i.\ n - i)\ \{0..n\}$
    by (*rule inj-onI*) *simp*
  show $\{0..n\} = (\lambda i.\ n - i)\ `\ \{0..n\}$
    by (*auto*, *rule-tac x*=*n - x* **in** *image-eqI*, *simp-all*)
next
  fix *i* **assume** $i \in \{0..n\}$
  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 ≠ 0*
assume $f \neq 0$
then have $\exists n.\ f\ \$\ n \neq 0$
by (*simp add*: *fps-nonzero-nth*)
then have *f $?n ≠ 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-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-const* (*c*::'*a*::*semiring-0*) = *Abs-fps* (λ*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-const* (*c*::'*a*::*semiring-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-const* (*c*::'*a*::*semiring-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* ≠ *0* **and** *b0*: *b* ≠ *0*
then obtain *i j* **where** *i*: *a$i*≠*0* ∀ *k*<*i*. *a$k*=*0*
and *j*: *b$j* ≠*0* ∀ *k*<*j*. *b$k* =*0* **unfolding** *fps-nonzero-nth-minimal*
by *blast+*
have (*a* ∗ *b*) $ (*i*+*j*) = (∑ *k*=*0*..*i*+*j*. *a$k* ∗ *b$*(*i*+*j*−*k*))
by (*rule fps-mult-nth*)
also have ... = (*a$i* ∗ *b$*(*i*+*j*−*i*)) + (∑ *k*∈{*0*..*i*+*j*}−{*i*}. *a$k* ∗ *b$*(*i*+*j*−*k*))
by (*rule setsum-diff1'*) *simp-all*
also have (∑ *k*∈{*0*..*i*+*j*}−{*i*}. *a$k* ∗ *b$*(*i*+*j*−*k*)) = *0*
proof (*rule setsum-0'* [*rule-format*])
fix *k* **assume** *k* ∈ {*0*..*i*+*j*} − {*i*}
then have *k* < *i* ∨ *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* ≠ *0* **using** *i j* **by** *simp*
finally have (*a*∗*b*) $ (*i*+*j*) ≠ *0* **.**
then show *a*∗*b* ≠ *0* **unfolding** *fps-nonzero-nth* **by** *blast*
qed

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
by (*intro-classes*, *rule number-of-fps-def*)
end

39.6 Inverses of formal power series

declare *setsum-cong*[*fundef-cong*]

instantiation *fps* :: ({*comm-monoid-add*,*inverse*, *times*, *uminus*}) *inverse*
begin

fun *natfun-inverse*:: *'a fps* ⇒ *nat* ⇒ *'a* **where**
natfun-inverse f 0 = *inverse* (*f$0*)
| *natfun-inverse f n* = − *inverse* (*f$0*) ∗ *setsum* (λ*i*. *f$i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*}

definition *fps-inverse-def*:
inverse f = (*if f$0* = *0 then 0 else Abs-fps* (*natfun-inverse f*))
definition *fps-divide-def*: *divide* = (λ(*f*::'*a fps*) *g*. *f* ∗ *inverse g*)
instance ..
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*)

instance *fps* :: ({*comm-monoid-add*,*inverse*, *times*, *uminus*}) *division-by-zero*
by *default* (*rule fps-inverse-zero*)

lemma *inverse-mult-eq-1*[*intro*]: **assumes** *f0*: *f$0* ≠ (*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*: ⋀*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*$*n*) =
(*setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*}) / (*f*$*0*)
by (*cases n*, *simp*, *simp add*: *divide-inverse fps-inverse-def*)
from *th0*[*symmetric*, *unfolded nonzero-divide-eq-eq*[*OF f0*]]
have *th1*: *setsum* (λ*i*. *f*$*i* ∗ *natfun-inverse f* (*n* − *i*)) {*1*..*n*} =
− (*f*$*0*) ∗ (*inverse f*)$*n*
by (*simp add*: *ring-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* ≠ (*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*(λ*n*. (*1*::'*a*::*field*)))
= *Abs-fps* (λ*n*. *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* = λ*i*. *if n* = *i then* (*1*::'*a*) *else if n* − *i* = *1 then* − *1 else 0*
let *?g* = λ*i*. *if i* = *n then 1 else if i*=*n* − *1 then* − *1 else 0*
let *?h* = λ*i*. *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

39.7 Formal Derivatives, and the MacLaurin theorem around 0

definition *fps-deriv f* = *Abs-fps* (λ*n*. *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*: *ring-simps*)

lemma *fps-deriv-mult*[*simp*]:
fixes *f* :: (′*a* :: *comm-ring-1*) *fps*
shows *fps-deriv* (*f* ∗ *g*) = *f* ∗ *fps-deriv g* + *fps-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 ring-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-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* (λ*i*. *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*: *ring-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*: *ring-simps of-nat-mult*)

39.8 Powers

instantiation *fps* :: (*semiring-1*) *power*
begin

fun *fps-pow* :: *nat* ⇒ '*a fps* ⇒ '*a fps* **where**
fps-pow 0 f = *1*
| *fps-pow* (*Suc n*) *f* = *f* ∗ *fps-pow n f*

definition *fps-power-def*: *power* (*f*::'*a fps*) *n* = *fps-pow n f*
instance ..
end

instantiation *fps* :: (*comm-ring-1*) *recpower*
begin
instance
apply (*intro-classes*)
by (*simp-all add*: *fps-power-def*)
end

lemma *fps-power-zeroth-eq-one*: *a*$*0* =*1* ⟹ *a*^*n* $ *0* = (*1*::'*a*::*semiring-1*)
by (*induct n*, *auto simp add*: *fps-power-def 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*$*1*
proof(*induct n*)
case *0* **thus** *?case* **by** (*simp add*: *fps-power-def*)
next
case (*Suc n*)
note *h* = *Suc.hyps*[*OF* ‹*a*$*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*: *ring-simps*)
qed

lemma *startsby-one-power*:*a* $ *0* = (*1*::'*a*::*comm-ring-1*) ⟹ *a*^*n* $ *0* = *1*
by (*induct n*, *auto simp add*: *fps-power-def 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-power-def fps-mult-nth*)

lemma *startsby-power*:*a* $*0* = (*v*::'*a*::{*comm-ring-1*, *recpower*}) ⟹ *a*^*n* $*0* = *v*^*n*

by (*induct n*, *auto simp add*: *fps-power-def fps-mult-nth power-Suc*)

lemma *startsby-zero-power-iff* [*simp*]:
a ^ n $ 0 = (0::'a::{idom, recpower}) ⟷ (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::{recpower, 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)$(Suc n) by (simp add: ring-simps
power-Suc)
  also have ... = setsum (λi. a^n$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, recpower}) 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 ring-simps fps-const-add[symmetric]
```

simp del: *fps-const-add*)
by (*case-tac n*, *auto simp add*: *power-Suc ring-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*: *ring-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*: *ring-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*: *ring-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 ring-simps power2-eq-square fps-inverse-mult inverse-mult-eq-1'*[*OF a0*])

39.9 The eXtractor series X

lemma *minus-one-power-iff*: (− (*1*::*'a* :: {*recpower*, *comm-ring-1*})) ^ *n* = (*if even n then 1 else* − *1*)
by (*induct n*, *auto*)

definition *X* = *Abs-fps* (λ*n*. *if n* = *1 then 1 else 0*)

lemma *fps-inverse-gp'*: *inverse* (*Abs-fps*(λ*n*. (*1*::*'a*::*field*)))
= *1* − *X*
by (*simp add*: *fps-inverse-gp fps-eq-iff X-def*)

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-power-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*)
lemma *fps-deriv-X*[*simp*]: *fps-deriv X = 1*
by (*simp add*: *fps-deriv-def X-def fps-eq-iff*)

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 *X-nth*[*simp*]: *X*$*n* = (*if n = 1 then 1 else 0*) **by** (*simp add*: *X-def*)
lemma *X-power-nth*[*simp*]: (*X^k*) $*n* = (*if n = k then 1 else* (*0::'a::comm-ring-1*))
by (*simp add*: *X-power-iff*)

lemma *fps-inverse-X-plus1*:
inverse (*1 + X*) = *Abs-fps* (*λn*. (− (*1::'a::*{*recpower*, *field*})) ^ *n*) (**is** - = *?r*)
proof−
have *eq*: (*1 + X*) ∗ *?r* = *1*
unfolding *minus-one-power-iff*
apply (*auto simp add*: *ring-simps fps-eq-iff*)
by *presburger*+
show *?thesis* **by** (*auto simp add*: *eq intro*: *fps-inverse-unique*)
qed

39.10 Integration

definition *fps-integral a a0 = Abs-fps* (*λn. if n = 0 then a0 else* (*a$*(*n − 1*) */ of-nat n*))

lemma *fps-deriv-fps-integral*: *fps-deriv* (*fps-integral a* (*a0 :: 'a :: {field, ring-char-0}*)) *= a*
by (*simp add*: *fps-integral-def fps-deriv-def fps-eq-iff field-simps del*: *of-nat-Suc*)

lemma *fps-integral-linear*: *fps-integral* (*fps-const* (*a::'a::{field, ring-char-0}*) *∗ f + fps-const b ∗ g*) (*a∗a0 + b∗b0*) *= fps-const a ∗ fps-integral f a0 + fps-const b ∗ fps-integral g b0* (**is** *?l = ?r*)
proof−
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

39.11 Composition of FPSs

definition *fps-compose* :: (*'a::semiring-1*) *fps ⇒ 'a fps ⇒ 'a fps* (**infixl** *oo 55*)
where
fps-compose-def: *a oo b = Abs-fps* (*λn. setsum* (*λi. a$i ∗* (*b ^ i$n*)) {*0..n*})

lemma *fps-compose-nth*: (*a oo b*)*$n = setsum* (*λi. a$i ∗* (*b ^ i$n*)) {*0..n*} **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 *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)

39.12 Rules from Herbert Wilf's Generatingfunctionology

39.12.1 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:: field*) *∗ 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* (λ*i*. *fps-const* (*a i* :: '*a*:: *field*) ∗ *X*^*i*) {*0* .. *Suc k*})$*n* = (*Abs-fps a* − *setsum* (λ*i*. *fps-const* (*a i* :: '*a*:: *field*) ∗ *X*^*i*) {*0* .. *k*} − *fps-const* (*a* (*Suc k*)) ∗ *X*^ *Suc k*) $ *n* **by** (*simp add*: *ring-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

39.12.2 Rule 2

definition *XD* = *op* ∗ *X o fps-deriv*

lemma *XD-add*[*simp*]:*XD* (*a* + *b*) = *XD a* + *XD* (*b* :: ('*a*::*comm-ring-1*) *fps*)
by (*simp add*: *XD-def ring-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 ring-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*, *recpower*, *ring-char-0*}) *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 ring-simps del*: *One-nat-def*)

39.12.3 Rule 3 is trivial and is given by *fps-times-def*

39.12.4 Rule 5 — summation and "division" by (1 - X)

lemma *fps-divide-X-minus1-setsum-lemma*:
a = ((*1*::('*a*::*comm-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*}
apply (*simp-all add*: *expand-set-eq*) **by** *presburger*+
have *d*: {*0*} ∩ ({*1*} ∪ {*2*..*n*}) = {} {*1*} ∩ {*2*..*n*} = {}
{*0*..*n* − *1*}∩{*n*} ={} **using** *n0*
by (*simp-all add*: *expand-set-eq*, *presburger*+)
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-fps* (λ*n*. *setsum* (λ*i*. *a* $ *i*) {*0*..*n*})
proof−
let *?X* = *1* − (*X*::('*a*::*field*) *fps*)
have *th0*: *?X* $ *0* ≠ *0* **by** *simp*
have *a* /*?X* = *?X* ∗ *Abs-fps* (λ*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-fps* (λ*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

39.12.5 Rule 4 in its more general form: generalizes Rule 3 for an arbitrary finite product of FPS, also the relvant instance of powers of a FPS

definition *natpermute n k* = {*l*:: *nat list*. *length l* = *k* ∧ *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* ∀*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
    unfolding image-Collect[symmetric]
    unfolding Collect-def mem-def
    apply (rule finite-imageI)
    apply blast
    done
qed

lemma natpermute-contain-maximal:
  {xs ∈ natpermute n (k+1). n ∈ set xs} = UNION {0 .. k} (λi. {(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*: $i \in \{0..\ k\}$ $xs!i = n$ **using** *H*
unfolding *in-set-conv-nth* **by** (*auto simp add*: *less-Suc-eq-le natpermute-def*)
have *eqs*: $(\{0..k\} - \{i\}) \cup \{i\} = \{0..k\}$ **using** *i* **by** *auto*
have *f*: *finite*$(\{0..k\} - \{i\})$ *finite* $\{i\}$ **by** *auto*
have *d*: $(\{0..k\} - \{i\}) \cap \{i\} = \{\}$ **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 $\ldots = n + setsum\ (nth\ xs)\ (\{0..k\} - \{i\})$
unfolding *setsum-Un-disjoint*[*OF f d*, *unfolded eqs*] **using** *i* **by** *simp*
finally have *zxs*: $\forall\ j \in \{0..k\} - \{i\}.\ xs!j = 0$ **by** *auto*
from *H* **have** *xsl*: *length xs* $= k+1$ **by** (*simp add*: *natpermute-def*)
from *i* **have** *i'*: $i < length\ (replicate\ (k+1)\ 0) \quad i < k+1$
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* $\in$ *?B* **using** *i* **by** *blast*}
moreover
{**fix** *i* **assume** *i*: $i \in \{0..k\}$
let *?xs* = *replicate* $(k+1)$ *0* [*i*:=*n*]
have *nxs*: $n \in set\ ?xs$
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 $\ldots = setsum\ (\lambda j.\ if\ j = i\ then\ n\ else\ 0)\ \{0..<\ k+1\}$
apply (*rule setsum-cong2*) **by** (*simp del*: *replicate.simps*)
also have $\ldots = n$ **using** *i* **by** (*simp add*: *setsum-delta*)
finally
have *?xs* $\in$ *natpermute n* $(k+1)$ **using** *xsl* **unfolding** *natpermute-def Collect-def mem-def*
by *blast*
then have *?xs* $\in$ *?A* **using** *nxs* **by** *blast*}
ultimately show *?thesis* **by** *auto*
qed

lemma *fps-setprod-nth*:
fixes *m* :: *nat* **and** *a* :: *nat* $\Rightarrow$ ('*a*::*comm-ring-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} ∪ {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} ∩ {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 ... = (∑ i = 0..n. (∑ v∈natpermute i (k + 1). ∏ j∈{0..k}. a j $
v ! j) * a m $ (n − i))
     unfolding fps-mult-nth H[rule-format, OF km] ..
   also have ... = (∑ v∈natpermute n (m + 1). ∏ j∈{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=λxs. 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-ring-1) fps
  shows (a ^ Suc m)$n = setsum (λv. setprod (λj. a $ (v!j)) {0..m}) (natpermute
n (m+1))
proof−
  have f: finite {0 ..m} by simp
  have th0: a^Suc m = setprod (λi. 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-ring-1*) *fps*
shows (*a* ^*m*)$*n* = (*if m=0 then 1$n else setsum* (λ*v*. *setprod* (λ*j*. *a* $ (*v*!*j*)) {*0..m* − *1*}) (*natpermute n m*))
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-ring-1*, *recpower*}) *fps*
shows (*a* ^*m*)$*0* = (*a*$*0*) ^ *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* ^*m*)$*0* = *setprod* (λ*i*. *a*$*0*) {*0..n*}
by (*simp add*: *m fps-power-nth del*: *replicate.simps power-Suc*)
also have ... = (*a*$*0*) ^ *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*$*0* = (*0*::*'a*::{*recpower*,*idom*})
and *a1*: *a*$*1* ≠ *0*
shows (*b oo a* = *c oo a*) ⟷ *b* = *c* (**is** *?lhs* ⟷*?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*: ∀ *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*: $(c\ oo\ a)\ \$n = (\sum i = 0..n1.\ c\ \$\ i * a\ \hat{}\ i\ \$\ n) + c\$n * (a\$1)\ \hat{}n$
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

39.13 Radicals

declare *setprod-cong*[*fundef-cong*]
function *radical* :: $(nat \Rightarrow 'a \Rightarrow 'a) \Rightarrow nat \Rightarrow ('a::\{field, recpower\})\ fps \Rightarrow nat \Rightarrow 'a$ **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* (λ*xs*. *setprod* (λ*j*. *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≠0
  shows card {xs ∈ natpermute n (k+1). n ∈ set xs} = k+1
  unfolding natpermute-contain-maximal
proof−
  let ?A= λi. {replicate (k + 1) 0[i := n]}
```

let *?K* = {*0* ..*k*}
have *fK*: *finite ?K* **by** *simp*
have *fAK*: ∀ *i*∈*?K*. *finite* (*?A i*) **by** *auto*
have *d*: ∀ *i*∈ *?K*. ∀ *j*∈ *?K*. *i* ≠ *j* ⟶ {*replicate* (*k* + *1*) *0*[*i* := *n*]} ∩ {*replicate* (*k* + *1*) *0*[*j* := *n*]} = {}
proof(*clarify*)
fix *i j* **assume** *i*: *i* ∈ *?K* **and** *j*: *j*∈ *?K* **and** *ij*: *i*≠*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*]} ∩ {*replicate* (*k* + *1*) *0*[*j* := *n*]} = {}
by *auto*
qed
from *card-UN-disjoint*[*OF fK fAK d*]
show *card* (⋃ *i*∈{*0*..*k*}. {*replicate* (*k* + *1*) *0*[*i* := *n*]}) = *k*+*1* **by** *simp*
qed

lemma *power-radical*:
fixes *a*:: *'a* ::{*field*, *ring-char-0*, *recpower*} *fps*
assumes *r0*: (*r* (*Suc k*) (*a*$*0*)) ^ *Suc k* = *a*$*0* **and** *a0*: *a*$*0* ≠ *0*
shows (*fps-radical r* (*Suc k*) *a*) ^ (*Suc k*) = *a*
proof−
let *?r* = *fps-radical r* (*Suc k*) *a*
from *a0 r0* **have** *r00*: *r* (*Suc k*) (*a*$*0*) ≠ *0* **by** *auto*
{fix *z* **have** *?r* ^ *Suc k* $ *z* = *a*$*z*
proof(*induct z rule*: *nat-less-induct*)
fix *n* **assume** *H*: ∀ *m*<*n*. *?r* ^ *Suc k* $ *m* = *a*$*m*
{assume *n* = *0* **hence** *?r* ^ *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* ≠ *0* **using** *n1* **by** *arith*
let *?Pnk* = *natpermute n* (*k* + *1*)
let *?Pnkn* = {*xs* ∈ *?Pnk*. *n* ∈ *set xs*}
let *?Pnknn* = {*xs* ∈ *?Pnk*. *n* ∉ *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* = λ*v*. ∏ *j*∈{*0*..*k*}. *?r* $ *v* ! *j*
have *setsum ?f ?Pnkn* = *setsum* (λ*v*. *?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 show ?thesis by (simp add: fps-eq-iff)
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)) ^ Suc k = b$0
  and a0: r (Suc k) (b$0 ::'a::{field, ring-char-0, recpower}) = a$0 and b0: b$0
≠ 0
  shows a^(Suc k) = b ⟷ a = fps-radical r (Suc k) b
proof−
  let ?r = fps-radical r (Suc k) b
  have r00: r (Suc k) (b$0) ≠ 0 using b0 r0 by auto
  {assume H: a = ?r
    from H have a^Suc k = b using power-radical[of r k, OF r0 b0] by simp}
  moreover
  {assume H: a^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*: ∀ *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 ≠ 0* **using** *n1* **by** *arith*
let *?Pnk = natpermute n* (*Suc k*)
let *?Pnkn* = {*xs ∈ ?Pnk*. *n ∈ set xs*}
let *?Pnknn* = {*xs ∈ ?Pnk*. *n ∉ 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* = λ*v*. ∏*j*∈{*0..k*}. *?r $ v ! j*
let *?g* = λ*v*. ∏*j*∈{*0..k*}. *a $ v ! j*
have *setsum ?g ?Pnkn = setsum* (λ*v*. *a $ n ∗* (*?r$0*) ^*k*) *?Pnkn*
proof(*rule setsum-cong2*)
fix *v* **assume** *v*: *v ∈* {*xs ∈ natpermute n* (*Suc k*). *n ∈ set xs*}
let *?ths* = (∏*j*∈{*0..k*}. *a $ v ! j*) = *a $ n ∗* (*?r$0*) ^*k*
from *v* **obtain** *i* **where** *i*: *i ∈* {*0..k*} *v = replicate* (*k+1*) *0* [*i*:= *n*]
unfolding *Suc-plus1 natpermute-contain-maximal* **by** (*auto simp del*: *replicate.simps*)
have (∏*j*∈{*0..k*}. *a $ v ! j*) = (∏*j*∈{*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 ... = *a $ n ∗* (*?r $ 0*) ^*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-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, ring-char-0, recpower}) ≠ 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*, *ring-char-0*, *recpower*} *fps*
assumes *r0*: *(r (Suc k) (a$0)) ^ Suc k = a$0* **and** *a0*: *a$0 ≠ 0*
shows *fps-deriv (fps-radical r (Suc k) a) = fps-deriv a / (fps-const (of-nat (Suc k)) * (fps-radical r (Suc k) a) ^ k)*
proof−
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 *power-radical*[*of r*, *OF r0 a0*]
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*, *ring-char-0*, *recpower*} *fps*
assumes
ra0: *r (k) (a $ 0) ^ k = a $ 0*
and *rb0*: *r (k) (b $ 0) ^ k = b $ 0*
and *r0'*: *r (k) ((a * b) $ 0) = r (k) (a $ 0) * r (k) (b $ 0)*
and *a0*: *a$0 ≠ 0*
and *b0*: *b$0 ≠ 0*
shows *fps-radical r (k) (a*b) = fps-radical r (k) a * fps-radical r (k) (b)*
proof−
from *r0'* **have** *r0*: *(r (k) ((a*b)$0)) ^ k = (a*b)$0*
by (*simp add*: *fps-mult-nth ra0 rb0 power-mult-distrib*)
{assume *k=0* **hence** *?thesis* **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 ≠ 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]
    power-radical[of r, OF ra0[unfolded k] a0] power-radical[of r, OF rb0[unfolded
k] b0] k
  have ?thesis by (auto simp add: power-mult-distrib simp del: power-Suc)}
ultimately show ?thesis by (cases k, auto)
qed

lemma radical-inverse:
  fixes a:: 'a ::{field, ring-char-0, recpower} fps
  assumes
  ra0: r (k) (a $ 0) ^ k = a $ 0
  and ria0: r (k) (inverse (a $ 0)) = inverse (r (k) (a $ 0))
  and r1: (r (k) 1) = 1
  and a0: a$0 ≠ 0
  shows fps-radical r (k) (inverse a) = inverse (fps-radical r (k) a)
proof−
  {assume k=0 then have ?thesis by simp}
  moreover
  {fix h assume k[simp]: k = Suc h
    let ?ra = fps-radical r (Suc h) a
    let ?ria = fps-radical r (Suc h) (inverse a)
    from ra0 a0 have th00: r (Suc h) (a$0) ≠ 0 by auto
    have ria0': r (Suc h) (inverse a $ 0) ^ Suc h = inverse a$0
    using ria0 ra0 a0
    by (simp add: fps-inverse-def  nonzero-power-inverse[OF th00, symmetric]
          del: power-Suc)
  from inverse-mult-eq-1[OF a0] have th0: a * inverse a = 1
    by (simp add: mult-commute)
  from radical-unique[where a=1 and b=1 and r=r and k=h, simplified, OF
r1[unfolded k]]
  have th01: fps-radical r (Suc h) 1 = 1 .
  have th1: r (Suc h) ((a * inverse a) $ 0) ^ Suc h = (a * inverse a) $ 0
    r (Suc h) ((a * inverse a) $ 0) =
r (Suc h) (a $ 0) * r (Suc h) (inverse a $ 0)
    using r1 unfolding th0  apply (simp-all add: ria0[symmetric])
    apply (simp add: fps-inverse-def a0)
    unfolding ria0[unfolded k]
    using th00 by simp
  from nonzero-imp-inverse-nonzero[OF a0] a0
  have th2: inverse a $ 0 ≠ 0 by (simp add: fps-inverse-def)
  from radical-mult-distrib[of r Suc h a inverse a, OF ra0[unfolded k] ria0' th1(2)
a0 th2]
  have th3: ?ra * ?ria = 1 unfolding th0 th01 by simp
  from th00 have ra0: ?ra $ 0 ≠ 0 by simp
  from fps-inverse-unique[OF ra0 th3] have ?thesis by simp}
ultimately show ?thesis by (cases k, auto)
```

qed

lemma *fps-divide-inverse*: $(a::('a::field)\ fps)\ /\ b = a * inverse\ b$
by (*simp add*: *fps-divide-def*)

lemma *radical-divide*:
fixes *a*:: *'a* ::{*field*, *ring-char-0*, *recpower*} *fps*
assumes
ra0: $r\ k\ (a\ \$\ 0)\ \hat{}\ k = a\ \$\ 0$
and *rb0*: $r\ k\ (b\ \$\ 0)\ \hat{}\ k = b\ \$\ 0$
and *r1*: $r\ k\ 1 = 1$
and *rb0'*: $r\ k\ (inverse\ (b\ \$\ 0)) = inverse\ (r\ k\ (b\ \$\ 0))$
and *raib'*: $r\ k\ (a\$0\ /\ (b\$0)) = r\ k\ (a\$0)\ /\ r\ k\ (b\$0)$
and *a0*: $a\$0 \neq 0$
and *b0*: $b\$0 \neq 0$
shows *fps-radical r k* (a/b) = *fps-radical r k a / fps-radical r k b*
proof−
from *raib'*
have *raib*: $r\ k\ (a\$0\ /\ (b\$0)) = r\ k\ (a\$0) * r\ k\ (inverse\ (b\$0))$
by (*simp add*: *divide-inverse rb0'*[*symmetric*])

{assume *k=0* **hence** *?thesis* **by** (*simp add*: *fps-divide-def*)**}**
moreover
{assume *k0*: $k \neq 0$
from *b0 k0 rb0* **have** *rbn0*: $r\ k\ (b\ \$0) \neq 0$
by (*auto simp add*: *power-0-left*)

from *rb0 rb0'* **have** *rib0*: $(r\ k\ (inverse\ (b\ \$\ 0)))\hat{}k = inverse\ (b\$0)$
by (*simp add*: *nonzero-power-inverse*[*OF rbn0*, *symmetric*])
from *rib0* **have** *th0*: $r\ k\ (inverse\ b\ \$\ 0)\ \hat{}\ k = inverse\ b\ \$\ 0$
by (*simp add*:*fps-inverse-def b0*)
from *raib*
have *th1*: $r\ k\ ((a * inverse\ b)\ \$\ 0) = r\ k\ (a\ \$\ 0) * r\ k\ (inverse\ b\ \$\ 0)$
by (*simp add*: *divide-inverse fps-inverse-def b0 fps-mult-nth*)
from *nonzero-imp-inverse-nonzero*[*OF b0*] *b0* **have** *th2*: $inverse\ b\ \$\ 0 \neq 0$
by (*simp add*: *fps-inverse-def*)
from *radical-mult-distrib*[*of r k a inverse b*, *OF ra0 th0 th1 a0 th2*]
have *th*: *fps-radical r k* (a/b) = *fps-radical r k a* $*$ *fps-radical r k* $(inverse\ b)$
by (*simp add*: *fps-divide-def*)
with *radical-inverse*[*of r k b*, *OF rb0 rb0' r1 b0*]
have *?thesis* **by** (*simp add*: *fps-divide-def*)**}**
ultimately show *?thesis* **by** *blast*
qed

39.14 Derivative of composition

lemma *fps-compose-deriv*:
fixes *a*:: (*'a*::*idom*) *fps*
assumes *b0*: $b\$0 = 0$

shows *fps-deriv* (*a oo b*) = ((*fps-deriv a*) *oo b*) ∗ (*fps-deriv b*)
proof−
{fix *n*
have (*fps-deriv* (*a oo b*))$*n* = *setsum* (λ*i*. *a* $ *i* ∗ (*fps-deriv* (*b* ^ *i*))$*n*) {*0*.. *Suc n*}
by (*simp add*: *fps-compose-def ring-simps setsum-right-distrib del*: *of-nat-Suc*)
also have ... = *setsum* (λ*i*. *a*$*i* ∗ ((*fps-const* (*of-nat i*)) ∗ (*fps-deriv b* ∗ (*b*^(*i* − *1*))))$*n*) {*0*.. *Suc n*}
by (*simp add*: *ring-simps fps-deriv-power del*: *fps-mult-left-const-nth of-nat-Suc*)
also have ... = *setsum* (λ*i*. *of-nat i* ∗ *a*$*i* ∗ (((*b*^(*i* − *1*)) ∗ *fps-deriv b*))$*n*) {*0*.. *Suc n*}
unfolding *fps-mult-left-const-nth* **by** (*simp add*: *ring-simps*)
also have ... = *setsum* (λ*i*. *of-nat i* ∗ *a*$*i* ∗ (*setsum* (λ*j*. (*b* ^ (*i* − *1*))$*j* ∗ (*fps-deriv b*)$(*n* − *j*)) {*0*..*n*})) {*0*.. *Suc n*}
unfolding *fps-mult-nth* **..**
also have ... = *setsum* (λ*i*. *of-nat i* ∗ *a*$*i* ∗ (*setsum* (λ*j*. (*b* ^ (*i* − *1*))$*j* ∗ (*fps-deriv b*)$(*n* − *j*)) {*0*..*n*})) {*1*.. *Suc n*}
apply (*rule setsum-mono-zero-right*)
apply (*auto simp add*: *mult-delta-left setsum-delta not-le*)
done
also have ... = *setsum* (λ*i*. *of-nat* (*i* + *1*) ∗ *a*$(*i*+*1*) ∗ (*setsum* (λ*j*. (*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*))$*n* = *setsum* (λ*i*. *of-nat* (*i* + *1*) ∗ *a*$(*i*+*1*) ∗ (*setsum* (λ*j*. (*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*))$*n* = *setsum* (λ*i*. (*fps-deriv b*)$ (*n* − *i*) ∗ ((*fps-deriv a*) *oo b*)$*i*) {*0*..*n*}
unfolding *fps-mult-nth* **by** (*simp add*: *mult-ac*)
also have ... = *setsum* (λ*i*. *setsum* (λ*j*. *of-nat* (*n* − *i* +*1*) ∗ *b*$(*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*$*x* = *0*)
apply *simp*
apply (*rule startsby-zero-power-prefix*[*OF b0*, *rule-format*])
by *simp*
also have ... = *setsum* (λ*i*. *of-nat* (*i* + *1*) ∗ *a*$(*i*+*1*) ∗ (*setsum* (λ*j*. (*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

39.15 Finite FPS (i.e. polynomials) and X

lemma *fps-poly-sum-X*:
assumes *z*: ∀ *i* > *n*. *a$i* = (*0*::*'a*::*comm-ring-1*)
shows *a* = *setsum* (λi. *fps-const* (*a$i*) * *X*^*i*) {*0..n*} (**is** *a* = *?r*)
proof−
{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

39.16 Compositional inverses

fun *compinv* :: *'a fps* ⇒ *nat* ⇒ *'a*::{*recpower*,*field*} **where**
compinv a 0 = *X$0*
| *compinv a* (*Suc n*) = (*X$ Suc n* − *setsum* (λi. (*compinv a i*) * (*a*^*i*)$*Suc n*) {*0 .. n*}) / (*a$1*) ^ *Suc n*

definition *fps-inv a* = *Abs-fps* (*compinv a*)

lemma *fps-inv*: **assumes** *a0*: *a$0* = *0* **and** *a1*: *a$1* ≠ *0*
shows *fps-inv a oo a* = *X*
proof−
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::{recpower,field} where
  gcompinv b a 0 = b$0
| gcompinv b a (Suc n) = (b$ Suc n − setsum (λi. (gcompinv b a i) * (a^i)$Suc
n) {0 .. n}) / (a$1) ^ Suc n

definition fps-ginv b a = Abs-fps (gcompinv b a)

lemma fps-ginv: assumes a0: a$0 = 0 and a1: a$1 ≠ 0
  shows fps-ginv b a oo a = b
proof−
  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 fps-power-def 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-pow-0: fps-pow n 0 = (if n = 0 then 1 else 0)
 by (induct n, simp-all)

lemma fps-compose-0-right[simp]: a oo 0 = fps-const (a$0)
 by (auto simp add: fps-eq-iff fps-compose-nth fps-power-def fps-pow-0 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  ring-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).
```

$i <= n \wedge j \leq n \wedge i + j = n\}$
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*: *ring-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 $(a\ oo\ c)\hat{}n = a\hat{}n\ oo\ c$ (**is** *?l* = *?r*)
proof−
{assume *n=0* **then have** *?thesis* **by** *simp***}**
moreover
{fix *m* **assume** *m*: $n = Suc\ m$
have *th0*: $a\hat{}n = setprod\ (\%k.\ a)\ \{0..m\}\ (a\ oo\ c)\ \hat{}\ n = setprod\ (\%k.\ a\ oo\ c)\ \{0..m\}$
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-const-mult-apply-left*:
$fps\text{-}const\ c * (a\ oo\ b) = (fps\text{-}const\ c * a)\ oo\ b$
by (*simp add*: *fps-eq-iff fps-compose-nth setsum-right-distrib mult-assoc*)

lemma *fps-const-mult-apply-right*:
$(a\ oo\ b) * fps\text{-}const\ (c::'a::comm\text{-}semiring\text{-}1) = (fps\text{-}const\ c * a)\ oo\ b$
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 $?l\$n = (setsum\ (\lambda i.\ (fps\text{-}const\ (a\$i) * b\hat{}i)\ oo\ c)\ \{0..n\})\$n$
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 $\ldots = ((setsum\ (\lambda i.\ fps\text{-}const\ (a\$i) * b\hat{}i)\ \{0..n\})\ oo\ c)\$n$
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::{recpower,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

39.17 Elementary series

39.17.1 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*, *recpower*, *ring-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 ring-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*,
ring-char-0, *recpower*})
(**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*
apply (*simp add*: *field-simps del*: *of-nat-Suc fact.simps*)
apply (*drule sym*)
by (*simp add*: *ring-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*::{*ring-char-0*, *field*, *recpower*}) ∗ *E b*
(**is** *?l* = *?r*)
proof−
have *fps-deriv* (*?r*) = *fps-const* (*a*+*b*) ∗ *?r*
by (*simp add*: *fps-const-add*[*symmetric*] *ring-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*, *recpower*}) = *1*
by (*simp add*: *fps-eq-iff power-0-left*)

lemma *E-neg*: *E* (− *a*) = *inverse* (*E* (*a*::'*a*::{*ring-char-0*, *field*, *recpower*}))
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*, *recpower*, *ring-char-0*}))
= (*fps-const a*)^*n* ∗ (*E a*)
by (*induct n*, *auto simp add*: *power-Suc*)

lemma *fps-compose-uminus*: − (*a*::'*a*::*ring-1 fps*) *oo c* = − (*a oo c*)
by (*simp add*: *fps-eq-iff fps-compose-nth ring-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 *X-compose-E*[*simp*]: *X oo E* (*a*::'*a*::{*field*, *recpower*}) = *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*:
inverse (*fps-const* (*a*::'*a*::{*field*, *division-by-zero*})) = *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*, *recpower*}) ^*n*)
(**is** *inverse ?l* = *?r*)
proof−
have *th*: *?l* ∗ *?r* = *1*
apply (*auto simp add*: *ring-simps fps-eq-iff X-mult-nth minus-one-power-iff*)
apply *presburger*+
done
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*,*recpower*,*ring-char-0*})) ^*n* = *E* (*of-nat n* ∗ *c*)
by (*induct n*, *auto simp add*: *ring-simps E-add-mult power-Suc*)

39.17.2 Logarithmic series

definition (*L*::'*a*::{*field*, *ring-char-0*,*recpower*} *fps*)
= *Abs-fps* (λ*n*. (− *1*) ^ *Suc n* / *of-nat n*)

lemma *fps-deriv-L*: *fps-deriv L* = *inverse* (*1* + *X*)
unfolding *inverse-one-plus-X*
by (*simp add*: *L-def fps-eq-iff power-Suc del*: *of-nat-Suc*)

lemma *L-nth*: *L* $ *n* = (− *1*) ^ *Suc n* / *of-nat n*
by (*simp add*: *L-def*)

lemma *L-E-inv*:
assumes *a*: *a*≠ (*0*::*'a*::{*field*,*division-by-zero*,*ring-char-0*,*recpower*})
shows *L* = *fps-const a* ∗ *fps-inv* (*E a* − *1*) (**is** *?l* = *?r*)
proof−
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*: *ring-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*: *ring-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*)
by (*simp add*: *fps-const-inverse eq fps-divide-def fps-inverse-mult*)
hence *fps-deriv* (*fps-const a* ∗ *fps-inv ?b*) = *inverse* (*X* + *1*)
using *a* **by** (*simp add*: *fps-divide-def field-simps*)
hence *fps-deriv ?l* = *fps-deriv ?r*
by (*simp add*: *fps-deriv-L add-commute*)
then show *?thesis* **unfolding** *fps-deriv-eq-iff*
by (*simp add*: *L-nth fps-inv-def*)
qed

39.17.3 Formal trigonometric functions

definition *fps-sin* (*c*::*'a*::{*field*, *recpower*, *ring-char-0*}) =
Abs-fps (λ*n*. *if even n then 0 else* (− *1*) ^((*n* − *1*) *div 2*) ∗ *c*^*n* /(*of-nat* (*fact n*)))

definition *fps-cos* (*c*::*'a*::{*field*, *recpower*, *ring-char-0*}) = *Abs-fps* (λ*n*. *if even n then* (− *1*) ^ (*n div 2*) ∗ *c*^*n* / (*of-nat* (*fact n*)) *else 0*)

lemma *fps-sin-deriv*:
fps-deriv (*fps-sin c*) = *fps-const c* ∗ *fps-cos c*
(**is** *?lhs* = *?rhs*)
proof−
{**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 ring-simps power-Suc )}
  then have ?lhs $ n = ?rhs $ n
    by (cases even n, simp-all add: fps-deriv-def fps-sin-def fps-cos-def) }
 then show ?thesis by (auto simp add: fps-eq-iff)
qed

lemma fps-cos-deriv:
 fps-deriv (fps-cos c) = fps-const (− c)* (fps-sin c)
 (is ?lhs = ?rhs)
proof−
 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 presburger
 {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 ring-simps power-Suc)}
  then have ?lhs $ n = ?rhs $ n
   by (cases even n, simp-all add: fps-deriv-def fps-sin-def
     fps-cos-def) }
 then show ?thesis by (auto simp add: fps-eq-iff)
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: fps-power-def ring-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 fps-power-def numeral-2-eq-2 fps-mult-nth fps-cos-def
fps-sin-def)
```

finally show *?thesis* **.**
qed

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*] *ring-simps power2-eq-square del*: *fps-const-neg*)
unfolding *right-distrib*[*symmetric*]
by *simp*
qed

end

40 FuncSet: Pi and Function Sets

theory *FuncSet*
imports *Hilbert-Choice Main*
begin

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

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

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

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

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

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

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

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

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

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

40.1 Basic Properties of *Pi*

lemma *Pi-I*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B x*) ==> *f* ∈ *Pi A B*
by (*simp add*: *Pi-def*)

lemma *funcsetI*: (!!*x*. *x* ∈ *A* ==> *f x* ∈ *B*) ==> *f* ∈ *A* -> *B*
by (*simp add*: *Pi-def*)

lemma *Pi-mem*: [|*f*: *Pi A B*; *x* ∈ *A*|] ==> *f x* ∈ *B x*
by (*simp add*: *Pi-def*)

lemma *funcset-mem*: [|*f* ∈ *A* -> *B*; *x* ∈ *A*|] ==> *f x* ∈ *B*
by (*simp add*: *Pi-def*)

lemma *funcset-image*: *f* ∈ *A*→*B* ==> *f* ‘ *A* ⊆ *B*
by (*auto simp add*: *Pi-def*)

lemma *Pi-eq-empty*: ((*PI x*: *A*. *B x*) = {}) = (∃ *x*∈*A*. *B*(*x*) = {})
apply (*simp add*: *Pi-def*, *auto*)

Converse direction requires Axiom of Choice to exhibit a function picking an element from each non-empty *B x*

apply (*drule-tac x* = %*u*. *SOME y*. *y* ∈ *B u* **in** *spec*, *auto*)
apply (*cut-tac P*= %*y*. *y* ∈ *B x* **in** *some-eq-ex*, *auto*)
done

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

lemma *Pi-UNIV* [*simp*]: *A* -> *UNIV* = *UNIV*
by (*simp add*: *Pi-def*)

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: (!!*x*. *x* ∈ *A* ==> *B x* <= *C x*) ==> *Pi A B* <= *Pi A C*

by (*simp add*: *Pi-def*, *blast*)

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' <= A ==> Pi\ A\ B <= Pi\ A'\ B$
by (*simp add*: *Pi-def*, *blast*)

40.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*:
$[|\ f \in A -> B;\ g \in B -> C\ |]==> compose\ A\ g\ f \in A -> C$
by (*simp add*: *Pi-def compose-def restrict-def*)

lemma *compose-assoc*:
$[|\ f \in A -> B;\ g \in B -> C;\ h \in C -> D\ |]$
$==> compose\ A\ h\ (compose\ A\ g\ f) = compose\ A\ (compose\ B\ h\ g)\ f$
by (*simp add*: *expand-fun-eq Pi-def compose-def restrict-def*)

lemma *compose-eq*: $x \in A ==> compose\ A\ g\ f\ x = g(f(x))$
by (*simp add*: *compose-def restrict-def*)

lemma *surj-compose*: $[|\ f\ `\ A = B;\ g\ `\ B = C\ |] ==> compose\ A\ g\ f\ `\ A = C$
by (*auto simp add*: *image-def compose-eq*)

40.3 Bounded Abstraction: *restrict*

lemma *restrict-in-funcset*: $(!!x.\ x \in A ==> f\ x \in B) ==> (\lambda x{\in}A.\ f\ x) \in A -> B$
by (*simp add*: *Pi-def restrict-def*)

lemma *restrictI*: $(!!x.\ x \in A ==> f\ x \in B\ x) ==> (\lambda x{\in}A.\ f\ x) \in Pi\ A\ B$
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*:
$(!!x.\ x \in A ==> f\ x = g\ x) ==> (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
by (*simp add*: *expand-fun-eq Pi-def Pi-def restrict-def*)

lemma *inj-on-restrict-eq* [*simp*]: *inj-on* $(restrict\ f\ A)\ A = inj\text{-}on\ f\ A$
by (*simp add*: *inj-on-def restrict-def*)

lemma *Id-compose*:
$[|f \in A -> B;\ f \in extensional\ A|] ==> compose\ A\ (\lambda y{\in}B.\ y)\ f = f$
by (*auto simp add*: *expand-fun-eq compose-def extensional-def Pi-def*)

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

40.4 Bijections Between Sets

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

lemma *bij-betw-imp-funcset*: *bij-betw f A B* ⟹ *f* ∈ *A* → *B*
by (*auto simp add*: *bij-betw-def inj-on-Inv Pi-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*)

40.5 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-funcset*: *f* ' *A* = *B* ==> (λ*x*∈*B*. *Inv A f x*) : *B* -> *A*
by (*unfold Inv-def*) (*fast intro*: *restrict-in-funcset someI2*)

lemma *compose-Inv-id*:
bij-betw f A B ==> *compose A* (λ*y*∈*B*. *Inv A f y*) *f* = (λ*x*∈*A*. *x*)
apply (*simp add*: *bij-betw-def compose-def*)
apply (*rule restrict-ext*, *auto*)
apply (*erule subst*)

apply (*simp add*: *Inv-f-f*)
done

lemma *compose-id-Inv*:
f ' *A* = *B* ==> *compose B f* (λ*y*∈*B*. *Inv A f y*) = (λ*x*∈*B*. *x*)
apply (*simp add*: *compose-def*)
apply (*rule restrict-ext*)
apply (*simp add*: *f-Inv-f*)
done

40.6 Cardinality

lemma *card-inj*: [|*f* ∈ *A*→*B*; *inj-on f A*; *finite B*|] ==> *card*(*A*) ≤ *card*(*B*)
apply (*rule card-inj-on-le*)
apply (*auto simp add*: *Pi-def*)
done

lemma *card-bij*:
[|*f* ∈ *A*→*B*; *inj-on f A*;
g ∈ *B*→*A*; *inj-on g B*; *finite A*; *finite B*|] ==> *card*(*A*) = *card*(*B*)
by (*blast intro*: *card-inj order-antisym*)

end

41 Polynomial: Univariate Polynomials

theory *Polynomial*
imports *Main*
begin

41.1 Definition of type *poly*

typedef (*Poly*) *'a poly* = {*f*::*nat* ⇒ *'a*::*zero*. ∃ *n*. ∀ *i*>*n*. *f i* = *0*}
morphisms *coeff Abs-poly*
by *auto*

lemma *expand-poly-eq*: *p* = *q* ⟷ (∀ *n*. *coeff p n* = *coeff q n*)
by (*simp add*: *coeff-inject* [*symmetric*] *expand-fun-eq*)

lemma *poly-ext*: (⋀*n*. *coeff p n* = *coeff q n*) ⟹ *p* = *q*
by (*simp add*: *expand-poly-eq*)

41.2 Degree of a polynomial

definition
degree :: *'a*::*zero poly* ⇒ *nat* **where**
degree p = (*LEAST n*. ∀ *i*>*n*. *coeff p i* = *0*)

lemma *coeff-eq-0*: *degree p* $<$ *n* $\Longrightarrow$ *coeff p n* = *0*
proof −
  have *coeff p* ∈ *Poly*
    by (*rule coeff*)
  hence ∃ *n*. ∀ *i*>*n*. *coeff p i* = *0*
    unfolding *Poly-def* **by** *simp*
  hence ∀ *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* ≠ *0* $\Longrightarrow$ *n* ≤ *degree p*
  by (*erule contrapos-np*, *rule coeff-eq-0*, *simp*)

lemma *degree-le*: ∀ *i*>*n*. *coeff p i* = *0* $\Longrightarrow$ *degree p* ≤ *n*
  unfolding *degree-def* **by** (*erule Least-le*)

lemma *less-degree-imp*: *n* $<$ *degree p* $\Longrightarrow$ ∃ *i*>*n*. *coeff p i* ≠ *0*
  unfolding *degree-def* **by** (*drule not-less-Least*, *simp*)

41.3 The zero polynomial

instantiation *poly* :: (*zero*) *zero*
begin

definition
  zero-poly-def: *0* = *Abs-poly* (λ*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* ≠ *0* **shows** *coeff p* (*degree p*) ≠ *0*
proof (*cases degree p*)
  case *0*
  from ‹*p* ≠ *0*› **have** ∃ *n*. *coeff p n* ≠ *0*
    by (*simp add*: *expand-poly-eq*)
  then obtain *n* **where** *coeff p n* ≠ *0* **..**
  hence *n* ≤ *degree p* **by** (*rule le-degree*)
  with ‹*coeff p n* ≠ *0*› **and** ‹*degree p* = *0*›
  show *coeff p* (*degree p*) ≠ *0* **by** *simp*

next
case (*Suc n*)
from ‹*degree p = Suc n*› **have** *n < degree p* **by** *simp*
hence ∃*i*>*n*. *coeff p i ≠ 0* **by** (*rule less-degree-imp*)
then obtain *i* **where** *n < i* **and** *coeff p i ≠ 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*)

41.4 List-style constructor for polynomials

definition
pCons :: *'a*::*zero* ⇒ *'a poly* ⇒ *'a poly*
where
[*code del*]: *pCons a p = Abs-poly* (*nat-case a* (*coeff p*))

syntax
-poly :: *args* ⇒ *'a poly* ([:(-):])

translations
[:*x*, *xs*:] == *CONST pCons x* [:*xs*:]
[:*x*:] == *CONST pCons x 0*
[:*x*:] <= *CONST pCons x* (*-constrain 0 t*)

lemma *Poly-nat-case*: *f ∈ Poly* ⟹ *nat-case a f ∈ Poly*
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*: $\bigwedge a\ p.\ P\ p \Longrightarrow P\ (pCons\ a\ p)$
  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

41.5 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*)

41.6 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

41.7 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* ⇒ *'a*
shows $f \in Poly \Longrightarrow (\lambda n.\ -\ f\ n) \in Poly$
unfolding *Poly-def* **by** *simp*

lemma *Poly-diff*:
fixes *f g* :: *nat* ⇒ *'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*) ≤ *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* ≤ *n*; *degree q* ≤ *n*⟧ ⟹ *degree* (*p* − *q*) ≤ *n*
by (*simp add*: *diff-minus degree-add-le*)

lemma *degree-diff-less*:
⟦*degree p* < *n*; *degree q* < *n*⟧ ⟹ *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* (∑ *x*∈*A*. *p x*) *i* = (∑ *x*∈*A*. *coeff* (*p x*) *i*)
by (*cases finite A*, *induct set*: *finite*, *simp-all*)

lemma *monom-setsum*: *monom* (∑ *x*∈*A*. *a x*) *n* = (∑ *x*∈*A*. *monom* (*a x*) *n*)
by (*rule poly-ext*) (*simp add*: *coeff-setsum*)

41.8 Multiplication by a constant

definition
smult :: *'a*::*comm-semiring-0* ⇒ *'a poly* ⇒ *'a poly* **where**
smult a p = *Abs-poly* (λ*n*. *a* ∗ *coeff p n*)

lemma *Poly-smult*:
fixes *f* :: *nat* ⇒ *'a*::*comm-semiring-0*
shows *f* ∈ *Poly* ⟹ (λ*n*. *a* ∗ *f n*) ∈ *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*)

41.9 Multiplication of polynomials

TODO: move to SetInterval.thy

lemma *setsum-atMost-Suc-shift*:
fixes *f* :: *nat* $\Rightarrow$ $'a$::*comm-monoid-add*
shows ($\sum$ *i*$\leq$*Suc n. f i*) = *f 0* + ($\sum$ *i*$\leq$*n. f* (*Suc i*))
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **note** *IH* = *this*
have ($\sum$ *i*$\leq$*Suc* (*Suc n*). *f i*) = ($\sum$ *i*$\leq$*Suc n. f i*) + *f* (*Suc* (*Suc n*))
by (*rule setsum-atMost-Suc*)
also have ($\sum$ *i*$\leq$*Suc n. f i*) = *f 0* + ($\sum$ *i*$\leq$*n. f* (*Suc i*))
by (*rule IH*)
also have *f 0* + ($\sum$ *i*$\leq$*n. f* (*Suc i*)) + *f* (*Suc* (*Suc n*)) =
f 0 + (($\sum$ *i*$\leq$*n. f* (*Suc i*)) + *f* (*Suc* (*Suc n*)))
by (*rule add-assoc*)
also have ($\sum$ *i*$\leq$*n. f* (*Suc i*)) + *f* (*Suc* (*Suc n*)) = ($\sum$ *i*$\leq$*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-rec 0* (λ*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*)

41.10 The unit polynomial and exponentiation

instantiation *poly* :: (*comm-semiring-1*) *comm-semiring-1*
begin

definition
one-poly-def:
1 = *pCons 1 0*

instance proof
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*)

instantiation *poly* :: (*comm-semiring-1*) *recpower*
begin

primrec *power-poly* **where**
(*p*::*'a poly*) ^ *0* = *1*
| (*p*::*'a poly*) ^ (*Suc n*) = *p* ∗ *p* ^ *n*

instance
by *default simp-all*

declare *power-poly.simps* [*simp del*]

end

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

41.11 Polynomials form an integral domain

lemma *coeff-mult-degree-sum*:
coeff (*p* ∗ *q*) (*degree p* + *degree q*) =
coeff p (*degree p*) ∗ *coeff q* (*degree q*)
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)
```

41.12 Polynomials form an ordered integral domain

```
definition
  pos-poly :: 'a::ordered-idom poly ⇒ bool
where
  pos-poly p ⟷ 0 < coeff p (degree p)

lemma pos-poly-pCons:
  pos-poly (pCons a p) ⟷ pos-poly p ∨ (p = 0 ∧ 0 < a)
  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* :: (*ordered-idom*) *ordered-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 $|x| = (if\ x < 0\ then\ -x\ else\ x)$
by (*rule abs-poly-def*)
next
fix $x :: 'a\ poly$
show $sgn\ x = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -1)$
by (*rule sgn-poly-def*)
qed

end

TODO: Simplification rules for comparisons

41.13 Long division of polynomials

definition
pdivmod-rel :: '*a*::*field poly* ⇒ '*a poly* ⇒ '*a poly* ⇒ '*a poly* ⇒ *bool*
where
[*code del*]:
pdivmod-rel x y q r ⟷
$x = q * y + r \wedge (if\ y = 0\ then\ q = 0\ else\ r = 0 \vee degree\ r < degree\ y)$

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 ∨ 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* $\longleftrightarrow$ $q = 0 \wedge r = x$
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*. $\exists$ *r*. *pdivmod-rel x y q r*)

definition *mod-poly* **where**
[*code del*]: *x mod y* = (*THE r*. $\exists$ *q*. *pdivmod-rel x y q r*)

lemma *div-poly-eq*:
pdivmod-rel x y q r $\Longrightarrow$ *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 $\Longrightarrow$ *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*)
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 ring-simps degree-mult-eq*)
apply (*cases r'* = *0*)
apply (*simp add*: *pdivmod-rel-def degree-mult-eq*)
apply (*simp add*: *pdivmod-rel-def ring-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

41.14 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*,*recpower*}
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* $(\sum k{\in}A.\ p\ k)$ x = $(\sum k{\in}A.$ *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*,*recpower*} *poly*
shows *poly* $(p\ \hat{}\ n)$ x = *poly* p $x\ \hat{}\ n$
by (*induct n*, *simp*, *simp add*: *power-Suc*)

41.15 Synthetic division

Synthetic division is simply division by the linear polynomial $x - c$.

definition
synthetic-divmod :: *'a*::*comm-semiring-0 poly* ⇒ *'a* ⇒ *'a poly* × *'a*
where [*code del*]:
synthetic-divmod p c =
poly-rec $(0, 0)$ $(\lambda a\ p\ (q, r).\ (pCons\ r\ q, a + c * r))$ 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 =
$(\lambda(q, r).\ (pCons\ r\ q, a + c * r))$ (*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
    with ‹k ≠ 0› have finite {x. poly k x = 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)
```

41.16 Composition of polynomials

definition
pcompose :: *'a::comm-semiring-0 poly* ⇒ *'a poly* ⇒ *'a poly*
where
pcompose p q = *poly-rec 0* (λ*a* - *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

41.17 Order of polynomial roots

definition
order :: *'a::idom* ⇒ *'a poly* ⇒ *nat*
where
[*code del*]:
order a p = (*LEAST n*. ¬ [:−*a*, *1*:] ^ *Suc n dvd p*)

lemma *coeff-linear-power*:
fixes *a* :: *'a::comm-semiring-1*
shows *coeff* ([:*a*, *1*:] ^ *n*) *n* = *1*
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* ≠ *0*
shows *order a p* ≤ *degree p*
proof −
have *order a p* = *degree* ([:−*a*, *1*:] ^ *order a p*)
by (*simp only*: *degree-linear-power*)
also have ... ≤ *degree p*
using *order-1 p* **by** (*rule dvd-imp-degree-le*)
finally show *?thesis* **.**
qed

lemma *order-root*: *poly p a* = *0* ⟷ *p* = *0* ∨ *order a p* ≠ *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

41.18 Configuration of the code generator

code-datatype *0*::*'a*::*zero poly pCons*

declare *pCons-0-0* [*code post*]

instantiation *poly* :: ({*zero*,*eq*}) *eq*

begin

definition [*code del*]:
eq-class.eq (*p*::'*a poly*) *q* ⟷ *p* = *q*

instance
by *default* (*rule eq-poly-def*)

end

lemma *eq-poly-code* [*code*]:
eq-class.eq (*0*::- *poly*) (*0*::- *poly*) ⟷ *True*
eq-class.eq (*0*::- *poly*) (*pCons b q*) ⟷ *eq-class.eq 0 b* ∧ *eq-class.eq 0 q*
eq-class.eq (*pCons a p*) (*0*::- *poly*) ⟷ *eq-class.eq a 0* ∧ *eq-class.eq p 0*
eq-class.eq (*pCons a p*) (*pCons b q*) ⟷ *eq-class.eq a b* ∧ *eq-class.eq p q*
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* ⇒ *'a poly* ⇒ *'a poly* × *'a poly*
where
[*code del*]: *pdivmod x y* = (*x div y*, *x mod y*)

lemma *div-poly-code* [*code*]: *x div y* = *fst* (*pdivmod x y*)
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

end

42 Fundamental-Theorem-Algebra: Fundamental Theorem of Algebra

theory *Fundamental-Theorem-Algebra*
imports *Polynomial Complex*
begin

42.1 Square root of complex numbers

definition *csqrt* :: *complex* ⇒ *complex* **where**

```
csqrt z = (if Im z = 0 then
           if 0 ≤ Re z then Complex (sqrt(Re z)) 0
           else Complex 0 (sqrt(− Re z))
          else Complex (sqrt((cmod z + Re z) /2))
                       ((Im z / abs(Im z)) * sqrt((cmod z − Re z) /2)))

lemma csqrt[algebra]: csqrt z ^ 2 = z
proof−
  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

42.2 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*

42.3 Basic lemmas about complex polynomials

lemma *poly-bound-exists*:
shows ∃ *m*. *m* > *0* ∧ (∀ *z*. *cmod z* <= *r* ⟶ *cmod* (*poly p z*) ≤ *m*)
proof(*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*: $\exists\, x.\ P\ x$ **and** *bz*: $\exists\, z.\ \forall\, x.\ P\ x \longrightarrow x < z$
shows $\exists\,(s::real).\ \forall\, y.\ (\exists\, x.\ P\ x \wedge y < x) \longleftrightarrow y < s$
proof−
from *ex bz* **obtain** *x Y* **where** *x*: *P x* **and** *Y*: $\bigwedge x.\ P\ x \Longrightarrow x < Y$ **by** *blast*
from *ex* **have** *thx*:$\exists\, x.\ x \in Collect\ P$ **by** *blast*
from *bz* **have** *thY*: $\exists\, Y.\ isUb\ UNIV\ (Collect\ P)\ Y$
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* ≥ *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

42.4 Fundamental theorem of algebra

lemma *unimodular-reduce-norm*:
assumes *md*: *cmod z* = *1*
shows *cmod* (*z* + *1*) < *1* ∨ *cmod* (*z* − *1*) < *1* ∨ *cmod* (*z* + *ii*) < *1* ∨ *cmod* (*z* − *ii*) < *1*
proof−
obtain *x y* **where** *z*: *z* = *Complex x y* **by** (*cases z*, *auto*)
from *md z* **have** *xy*: *x^2* + *y^2* = *1* **by** (*simp add*: *cmod-def*)
{assume *C*: *cmod* (*z* + *1*) ≥ *1 cmod* (*z* − *1*) ≥ *1 cmod* (*z* + *ii*) ≥ *1 cmod* (*z* − *ii*) ≥ *1*
from *C z xy* **have** *2*∗*x* ≤ *1 2*∗*x* ≥ −*1 2*∗*y* ≤ *1 2*∗*y* ≥ −*1*
by (*simp-all add*: *cmod-def power2-eq-square algebra-simps*)
hence *abs* (*2*∗*x*) ≤ *1 abs* (*2*∗*y*) ≤ *1* **by** *simp-all*
hence (*abs* (*2* ∗ *x*)) *^2* <= *1^2* (*abs* (*2* ∗ *y*)) *^2* <= *1^2*
by − (*rule power-mono*, *simp*, *simp*)+

```
    hence th0: 4*x^2 ≤ 1 4*y^2 ≤ 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 ≠ 0 and n: n≠0
  shows ∃z. cmod (1 + b * z^n) < 1
using n
proof(induct n rule: nat-less-induct)
  fix n
  assume IH: ∀m<n. m ≠ 0 ⟶ (∃z. cmod (1 + b * z ^ m) < 1) and n: n ≠ 0
  let ?P = λz n. cmod (1 + b * z ^ n) < 1
  {assume e: even n
    hence ∃m. n = 2*m by presburger
    then obtain m where m: n = 2*m by blast
    from n m have m≠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 ∃z. ?P z n ..}
  moreover
  {assume o: odd n
    from b have b': b^2 ≠ 0 unfolding power2-eq-square by simp
    have Im (inverse b) * (Im (inverse b) * |Im b * Im b + Re b * Re b|) +
    Re (inverse b) * (Re (inverse b) * |Im b * Im b + Re b * Re b|) =
    ((Re (inverse b))^2 + (Im (inverse b))^2) * |Im b * Im b + Re b * Re b| by
algebra
    also have ... = cmod (inverse b) ^2 * cmod b ^ 2
      apply (simp add: cmod-def) using realpow-two-le-add-order[of Re b Im b]
      by (simp add: power2-eq-square)
    finally
    have th0: Im (inverse b) * (Im (inverse b) * |Im b * Im b + Re b * Re b|) +
    Re (inverse b) * (Re (inverse b) * |Im b * Im b + Re b * Re b|) =
    1
    apply (simp add: power2-eq-square norm-mult[symmetric] norm-inverse[symmetric])
      using right-inverse[OF b']
    by (simp add: power2-eq-square[symmetric] power-inverse[symmetric] algebra-simps)
    have th0: cmod (complex-of-real (cmod b) / b) = 1
    apply (simp add: complex-Re-mult cmod-def power2-eq-square Re-complex-of-real
Im-complex-of-real divide-inverse algebra-simps )
      by (simp add: real-sqrt-mult[symmetric] th0)
    from o have ∃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 ∃v. cmod (complex-of-real (cmod b) / b + v^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*^*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* ^ *n* = *v*^*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* ^ *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 ∃ *z*. *?P z n* **.. }**
ultimately show ∃ *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*: ∀ *n*. *cmod* (*s n*) ≤ *r*
shows ∃ *f z*. *subseq f* ∧ (∀ *e* >*0*. ∃ *N*. ∀ *n* ≥ *N*. *cmod* (*s* (*f n*) − *z*) < *e*)
proof−
from *seq-monosub*[*of Re o s*]
obtain *f g* **where** *f*: *subseq f monoseq* (λ*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* (λ*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* ≥ *0* **using** *norm-ge-zero*[*of s 0*] **by** *arith*
have *th*:∀ *n*. *r* + *1* ≥ | *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-def real-norm-def* **.**

from *conv2*[*unfolded convergent-def*] **obtain** *y* **where** *LIMSEQ* ($\lambda n.\ Im\ (s\ (f\ (g\ n))))$) *y*
by *blast*
hence *y*: $\forall r{>}0.\ \exists n0.\ \forall n{\geq}n0.\ |\ Im\ (s\ (f\ (g\ n))) - y\ | < r$
unfolding *LIMSEQ-def 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*: $\forall n{\geq}N1.\ |Re\ (s\ (f\ n)) - x| < e\ /\ 2$ **and** *N2*: $\forall n{\geq}N2.\ |Im\ (s\ (f\ (g\ n))) - y| < e\ /\ 2$ **by** *blast*
{fix *n* **assume** *nN12*: $n \geq N1 + N2$
hence *nN1*: $g\ n \geq N1$ **and** *nN2*: $n \geq N2$ **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 $\exists N.\ \forall n{\geq}N.$ *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 $\exists d > 0.\ \forall w.\ 0 < cmod\ (w - z) \wedge cmod\ (w - z) < d \longrightarrow cmod\ (poly\ p\ w - poly\ p\ z) < e$
proof−
obtain *q* **where** *q*: *degree q* = *degree p* $\bigwedge x.$ *poly q x* = *poly p* $(z + x)$
proof
show *degree* (*offset-poly p z*) = *degree p*
by (*rule degree-offset-poly*)
show $\bigwedge x.$ *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*: $m > 0$ $\bigwedge z.$ *cmod z* $\leq 1 \Longrightarrow$ *cmod* (*poly cs z*) $\leq m$ **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 real-mult-order*)
show *?case*
proof(*rule ex-forward*[*OF real-lbound-gt-zero*[*OF one0 em0*]], *clarsimp simp add*: *norm-mult*)
fix *d w*
assume *H*: $d > 0$ $d < 1$ $d < e/m$ $w \neq z$ *cmod* $(w-z) < d$
hence *d1*: *cmod* $(w-z) \leq 1$ $d \geq 0$ **by** *simp-all*
from *H*(*3*) *m*(*1*) **have** *dme*: $d*m < e$ **by** (*simp add*: *field-simps*)
from *H* **have** *th*: *cmod* $(w-z) \leq d$ **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: ⋀z x. cmod z ≤ r ⟹ cmod (poly p z) ≥ ?m
     by auto
    {fix n::nat
     from s1[rule-format, of ?m + 1/real (Suc n)]
     have ∃z. cmod z ≤ r ∧ cmod (poly p z) < − s + 1 / real (Suc n)
       by simp}
    hence th: ∀n. ∃z. cmod z ≤ r ∧ cmod (poly p z) < − s + 1 / real (Suc n) ..
    from choice[OF th] obtain g where
     g: ∀n. cmod (g n) ≤ r ∀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 ∀e>0. ∃N. ∀n≥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* ∀ *w*. *0*<*cmod* (*w* − *z*)∧ *cmod*(*w* − *z*) < *d* ⟶ *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: ∀ *n*≥*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*:
⋀*m x e*. *m* <= *x* ⟹ *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::{recpower, 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::{recpower,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 simp add: power-Suc)}
  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::{recpower,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 algebral

lemma *fundamental-theorem-of-algebra*:
assumes *nc*: ~*constant*(*poly p*)
shows ∃ *z*::*complex*. *poly p z* = *0*
using *nc*
proof(*induct n*≡ *psize p arbitrary*: *p rule*: *nat-less-induct*)
fix *n* **fix** *p* :: *complex poly*
let *?p* = *poly p*
assume *H*: ∀ *m*<*n*. ∀ *p*. ¬ *constant* (*poly p*) ⟶ *m* = *psize p* ⟶ (∃ (*z*::*complex*). *poly p z* = *0*) **and** *nc*: ¬ *constant ?p* **and** *n*: *n* = *psize p*
let *?ths* = ∃ *z*. *?p z* = *0*

from *nonconstant-length*[*OF nc*] **have** *n2*: *n*≥ *2* **by** (*simp add*: *n*)
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 *nc* **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*: $\forall z.$ *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*: $\bigwedge x\ y.$ *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*
$\forall z.$ *poly ?r z* = *poly ?r 0* + *z*^*k*∗ *poly* (*pCons a s*) *z* **by** *blast*
{assume *k* + *1* = *n*
with *kas*(*3*) *lgqr*[*symmetric*] *q*(*1*) *n*[*symmetric*] **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*: *k*+*1* ≠ *n*
from *kn kas*(*3*) *q*(*1*) *n*[*symmetric*] *lgqr* **have** *k1n*: *k* + *1* < *n* **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 *H*[*rule-format*, *OF k1n th01 th02*]
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: $m > 0 \; \forall z.\ cmod\ z \leq cmod\ w \longrightarrow cmod\ (poly\ s\ z) \leq m$ **by** *blast*
have *w0*: $w \neq 0$ **using** *kas(2) w* **by** (*auto simp add*: *power-0-left*)
from *w* **have** $(1 + w\ \hat{}\ k * a) - 1 = 0 - 1$ **by** *simp*
then have *wm1*: $w\hat{}k * a = -1$ **by** *simp*
have *inv0*: $0 < inverse\ (cmod\ w\ \hat{}\ (k + 1) * m)$
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: $t > 0\ t < 1\ t < inverse\ (cmod\ w\ \hat{}\ (k + 1) * m)$ **by** *blast*
let *?ct* = *complex-of-real t*
let *?w* = *?ct* * *w*
have $1 + ?w\hat{}k * (a + ?w * poly\ s\ ?w) = 1 + ?ct\hat{}k * (w\hat{}k * a) + ?w\hat{}k * ?w * poly\ s\ ?w$ **using** *kas*(*1*) **by** (*simp add*: *algebra-simps power-mult-distrib*)
also have ... = *complex-of-real* $(1 - t\hat{}k) + ?w\hat{}k * ?w * poly\ s\ ?w$
unfolding *wm1* **by** (*simp*)
finally have $cmod\ (1 + ?w\hat{}k * (a + ?w * poly\ s\ ?w)) = cmod\ (complex\text{-}of\text{-}real\ (1 - t\hat{}k) + ?w\hat{}k * ?w * poly\ s\ ?w)$
apply −
apply (*rule cong*[*OF refl*[*of cmod*]])
apply *assumption*
done
with *norm-triangle-ineq*[*of complex-of-real* $(1 - t\hat{}k)\ ?w\hat{}k * ?w * poly\ s\ ?w$]
have *th11*: $cmod\ (1 + ?w\hat{}k * (a + ?w * poly\ s\ ?w)) \leq |1 - t\hat{}k| + cmod\ (?w\hat{}k * ?w * poly\ s\ ?w)$ **unfolding** *norm-of-real* **by** *simp*
have *ath*: $\bigwedge x\ (t{::}real).\ 0 \leq x \Longrightarrow x < t \Longrightarrow t \leq 1 \Longrightarrow |1 - t| + x < 1$ **by** *arith*
have $t * cmod\ w \leq 1 * cmod\ w$ **apply** (*rule mult-mono*) **using** *t*(*1*,*2*) **by** *auto*
then have *tw*: $cmod\ ?w \leq cmod\ w$ **using** *t*(*1*) **by** (*simp add*: *norm-mult*)
from *t inv0* **have** $t * (cmod\ w\ \hat{}\ (k + 1) * m) < 1$
by (*simp add*: *inverse-eq-divide field-simps*)
with *zero-less-power*[*OF t*(*1*), *of k*]
have *th30*: $t\hat{}k * (t * (cmod\ w\ \hat{}\ (k + 1) * m)) < t\hat{}k * 1$
apply − **apply** (*rule mult-strict-left-mono*) **by** *simp-all*
have $cmod\ (?w\hat{}k * ?w * poly\ s\ ?w) = t\hat{}k * (t * (cmod\ w\ \hat{}\ (k+1) * cmod\ (poly\ s\ ?w)))$ **using** *w0 t*(*1*)
by (*simp add*: *algebra-simps power-mult-distrib norm-of-real norm-power norm-mult*)
then have $cmod\ (?w\hat{}k * ?w * poly\ s\ ?w) \leq t\hat{}k * (t * (cmod\ w\ \hat{}\ (k + 1) * m))$
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*: $cmod\ (?w\hat{}k * ?w * poly\ s\ ?w) < t\hat{}k$ **by** *simp*
from *power-strict-mono*[*OF t*(*2*), *of k*] *t*(*1*) *kas*(*2*) **have** *th121*: $t\hat{}k \leq 1$
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
```

42.5 Nullstellenstatz, 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: (∃x. poly p x ≠ (0::complex)) ≡ (p ≠ 0)
proof−
  have p = 0 ⟷ poly p = poly 0
    by (simp add: poly-zero)
  also have ... ⟷ (¬ (∃x. poly p x ≠ 0)) by (auto intro: ext)
  finally show (∃x. poly p x ≠ (0::complex)) ≡ p ≠ 0
    by − (atomize (full), blast)
qed

lemma basic-cqe-conv3:
  fixes p q :: complex poly
  assumes l: p ≠ 0
  shows (∃x. poly (pCons a p) x = 0 ∧ poly q x ≠ 0) ≡ ¬ ((pCons a p) dvd (q
^ (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 (∃x. poly (pCons a p) x = 0 ∧ poly q x ≠ 0) ≡ ¬ ((pCons a p) dvd (q ^
(psize p)))
```

unfolding *dp*
by − (*atomize* (*full*), *auto*)
qed

lemma *basic-cqe-conv4*:
fixes *p q* :: *complex poly*
assumes *h*: $\bigwedge x.$ *poly* (*q* ^ *n*) *x* ≡ *poly r x*
shows *p dvd* (*q* ^ *n*) ≡ *p dvd r*
proof−
from *h* **have** *poly* (*q* ^ *n*) = *poly r* **by** (*auto intro*: *ext*)
then have (*q* ^ *n*) = *r* **by** (*simp add*: *poly-eq-iff*)
thus *p dvd* (*q* ^ *n*) ≡ *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* ≡ (−*1*) ∗ (*z*::*complex*) **by** *simp*
lemma *eqT-intr*: *PROP P* ⟹ (*True* ⟹ *PROP P*) *PROP P* ⟹ *True* **by** *blast*+
lemma *negate-negate-rule*: *Trueprop P* ≡ ¬ *P* ≡ *False* **by** (*atomize* (*full*), *auto*)

lemma *complex-entire*: (*z*::*complex*) ≠ *0* ∧ *w* ≠ *0* ≡ *z*∗*w* ≠ *0* **by** *simp*
lemma *resolve-eq-ne*: (*P* ≡ *True*) ≡ (¬*P* ≡ *False*) (*P* ≡ *False*) ≡ (¬*P* ≡ *True*)
by (*atomize* (*full*)) *simp-all*
lemma *cqe-conv1*: *poly 0 x* = *0* ⟷ *True* **by** *simp*
lemma *cqe-conv2*: (*p* ⟹ (*q* ≡ *r*)) ≡ ((*p* ∧ *q*) ≡ (*p* ∧ *r*)) (**is** *?l* ≡ *?r*)
proof
assume *p* ⟹ *q* ≡ *r* **thus** *p* ∧ *q* ≡ *p* ∧ *r* **apply** − **apply** (*atomize* (*full*)) **by** *blast*
next
assume *p* ∧ *q* ≡ *p* ∧ *r p*
thus *q* ≡ *r* **apply** − **apply** (*atomize* (*full*)) **apply** *blast* **done**
qed
lemma *poly-const-conv*: *poly* [:*c*:] (*x*::*complex*) = *y* ⟷ *c* = *y* **by** *simp*

end

43 Lattice-Syntax: Pretty syntax for lattice operations

44 ListVector: Lists as vectors

theory *ListVector*
imports *List Main*
begin

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

abbreviation *scale* :: (*'a*::*times*) ⇒ *'a list* ⇒ *'a list* (**infix** $*_s$ *70*)
where $x *_s xs \equiv map\ (op * x)\ xs$

lemma *scale1*[*simp*]: (*1*::*'a*::*monoid-mult*) $*_s$ *xs* = *xs*
by (*induct xs*) *simp-all*

44.1 + and −

fun *zipwith0* :: (*'a*::*zero* ⇒ *'b*::*zero* ⇒ *'c*) ⇒ *'a list* ⇒ *'b list* ⇒ *'c list*
where
zipwith0 f [] [] = [] |
zipwith0 f (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipwith0 f xs ys* |
zipwith0 f (*x*#*xs*) [] = *f x 0* # *zipwith0 f xs* [] |
zipwith0 f [] (*y*#*ys*) = *f 0 y* # *zipwith0 f* [] *ys*

instantiation *list* :: ({*zero*, *plus*}) *plus*
begin

definition
list-add-def: *op* + = *zipwith0* (*op* +)

instance ..

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

44.2 Inner product

definition *iprod* :: '*a*::*ring list* ⇒ '*a list* ⇒ '*a* (⟨-,-⟩) **where**
⟨*xs*,*ys*⟩ = (∑ (*x*,*y*) ← *zip xs ys*. *x*∗*y*)

lemma *iprod-Nil*[*simp*]: ⟨[],*ys*⟩ = *0*
by(*simp add*:*iprod-def*)

lemma *iprod-Nil2*[*simp*]: ⟨*xs*,[]⟩ = *0*

by(*simp add*:*iprod-def*)

lemma *iprod-Cons*[*simp*]: $\langle x\#xs,y\#ys\rangle = x*y + \langle xs,ys\rangle$
by(*simp add*:*iprod-def*)

lemma *iprod0-if-coeffs0*: $\forall\, c{\in}set\ cs.\ c = 0 \Longrightarrow \langle cs,xs\rangle = 0$
apply(*induct cs arbitrary*:*xs*)
apply *simp*
apply(*case-tac xs*) **apply** *simp*
apply *auto*
done

lemma *iprod-uminus*[*simp*]: $\langle -xs,ys\rangle = -\langle xs,ys\rangle$
by(*simp add*: *iprod-def uminus-listsum-map o-def split-def map-zip-map list-uminus-def*)

lemma *iprod-left-add-distrib*: $\langle xs + ys,zs\rangle = \langle xs,zs\rangle + \langle ys,zs\rangle$
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*: $\langle xs - ys,\ zs\rangle = \langle xs,zs\rangle - \langle ys,zs\rangle$
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*: $\langle x *_s xs,\ ys\rangle = x * \langle xs,ys\rangle$
apply(*induct xs arbitrary*: *ys*)
apply *simp*
apply(*case-tac ys*)
apply (*simp*)
apply (*simp add*:*right-distrib mult-assoc*)
done

end

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

theory *Mapping*
imports *Map Main*
begin

45.1 Type definition and primitive operations

datatype $('a, 'b)$ *map* = *Map* $'a \rightharpoonup 'b$

definition *empty* :: $('a, 'b)$ *map* **where**
empty = *Map* $(\lambda\text{-}.$ *None*$)$

primrec *lookup* :: $('a, 'b)$ *map* $\Rightarrow$ $'a \rightharpoonup 'b$ **where**
lookup (*Map f*) = *f*

primrec *update* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *map* $\Rightarrow ('a, 'b)$ *map* **where**
update k v (*Map f*) = *Map* $(f\ (k \mapsto v))$

primrec *delete* :: $'a \Rightarrow ('a, 'b)$ *map* $\Rightarrow ('a, 'b)$ *map* **where**
delete k (*Map f*) = *Map* $(f\ (k :=$ *None*$))$

primrec *keys* :: $('a, 'b)$ *map* $\Rightarrow$ $'a$ *set* **where**
keys (*Map f*) = *dom f*

45.2 Derived operations

definition *size* :: $('a, 'b)$ *map* $\Rightarrow$ *nat* **where**
size m = (*if finite* (*keys m*) *then card* (*keys m*) *else 0*)

definition *replace* :: $'a \Rightarrow 'b \Rightarrow ('a, 'b)$ *map* $\Rightarrow ('a, 'b)$ *map* **where**
replace k v m = (*if lookup m k* = *None then m else update k v m*)

definition *tabulate* :: $'a$ *list* $\Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a, 'b)$ *map* **where**
tabulate ks f = *Map* (*map-of* (*map* $(\lambda k.\ (k, f\ k))$ *ks*))

definition *bulkload* :: $'a$ *list* $\Rightarrow$ (*nat*, $'a$) *map* **where**
bulkload xs = *Map* $(\lambda k.$ *if k* $<$ *length xs then Some* (*xs* ! *k*) *else None*$)$

45.3 Properties

lemma *lookup-inject*:
lookup m = *lookup n* $\longleftrightarrow$ *m* = *n*
by (*cases m*, *cases n*) *simp*

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*:
lookup (*delete k m*) *k* = *None*
k ≠ *l* ⟹ *lookup* (*delete k m*) *l* = *lookup m l*
by (*cases m*, *simp*)+

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

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

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 (*cases m*, *simp add*: *expand-fun-eq*)+

lemma *replace-update*:
lookup m k = *None* ⟹ *replace k v m* = *m*
lookup m k ≠ *None* ⟹ *replace k v m* = *update k v m*
by (*auto simp add*: *replace-def*)

lemma *delete-empty* [*simp*]:
delete k empty = *empty*
by (*simp add*: *empty-def*)

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 (*cases m*, *simp add*: *expand-fun-eq*)+

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

lemma *keys-empty* [*simp*]:
keys empty = {}
unfolding *empty-def* **by** *simp*

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

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

lemma *keys-tabulate* [*simp*]:
keys (*tabulate ks f*) = *set ks*
by (*auto simp add*: *tabulate-def dest*: *map-of-SomeD intro*!: *weak-map-of-SomeI*)

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

lemma *size-update*:
finite (*keys m*) ⟹ *size* (*update k v m*) =
(*if k* ∈ *keys m then size m else Suc* (*size m*))
by (*simp add*: *size-def keys-update*)
(*auto simp only*: *card-insert card-Suc-Diff1*)

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

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

lemma *bulkload-tabulate*:
bulkload xs = *tabulate* [*0*..<*length xs*] (*nth xs*)
by (*rule sym*)
(*auto simp add*: *bulkload-def tabulate-def expand-fun-eq map-of-eq-None-iff map-compose* [*symmetric*] *comp-def*)

end

46 Multiset: Multisets

theory *Multiset*
imports *List Main*
begin

46.1 The type of multisets

typedef *'a multiset* = {*f*::*'a* => *nat*. *finite* {*x* . *f x* > *0*}}
proof
show (λ*x*. *0*::*nat*) ∈ *?multiset* **by** *simp*
qed

lemmas *multiset-typedef* [*simp*] =
Abs-multiset-inverse Rep-multiset-inverse Rep-multiset

and [*simp*] = *Rep-multiset-inject* [*symmetric*]

definition *Mempty* :: *'a multiset* ({#}) **where**
[*code del*]: {#} = *Abs-multiset* (λ*a*. *0*)

definition *single* :: *'a => 'a multiset* **where**
[*code del*]: *single a* = *Abs-multiset* (λ*b*. *if b* = *a then 1 else 0*)

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

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

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

notation (*xsymbols*)
Melem (**infix** ∈# *50*)

syntax
-MCollect :: *pttrn => 'a multiset => bool => 'a multiset* ((*1*{# - :# -./ -#}))
translations
{#*x* :# *M*. *P*#} == *CONST MCollect M* (λ*x*. *P*)

definition *set-of* :: *'a multiset => 'a set* **where**
set-of M = {*x*. *x* :# *M*}

instantiation *multiset* :: (*type*) {*plus*, *minus*, *zero*, *size*}
begin

definition *union-def* [*code del*]:
M + *N* = *Abs-multiset* (λ*a*. *Rep-multiset M a* + *Rep-multiset N a*)

definition *diff-def* [*code del*]:
M − *N* = *Abs-multiset* (λ*a*. *Rep-multiset M a* − *Rep-multiset N a*)

definition *Zero-multiset-def* [*simp*]:
0 = {#}

definition *size-def*:
size M = *setsum* (*count M*) (*set-of M*)

instance ..

end

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

multiset-inter A B = *A* − (*A* − *B*)

Multiset Enumeration

syntax
-multiset :: *args* => *'a multiset* ({#(-)#})
translations
{#*x*, *xs*#} == {#*x*#} + {#*xs*#}
{#*x*#} == *CONST single x*

Preservation of the representing set *multiset*.

lemma *const0-in-multiset*: (λ*a*. *0*) ∈ *multiset*
by (*simp add*: *multiset-def*)

lemma *only1-in-multiset*: (λ*b*. *if b* = *a then 1 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*:
M ∈ *multiset* ==> (λ*a*. *M a* − *N a*) ∈ *multiset*
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*)
apply *auto*
done

lemma *MCollect-preserves-multiset*:
M ∈ *multiset* ==> (λ*x*. *if P x then M x else 0*) ∈ *multiset*
apply (*simp add*: *multiset-def*)
apply (*rule finite-subset*, *auto*)
done

lemmas *in-multiset* = *const0-in-multiset only1-in-multiset*
union-preserves-multiset diff-preserves-multiset MCollect-preserves-multiset

46.2 Algebraic properties

46.2.1 Union

lemma *union-empty* [*simp*]: *M* + {#} = *M* ∧ {#} + *M* = *M*
by (*simp add*: *union-def Mempty-def in-multiset*)

lemma *union-commute*: *M* + *N* = *N* + (*M*::*'a multiset*)
by (*simp add*: *union-def add-ac in-multiset*)

lemma *union-assoc*: (*M* + *N*) + *K* = *M* + (*N* + (*K*::*'a multiset*))
by (*simp add*: *union-def add-ac in-multiset*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
proof −
  have $M + (N + K) = (N + K) + M$ **by** (*rule union-commute*)
  also have $\ldots = N + (K + M)$ **by** (*rule union-assoc*)
  also have $K + M = M + K$ **by** (*rule union-commute*)
  finally show *?thesis* **.**
qed

lemmas *union-ac = union-assoc union-commute union-lcomm*

instance *multiset* :: (*type*) *comm-monoid-add*
proof
  fix *a b c* :: $'a$ *multiset*
  show $(a + b) + c = a + (b + c)$ **by** (*rule union-assoc*)
  show $a + b = b + a$ **by** (*rule union-commute*)
  show $0 + a = a$ **by** *simp*
qed

46.2.2 Difference

lemma *diff-empty* [*simp*]: $M - \{\#\} = M \wedge \{\#\} - M = \{\#\}$
by (*simp add*: *Mempty-def diff-def in-multiset*)

lemma *diff-union-inverse2* [*simp*]: $M + \{\#a\#\} - \{\#a\#\} = M$
by (*simp add*: *union-def diff-def in-multiset*)

lemma *diff-cancel*: $A - A = \{\#\}$
by (*simp add*: *diff-def Mempty-def*)

46.2.3 Count of elements

lemma *count-empty* [*simp*]: *count* $\{\#\}$ $a = 0$
by (*simp add*: *count-def Mempty-def in-multiset*)

lemma *count-single* [*simp*]: *count* $\{\#b\#\}$ $a = (\textit{if } b = a \textit{ then } 1 \textit{ else } 0)$
by (*simp add*: *count-def single-def in-multiset*)

lemma *count-union* [*simp*]: *count* $(M + N)$ $a =$ *count* M $a +$ *count* N a
by (*simp add*: *count-def union-def in-multiset*)

lemma *count-diff* [*simp*]: *count* $(M - N)$ $a =$ *count* M $a -$ *count* N a
by (*simp add*: *count-def diff-def in-multiset*)

lemma *count-MCollect* [*simp*]:
  count $\{\#\ x{:}\#M.\ P\ x\ \#\}$ $a = (\textit{if } P\ a \textit{ then count } M\ a \textit{ else } 0)$
by (*simp add*: *count-def MCollect-def in-multiset*)

46.2.4 Set of elements

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

lemma *set-of-single* [*simp*]: *set-of* {#*b*#} = {*b*}
by (*simp add*: *set-of-def*)

lemma *set-of-union* [*simp*]: *set-of* (*M* + *N*) = *set-of M* ∪ *set-of N*
by (*auto simp add*: *set-of-def*)

lemma *set-of-eq-empty-iff* [*simp*]: (*set-of M* = {}) = (*M* = {#})
by (*auto simp*: *set-of-def Mempty-def in-multiset count-def expand-fun-eq* [**where** *f*=*Rep-multiset M*])

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*)

46.2.5 Size

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

lemma *size-single* [*simp*]: *size* {#*b*#} = *1*
by (*simp add*: *size-def*)

lemma *finite-set-of* [*iff*]: *finite* (*set-of M*)
using *Rep-multiset* [*of M*] **by** (*simp add*: *multiset-def set-of-def count-def*)

lemma *setsum-count-Int*:
finite A ==> *setsum* (*count N*) (*A* ∩ *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* = {#})

apply (*unfold size-def Mempty-def count-def*, *auto simp*: *in-multiset*)
apply (*simp add*: *set-of-def count-def in-multiset expand-fun-eq*)
done

lemma *nonempty-has-size*: $(S \neq \{\#\}) = (0 < size\ S)$
by (*metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty*)

lemma *size-eq-Suc-imp-elem*: $size\ M = Suc\ n ==> \exists a.\ a :\# M$
apply (*unfold size-def*)
apply (*drule setsum-SucD*)
apply *auto*
done

46.2.6 Equality of multisets

lemma *multiset-eq-conv-count-eq*: $(M = N) = (\forall a.\ count\ M\ a = count\ N\ a)$
by (*simp add*: *count-def expand-fun-eq*)

lemma *single-not-empty* [*simp*]: $\{\#a\#\} \neq \{\#\} \wedge \{\#\} \neq \{\#a\#\}$
by (*simp add*: *single-def Mempty-def in-multiset expand-fun-eq*)

lemma *single-eq-single* [*simp*]: $(\{\#a\#\} = \{\#b\#\}) = (a = b)$
by (*auto simp add*: *single-def in-multiset expand-fun-eq*)

lemma *union-eq-empty* [*iff*]: $(M + N = \{\#\}) = (M = \{\#\} \wedge N = \{\#\})$
by (*auto simp add*: *union-def Mempty-def in-multiset expand-fun-eq*)

lemma *empty-eq-union* [*iff*]: $(\{\#\} = M + N) = (M = \{\#\} \wedge N = \{\#\})$
by (*auto simp add*: *union-def Mempty-def in-multiset expand-fun-eq*)

lemma *union-right-cancel* [*simp*]: $(M + K = N + K) = (M = (N::'a\ multiset))$
by (*simp add*: *union-def in-multiset expand-fun-eq*)

lemma *union-left-cancel* [*simp*]: $(K + M = K + N) = (M = (N::'a\ multiset))$
by (*simp add*: *union-def in-multiset expand-fun-eq*)

lemma *union-is-single*:
$(M + N = \{\#a\#\}) = (M = \{\#a\#\} \wedge N=\{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\})$
apply (*simp add*: *Mempty-def single-def union-def in-multiset add-is-1 expand-fun-eq*)
apply *blast*
done

lemma *single-is-union*:
$(\{\#a\#\} = M + N) \longleftrightarrow (\{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N)$
apply (*unfold Mempty-def single-def union-def*)
apply (*simp add*: *add-is-1 one-is-add in-multiset expand-fun-eq*)
apply (*blast dest*: *sym*)
done

lemma *add-eq-conv-diff*:
(*M* + {#*a*#} = *N* + {#*b*#}) =
(*M* = *N* ∧ *a* = *b* ∨ *M* = *N* − {#*a*#} + {#*b*#} ∧ *N* = *M* − {#*b*#} + {#*a*#})
using [[*simproc del*: *neq*]]
apply (*unfold single-def union-def diff-def*)
apply (*simp* (*no-asm*) *add*: *in-multiset expand-fun-eq*)
apply (*rule conjI*, *force*, *safe*, *simp-all*)
apply (*simp add*: *eq-sym-conv*)
done

declare *Rep-multiset-inject* [*symmetric*, *simp del*]

instance *multiset* :: (*type*) *cancel-ab-semigroup-add*
proof
fix *a b c* :: *'a multiset*
show *a* + *b* = *a* + *c* ⟹ *b* = *c* **by** *simp*
qed

lemma *insert-DiffM*:
x ∈# *M* ⟹ {#*x*#} + (*M* − {#*x*#}) = *M*
by (*clarsimp simp*: *multiset-eq-conv-count-eq*)

lemma *insert-DiffM2*[*simp*]:
x ∈# *M* ⟹ *M* − {#*x*#} + {#*x*#} = *M*
by (*clarsimp simp*: *multiset-eq-conv-count-eq*)

lemma *multi-union-self-other-eq*:
(*A*::*'a multiset*) + *X* = *A* + *Y* ⟹ *X* = *Y*
by (*induct A arbitrary*: *X Y*) *auto*

lemma *multi-self-add-other-not-self*[*simp*]: (*A* = *A* + {#*x*#}) = *False*
by (*metis single-not-empty union-empty union-left-cancel*)

lemma *insert-noteq-member*:
assumes *BC*: *B* + {#*b*#} = *C* + {#*c*#}
and *bnotc*: *b* ≠ *c*
shows *c* ∈# *B*
proof −
have *c* ∈# *C* + {#*c*#} **by** *simp*
have *nc*: ¬ *c* ∈# {#*b*#} **using** *bnotc* **by** *simp*
then have *c* ∈# *B* + {#*b*#} **using** *BC* **by** *simp*
then show *c* ∈# *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*)

lemma *empty-multiset-count*:
$(\forall x.\ count\ A\ x = 0) = (A = \{\#\})$
by (*metis count-empty multiset-eq-conv-count-eq*)

46.2.7 Intersection

lemma *multiset-inter-count*:
$count\ (A \#\cap B)\ x = min\ (count\ A\ x)\ (count\ B\ x)$
by (*simp add*: *multiset-inter-def min-def*)

lemma *multiset-inter-commute*: $A \#\cap B = B \#\cap A$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-max.inf-commute*)

lemma *multiset-inter-assoc*: $A \#\cap (B \#\cap C) = A \#\cap B \#\cap C$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-max.inf-assoc*)

lemma *multiset-inter-left-commute*: $A \#\cap (B \#\cap C) = B \#\cap (A \#\cap C)$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def*)

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *multiset-inter-single*: $a \neq b \Longrightarrow \{\#a\#\} \#\cap \{\#b\#\} = \{\#\}$
by (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count*)

lemma *multiset-union-diff-commute*: $B \#\cap C = \{\#\} \Longrightarrow A + B - C = A - C + B$
apply (*simp add*: *multiset-eq-conv-count-eq multiset-inter-count min-def split*: *split-if-asm*)
apply *clarsimp*
apply (*erule-tac* $x = a$ **in** *allE*)
apply *auto*
done

46.2.8 Comprehension (filter)

lemma *MCollect-empty* [*simp*]: $MCollect\ \{\#\}\ P = \{\#\}$
by (*simp add*: *MCollect-def Mempty-def Abs-multiset-inject in-multiset expand-fun-eq*)

lemma *MCollect-single* [*simp*]:
$MCollect\ \{\#x\#\}\ P = (if\ P\ x\ then\ \{\#x\#\}\ else\ \{\#\})$

by (*simp add*: *MCollect-def Mempty-def single-def Abs-multiset-inject in-multiset expand-fun-eq*)

lemma *MCollect-union* [*simp*]:
MCollect (*M*+*N*) *f* = *MCollect M f* + *MCollect N f*
by (*simp add*: *MCollect-def union-def Abs-multiset-inject in-multiset expand-fun-eq*)

46.3 Induction and case splits

lemma *setsum-decr*:
finite F ==> (*0*::*nat*) < *f a* ==>
setsum (*f* (*a* := *f a* − *1*)) *F* = (*if a*∈*F then setsum f F* − *1 else setsum f F*)
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* (λ*a*. (*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* = (λ*a*.*0*))
apply *simp*
apply (*rule 1*)
apply (*rule ext*, *force*, *clarify*)
apply (*frule setsum-SucD*, *clarify*)
apply (*rename-tac a*)
apply (*subgoal-tac finite* {*x*. (*f* (*a* := *f a* − *1*)) *x* > *0*})
prefer *2*
apply (*rule finite-subset*)
prefer *2*
apply *assumption*
apply *simp*
apply *blast*
apply (*subgoal-tac f* = (*f* (*a* := *f a* − *1*))(*a* := (*f* (*a* := *f a* − *1*)) *a* + *1*))
prefer *2*
apply (*rule ext*)
apply (*simp* (*no-asm-simp*))
apply (*erule ssubst*, *rule 2* [*unfolded multiset-def*], *blast*)
apply (*erule allE*, *erule impE*, *erule-tac* [*2*] *mp*, *blast*)
apply (*simp* (*no-asm-simp*) *add*: *setsum-decr del*: *fun-upd-apply One-nat-def*)
apply (*subgoal-tac* {*x*. *x* ≠ *a* −−> *f x* ≠ *0*} = {*x*. *f x* ≠ *0*})
prefer *2*
apply *blast*
apply (*subgoal-tac* {*x*. *x* ≠ *a* ∧ *f x* ≠ *0*} = {*x*. *f x* ≠ *0*} − {*a*})
prefer *2*

```
 apply blast
apply (simp add: le-imp-diff-is-add setsum-diff1-nat cong: conj-cong)
done

theorem rep-multiset-induct:
  f ∈ multiset ==> P (λa. 0) ==>
    (!!f b. f ∈ multiset ==> P f ==> P (f (b := f b + 1))) ==> P f
using rep-multiset-induct-aux by blast

theorem multiset-induct [case-names empty add, induct type: multiset]:
assumes empty: P {#}
  and add: !!M x. P M ==> P (M + {#x#})
shows P M
proof −
  note defns = union-def single-def Mempty-def
  show ?thesis
    apply (rule Rep-multiset-inverse [THEN subst])
    apply (rule Rep-multiset [THEN rep-multiset-induct])
     apply (rule empty [unfolded defns])
    apply (subgoal-tac f(b := f b + 1) = (λa. f a + (if a=b then 1 else 0)))
     prefer 2
     apply (simp add: expand-fun-eq)
    apply (erule ssubst)
    apply (erule Abs-multiset-inverse [THEN subst])
    apply (drule add [unfolded defns, simplified])
    apply(simp add:in-multiset)
    done
qed

lemma multi-nonempty-split: M ≠ {#} ⟹ ∃A a. M = A + {#a#}
by (induct M) auto

lemma multiset-cases [cases type, case-names empty add]:
assumes em:  M = {#} ⟹ P
assumes add: ⋀N x. M = N + {#x#} ⟹ P
shows P
proof (cases M = {#})
  assume M = {#} then show ?thesis using em by simp
next
  assume M ≠ {#}
  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 ∈# M ⟹ ∃A. M = A + {#x#}
apply (cases M)
 apply simp
apply (rule-tac x=M − {#x#} in exI, simp)
```

done

lemma *multiset-partition*: $M = \{\#\ x{:}\#M.\ P\ x\ \#\} + \{\#\ x{:}\#M.\ \neg\ P\ x\ \#\}$
apply (*subst multiset-eq-conv-count-eq*)
apply *auto*
done

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

lemma *multi-drop-mem-not-eq*: $c \in\# B \Longrightarrow B - \{\#c\#\} \neq B$
by (*cases* $B = \{\#\}$) (*auto dest*: *multi-member-split*)

46.4 Orderings

46.4.1 Well-foundedness

definition *mult1* :: $('a \times 'a)$ *set* => $('a$ *multiset* $\times$ $'a$ *multiset*) *set* **where**
[*code del*]: *mult1* $r = \{(N, M).\ \exists\, a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$
$(\forall\, b.\ b :\# K \longrightarrow (b, a) \in r)\}$

definition *mult* :: $('a \times 'a)$ *set* => $('a$ *multiset* $\times$ $'a$ *multiset*) *set* **where**
mult $r = (mult1\ r)^+$

lemma *not-less-empty* [*iff*]: $(M, \{\#\}) \notin mult1\ r$
by (*simp add*: *mult1-def*)

lemma *less-add*: $(N, M0 + \{\#a\#\}) \in mult1\ r \Longrightarrow$
$(\exists\, M.\ (M, M0) \in mult1\ r \wedge N = M + \{\#a\#\}) \vee$
$(\exists\, K.\ (\forall\, b.\ b :\# K \longrightarrow (b, a) \in r) \wedge N = M0 + K)$
(**is** - $\Longrightarrow$ *?case1* $(mult1\ r) \vee$ *?case2*)
proof (*unfold mult1-def*)
let *?r* $= \lambda K\ a.\ \forall\, b.\ b :\# K \longrightarrow (b, a) \in r$
let *?R* $= \lambda N\ M.\ \exists\, a\ M0\ K.\ M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge$ *?r K a*
let *?case1* = *?case1* $\{(N, M).$ *?R N M*$\}$

assume $(N, M0 + \{\#a\#\}) \in \{(N, M).$ *?R N M*$\}$
then have $\exists\, a'\ M0'\ K.$
$M0 + \{\#a\#\} = M0' + \{\#a'\#\} \wedge N = M0' + K \wedge$ *?r K* a' **by** *simp*
then show *?case1* $\vee$ *?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 $M0 = M0' \wedge a = a' \vee$
$(\exists\, K'.\ M0 = K' + \{\#a'\#\} \wedge M0' = K' + \{\#a\#\})$
by (*simp only*: *add-eq-conv-ex*)
then show *?thesis*
proof (*elim disjE conjE exE*)
assume $M0 = M0'\ a = a'$

```
      with N r have ?r K a ∧ N = M0 + K by simp
      then have ?case2 .. then show ?thesis ..
    next
      fix K'
      assume M0' = K' + {#a#}
      with N have n: N = K' + K + {#a#} by (simp add: union-ac)

      assume M0 = K' + {#a'#}
      with r have ?R (K' + K) M0 by blast
      with n have ?case1 by simp then show ?thesis ..
    qed
  qed
qed

lemma all-accessible: wf r ==> ∀ M. M ∈ acc (mult1 r)
proof
  let ?R = mult1 r
  let ?W = acc ?R
  {
    fix M M0 a
    assume M0: M0 ∈ ?W
      and wf-hyp: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
      and acc-hyp: ∀ M. (M, M0) ∈ ?R --> M + {#a#} ∈ ?W
    have M0 + {#a#} ∈ ?W
    proof (rule accI [of M0 + {#a#}])
      fix N
      assume (N, M0 + {#a#}) ∈ ?R
      then have ((∃ M. (M, M0) ∈ ?R ∧ N = M + {#a#}) ∨
          (∃ K. (∀ b. b :# K --> (b, a) ∈ r) ∧ N = M0 + K))
        by (rule less-add)
      then show N ∈ ?W
      proof (elim exE disjE conjE)
        fix M assume (M, M0) ∈ ?R and N: N = M + {#a#}
        from acc-hyp have (M, M0) ∈ ?R --> M + {#a#} ∈ ?W ..
        from this and ‹(M, M0) ∈ ?R› have M + {#a#} ∈ ?W ..
        then show N ∈ ?W by (simp only: N)
      next
        fix K
        assume N: N = M0 + K
        assume ∀ b. b :# K --> (b, a) ∈ r
        then have M0 + K ∈ ?W
        proof (induct K)
          case empty
          from M0 show M0 + {#} ∈ ?W by simp
        next
          case (add K x)
          from add.prems have (x, a) ∈ r by simp
          with wf-hyp have ∀ M ∈ ?W. M + {#x#} ∈ ?W by blast
          moreover from add have M0 + K ∈ ?W by simp
```

```
        ultimately have (M0 + K) + {#x#} ∈ ?W ..
        then show M0 + (K + {#x#}) ∈ ?W by (simp only: union-assoc)
      qed
      then show N ∈ ?W by (simp only: N)
    qed
  qed
 } note tedious-reasoning = this

 assume wf: wf r
 fix M
 show M ∈ ?W
 proof (induct M)
   show {#} ∈ ?W
   proof (rule accI)
     fix b assume (b, {#}) ∈ ?R
     with not-less-empty show b ∈ ?W by contradiction
   qed

   fix M a assume M ∈ ?W
   from wf have ∀ M ∈ ?W. M + {#a#} ∈ ?W
   proof induct
     fix a
     assume r: !!b. (b, a) ∈ r ==> (∀ M ∈ ?W. M + {#b#} ∈ ?W)
     show ∀ M ∈ ?W. M + {#a#} ∈ ?W
     proof
       fix M assume M ∈ ?W
       then show M + {#a#} ∈ ?W
         by (rule acc-induct) (rule tedious-reasoning [OF - r])
     qed
   qed
   from this and ‹M ∈ ?W› show M + {#a#} ∈ ?W ..
 qed
qed

theorem wf-mult1: wf r ==> wf (mult1 r)
by (rule acc-wfI) (rule all-accessible)

theorem wf-mult: wf r ==> wf (mult r)
unfolding mult-def by (rule wf-trancl) (rule wf-mult1)
```

46.4.2 Closure-free presentation

```
lemma diff-union-single-conv: a :# J ==> I + J - {#a#} = I + (J - {#a#})
by (simp add: multiset-eq-conv-count-eq)
```

One direction.

```
lemma mult-implies-one-step:
  trans r ==> (M, N) ∈ mult r ==>
    ∃ I J K. N = I + J ∧ M = I + K ∧ J ≠ {#} ∧
    (∀ k ∈ set-of K. ∃ j ∈ set-of J. (k, j) ∈ r)
```

apply (*unfold mult-def mult1-def set-of-def*)
apply (*erule converse-trancl-induct*, *clarify*)
apply (*rule-tac x = M0* **in** *exI*, *simp*, *clarify*)
apply (*case-tac a :# K*)
apply (*rule-tac x = I* **in** *exI*)
apply (*simp* (*no-asm*))
apply (*rule-tac x = (K − {#a#}) + Ka* **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *union-assoc* [*symmetric*])
apply (*drule-tac f = λM. M − {#a#}* **in** *arg-cong*)
apply (*simp add*: *diff-union-single-conv*)
apply (*simp* (*no-asm-use*) *add*: *trans-def*)
apply *blast*
apply (*subgoal-tac a :# I*)
apply (*rule-tac x = I − {#a#}* **in** *exI*)
apply (*rule-tac x = J + {#a#}* **in** *exI*)
apply (*rule-tac x = K + Ka* **in** *exI*)
apply (*rule conjI*)
apply (*simp add*: *multiset-eq-conv-count-eq split*: *nat-diff-split*)
apply (*rule conjI*)
apply (*drule-tac f = λM. M − {#a#}* **in** *arg-cong*, *simp*)
apply (*simp add*: *multiset-eq-conv-count-eq split*: *nat-diff-split*)
apply (*simp* (*no-asm-use*) *add*: *trans-def*)
apply *blast*
apply (*subgoal-tac a :# (M0 + {#a#})*)
apply *simp*
apply (*simp* (*no-asm*))
done

lemma *elem-imp-eq-diff-union*: *a :# M ==> M = M − {#a#} + {#a#}*
by (*simp add*: *multiset-eq-conv-count-eq*)

lemma *size-eq-Suc-imp-eq-union*: *size M = Suc n ==> ∃ a N. M = N + {#a#}*
apply (*erule size-eq-Suc-imp-elem* [*THEN exE*])
apply (*drule elem-imp-eq-diff-union*, *auto*)
done

lemma *one-step-implies-mult-aux*:
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*: *union-assoc* [*symmetric*] *mult-def*)
apply (*erule trancl-trans*)
apply (*rule r-into-trancl*)
apply (*simp add*: *mult1-def set-of-def*)
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = I + J'$ **in** *exI*)
apply (*simp add*: *union-ac*)
done

lemma *one-step-implies-mult*:
 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*

46.4.3 Partial-order properties

instantiation *multiset* :: (*order*) *order*
begin

definition *less-multiset-def* [*code del*]:
 $M' < M \longleftrightarrow (M',\ M) \in$ *mult* $\{(x',\ x).\ x' < x\}$

definition *le-multiset-def* [*code del*]:
 $M' <= M \longleftrightarrow M' = M \vee M' <$ (M::'*a multiset*)

lemma *trans-base-order*: *trans* $\{(x',\ x).\ x' <$ (x::'*a*::*order*)$\}$
unfolding *trans-def* **by** (*blast intro*: *order-less-trans*)

Irreflexivity.

lemma *mult-irrefl-aux*:
 finite A ==> $(\forall x \in A.\ \exists y \in A.\ x <$ (y::'*a*::*order*)$) \Longrightarrow A = \{\}$
by (*induct rule*: *finite-induct*) (*auto intro*: *order-less-trans*)

lemma *mult-less-not-refl*: $\neg\ M <$ (M::'*a*::*order multiset*)
apply (*unfold less-multiset-def*, *auto*)
apply (*drule trans-base-order* [*THEN mult-implies-one-step*], *auto*)
apply (*drule finite-set-of* [*THEN mult-irrefl-aux* [*rule-format* (*no-asm*)]])
apply (*simp add*: *set-of-eq-empty-iff*)
done

lemma *mult-less-irrefl* [*elim*!]: $M < (M::'a::order\ multiset) ==> R$
using *insert mult-less-not-refl* **by** *fast*

Transitivity.

theorem *mult-less-trans*: $K < M ==> M < N ==> K < (N::'a::order\ multiset)$
unfolding *less-multiset-def mult-def* **by** (*blast intro*: *trancl-trans*)

Asymmetry.

theorem *mult-less-not-sym*: $M < N ==> \neg\ N < (M::'a::order\ multiset)$
apply *auto*
apply (*rule mult-less-not-refl* [*THEN notE*])
apply (*erule mult-less-trans*, *assumption*)
done

theorem *mult-less-asym*:
$M < N ==> (\neg\ P ==> N < (M::'a::order\ multiset)) ==> P$
using *mult-less-not-sym* **by** *blast*

theorem *mult-le-refl* [*iff*]: $M <= (M::'a::order\ multiset)$
unfolding *le-multiset-def* **by** *auto*

Anti-symmetry.

theorem *mult-le-antisym*:
$M <= N ==> N <= M ==> M = (N::'a::order\ multiset)$
unfolding *le-multiset-def* **by** (*blast dest*: *mult-less-not-sym*)

Transitivity.

theorem *mult-le-trans*:
$K <= M ==> M <= N ==> K <= (N::'a::order\ multiset)$
unfolding *le-multiset-def* **by** (*blast intro*: *mult-less-trans*)

theorem *mult-less-le*: $(M < N) = (M <= N \wedge M \neq (N::'a::order\ multiset))$
unfolding *le-multiset-def* **by** *auto*

instance proof
qed (*auto simp add*: *mult-less-le dest*: *mult-le-antisym elim*: *mult-le-trans*)

end

46.4.4 Monotonicity of multiset union

lemma *mult1-union*:
$(B, D) \in mult1\ r ==> trans\ r ==> (C + B, C + D) \in mult1\ r$
apply (*unfold mult1-def*)
apply *auto*
apply (*rule-tac* $x = a$ **in** *exI*)
apply (*rule-tac* $x = C + M0$ **in** *exI*)
apply (*simp add*: *union-assoc*)
done

lemma *union-less-mono2*: $B < D \Longrightarrow C + B < C + (D::'a::order\ multiset)$
apply (*unfold less-multiset-def mult-def*)
apply (*erule trancl-induct*)
apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl*)
apply (*blast intro*: *mult1-union transI order-less-trans r-into-trancl trancl-trans*)
done

lemma *union-less-mono1*: $B < D \Longrightarrow B + C < D + (C::'a::order\ multiset)$
apply (*subst union-commute* [*of B C*])
apply (*subst union-commute* [*of D C*])
apply (*erule union-less-mono2*)
done

lemma *union-less-mono*:
$A < C \Longrightarrow B < D \Longrightarrow A + B < C + (D::'a::order\ multiset)$
by (*blast intro*!: *union-less-mono1 union-less-mono2 mult-less-trans*)

lemma *union-le-mono*:
$A <= C \Longrightarrow B <= D \Longrightarrow A + B <= C + (D::'a::order\ multiset)$
unfolding *le-multiset-def*
by (*blast intro*: *union-less-mono union-less-mono1 union-less-mono2*)

lemma *empty-leI* [*iff*]: $\{\#\} <= (M::'a::order\ multiset)$
apply (*unfold le-multiset-def less-multiset-def*)
apply (*case-tac* $M = \{\#\}$)
prefer *2*
apply (*subgoal-tac* $(\{\#\} + \{\#\}, \{\#\} + M) \in mult\ (Collect\ (split\ op <)))$
prefer *2*
apply (*rule one-step-implies-mult*)
apply (*simp only*: *trans-def*)
apply *auto*
done

lemma *union-upper1*: $A <= A + (B::'a::order\ multiset)$
proof −
have $A + \{\#\} <= A + B$ **by** (*blast intro*: *union-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *union-upper2*: $B <= A + (B::'a::order\ multiset)$
by (*subst union-commute*) (*rule union-upper1*)

instance *multiset* :: (*order*) *pordered-ab-semigroup-add*
apply *intro-classes*
apply (*erule union-le-mono*[*OF mult-le-refl*])
done

46.5 Link with lists

primrec *multiset-of* :: *'a list* ⇒ *'a multiset* **where**
multiset-of [] = {#} |
multiset-of (*a* # *x*) = *multiset-of x* + {# *a* #}

lemma *multiset-of-zero-iff*[*simp*]: (*multiset-of x* = {#}) = (*x* = [])
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*: *union-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-eq-conv-count-eq 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-eq-conv-count-eq distinct-count-atmost-1*)

lemma *set-eq-iff-multiset-of-remdups-eq*:
  (*set x* = *set y*) = (*multiset-of* (*remdups x*) = *multiset-of* (*remdups y*))
apply (*rule iffI*)
apply (*simp add*: *set-eq-iff-multiset-of-eq-distinct*[*THEN iffD1*])
apply (*drule distinct-remdups* [*THEN distinct-remdups*
    [*THEN set-eq-iff-multiset-of-eq-distinct* [*THEN iffD2*]]])
apply *simp*
done

lemma *multiset-of-compl-union* [*simp*]:
  multiset-of [*x*←*xs*. *P x*] + *multiset-of* [*x*←*xs*. ¬*P x*] = *multiset-of xs*
by (*induct xs*) (*auto simp*: *union-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*: *multiset-of* (*remove1 a xs*) = *multiset-of xs* − {#*a*#}
by (*induct xs*) (*auto simp add*: *multiset-eq-conv-count-eq*)

lemma *multiset-of-eq-length*:
assumes *multiset-of xs* = *multiset-of ys*
shows *length xs* = *length ys*
using *assms*
proof (*induct arbitrary*: *ys rule*: *length-induct*)
  case (*1 xs ys*)
  show *?case*
  proof (*cases xs*)
    case *Nil* **with** *1.prems* **show** *?thesis* **by** *simp*
  next
    case (*Cons x xs'*)
    note *xCons* = *Cons*
    show *?thesis*
    proof (*cases ys*)
      case *Nil*
      with *1.prems Cons* **show** *?thesis* **by** *simp*
    next
      case (*Cons y ys'*)
      have *x-in-ys*: *x* = *y* ∨ *x* ∈ *set ys'*
      proof (*cases x* = *y*)

```
      case True then show ?thesis ..
    next
      case False
     from 1.prems [symmetric] xCons Cons have x :# multiset-of ys' + {#y#}
by simp
      with False show ?thesis by (simp add: mem-set-multiset-eq)
    qed
    from 1.hyps have IH: length xs' < length xs ⟶
      (∀ x. multiset-of xs' = multiset-of x ⟶ length xs' = length x) by blast
   from 1.prems x-in-ys Cons xCons have multiset-of xs' = multiset-of (remove1
x (y#ys'))
      apply −
      apply (simp add: multiset-of-remove1, simp only: add-eq-conv-diff)
      apply fastsimp
      done
   with IH xCons have IH': length xs' = length (remove1 x (y#ys')) by fastsimp
    from x-in-ys have x ≠ y ⟹ length ys' > 0 by auto
   with Cons xCons x-in-ys IH' show ?thesis by (auto simp add: length-remove1)
   qed
  qed
qed
```

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

```
lemma properties-for-sort:
  multiset-of ys = multiset-of xs ⟹ sorted ys ⟹ sort xs = ys
proof (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
```

46.6 Pointwise ordering induced by count

```
definition mset-le :: 'a multiset ⇒ 'a multiset ⇒ bool  (infix ≤# 50) where
  [code del]: A ≤# B ⟷ (∀ a. count A a ≤ count B a)

definition mset-less :: 'a multiset ⇒ 'a multiset ⇒ bool  (infix <# 50) where
  [code del]: A <# B ⟷ A ≤# B ∧ A ≠ B

notation mset-le  (infix ⊆# 50)
notation mset-less  (infix ⊂# 50)

lemma mset-le-refl[simp]: A ≤# A
unfolding mset-le-def by auto
```

lemma *mset-le-trans*: $A \leq\# B \Longrightarrow B \leq\# C \Longrightarrow A \leq\# C$
unfolding *mset-le-def* **by** (*fast intro*: *order-trans*)

lemma *mset-le-antisym*: $A \leq\# B \Longrightarrow B \leq\# A \Longrightarrow A = B$
apply (*unfold mset-le-def*)
apply (*rule multiset-eq-conv-count-eq* [*THEN iffD2*])
apply (*blast intro*: *order-antisym*)
done

lemma *mset-le-exists-conv*: $(A \leq\# B) = (\exists C.\ B = A + C)$
apply (*unfold mset-le-def*, *rule iffI*, *rule-tac* $x = B - A$ **in** *exI*)
apply (*auto intro*: *multiset-eq-conv-count-eq* [*THEN iffD2*])
done

lemma *mset-le-mono-add-right-cancel*[*simp*]: $(A + C \leq\# B + C) = (A \leq\# B)$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-mono-add-left-cancel*[*simp*]: $(C + A \leq\# C + B) = (A \leq\# B)$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-mono-add*: $[\![A \leq\# B;\ C \leq\# D]\!] \Longrightarrow A + C \leq\# B + D$
apply (*unfold mset-le-def*)
apply *auto*
apply (*erule-tac* $x = a$ **in** *allE*)+
apply *auto*
done

lemma *mset-le-add-left*[*simp*]: $A \leq\# A + B$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-add-right*[*simp*]: $B \leq\# A + B$
unfolding *mset-le-def* **by** *auto*

lemma *mset-le-single*: $a :\# B \Longrightarrow \{\#a\#\} \leq\# B$
by (*simp add*: *mset-le-def*)

lemma *multiset-diff-union-assoc*: $C \leq\# B \Longrightarrow A + B - C = A + (B - C)$
by (*simp add*: *multiset-eq-conv-count-eq mset-le-def*)

lemma *mset-le-multiset-union-diff-commute*:
assumes $B \leq\# A$
shows $A - B + C = A + C - B$
proof −
  from *mset-le-exists-conv* [*of B A*] *assms* **have** $\exists D.\ A = B + D$ **..**
  from *this* **obtain** D **where** $A = B + D$ **..**
  then show *?thesis*
    apply *simp*
    apply (*subst union-commute*)

```
    apply (subst multiset-diff-union-assoc)
    apply simp
    apply (simp add: diff-cancel)
    apply (subst union-assoc)
    apply (subst union-commute[of B -])
    apply (subst multiset-diff-union-assoc)
    apply simp
    apply (simp add: diff-cancel)
    done
qed

lemma multiset-of-remdups-le: multiset-of (remdups xs) ≤# multiset-of xs
apply (induct xs)
 apply auto
apply (rule mset-le-trans)
 apply auto
done

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 union-assoc)
      apply (subst union-commute [where M = {#v#} and N = {#x#}])
      apply (subst union-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
apply (case-tac i = j)
 apply simp
apply (simp add: multiset-of-update)
apply (subst elem-imp-eq-diff-union[symmetric])
 apply (simp add: nth-mem-multiset-of)
```

apply *simp*
done

interpretation *mset-order*: *order op ≤# op <#*
proof qed (*auto intro*: *order.intro mset-le-refl mset-le-antisym*
mset-le-trans simp: *mset-less-def*)

interpretation *mset-order-cancel-semigroup*:
pordered-cancel-ab-semigroup-add op + op ≤# op <#
proof qed (*erule mset-le-mono-add* [*OF mset-le-refl*])

interpretation *mset-order-semigroup-cancel*:
pordered-ab-semigroup-add-imp-le op + op ≤# op <#
proof qed *simp*

lemma *mset-lessD*: $A \subset\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x=x* **in** *allE*)
apply *auto*
done

lemma *mset-leD*: $A \subseteq\# B \Longrightarrow x \in\# A \Longrightarrow x \in\# B$
apply (*clarsimp simp*: *mset-le-def mset-less-def*)
apply (*erule-tac x = x* **in** *allE*)
apply *auto*
done

lemma *mset-less-insertD*: $(A + \{\#x\#\} \subset\# B) \Longrightarrow (x \in\# B \wedge A \subset\# B)$
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*: $(A + \{\#x\#\} \subseteq\# B) \Longrightarrow (x \in\# B \wedge A \subseteq\# B)$
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*]: $A \subset\# \{\#\} = False$
by (*induct A*) (*auto simp*: *mset-le-def mset-less-def*)

lemma *multi-psub-of-add-self*[*simp*]: $A \subset\# A + \{\#x\#\}$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *multi-psub-self*[*simp*]: $A \subset\# A = False$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-add-bothsides*:
$T + \{\#x\#\} \subset\# S + \{\#x\#\} \Longrightarrow T \subset\# S$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-empty-nonempty*: $(\{\#\} \subset\# S) = (S \neq \{\#\})$
by (*auto simp*: *mset-le-def mset-less-def*)

lemma *mset-less-size*: $A \subset\# B \Longrightarrow size\ A < size\ B$
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*: $\bigwedge B.\ S \subset\# B \Longrightarrow size\ S < size\ B$ **by** *fact*
have *SxsubT*: $S + \{\#x\#\} \subset\# T$ **by** *fact*
then have $x \in\# T$ **and** $S \subset\# T$ **by** (*auto dest*: *mset-less-insertD*)
then obtain T' **where** T: $T = T' + \{\#x\#\}$
by (*blast dest*: *multi-member-split*)
then have $S \subset\# T'$ **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

lemmas *mset-less-trans* = *mset-order.less-trans*

lemma *mset-less-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
by (*auto simp*: *mset-le-def mset-less-def multi-drop-mem-not-eq*)

46.7 Strong induction and subset induction for multisets

Well-foundedness of proper subset operator:

proper multiset subset

definition
mset-less-rel :: ('*a multiset* * '*a multiset*) *set* **where**
mset-less-rel = {(*A*,*B*). $A \subset\# B$}

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$ **and** $b \neq c$
shows $B - \{\#c\#\} + \{\#b\#\} = B + \{\#b\#\} - \{\#c\#\}$
proof −
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: union-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: ⋀B. ∀A. A ⊂# B ⟶ P A ⟹ 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 ⊆# A
  and empty: P {#}
  and insert: ⋀a F. a ∈# A ⟹ P F ⟹ P (F + {#a#})
shows P F
proof -
  from ‹F ⊆# A›
  show ?thesis
  proof (induct F)
    show P {#} by fact
  next
    fix x F
    assume P: F ⊆# A ⟹ P F and i: F + {#x#} ⊆# A
    show P (F + {#x#})
    proof (rule insert)
      from i show x ∈# A by (auto dest: mset-le-insertD)
      from i have F ⊆# A by (auto dest: mset-le-insertD)
      with P show P F .
    qed
  qed
qed
```

A consequence: Extensionality.

```
lemma multi-count-eq: (∀x. count A x = count B x) = (A = B)
apply (rule iffI)
 prefer 2
 apply clarsimp
apply (induct A arbitrary: B rule: full-multiset-induct)
```

apply (*rename-tac C*)
apply (*case-tac B rule*: *multiset-cases*)
apply (*simp add*: *empty-multiset-count*)
apply *simp*
apply (*case-tac x* ∈# *C*)
apply (*force dest*: *multi-member-split*)
apply (*erule-tac x* = *x* **in** *allE*)
apply *simp*
done

lemmas *multi-count-ext* = *multi-count-eq* [*THEN iffD1*, *rule-format*]

46.8 The fold combinator

The intended behaviour is *fold-mset f z* {#x_1, ..., x_n#} = *f* x_1 (... (*f* x_n *z*)...) 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* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ '*a multiset* ⇒ '*b* ⇒ *bool*
for *f* :: '*a* ⇒ '*b* ⇒ '*b*
and *z* :: '*b*
where
emptyI [*intro*]: *fold-msetG f z* {#} *z*
| *insertI* [*intro*]: *fold-msetG f z A y* ⟹ *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* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ '*a multiset* ⇒ '*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* ⟹ *x* ∈# *A* ⟹ *fold-msetG f z A* (*f x y*)
apply (*frule-tac x* = *x* **in** *fold-msetG.insertI*)
apply *auto*
done

lemma *fold-msetG-nonempty*: ∃ *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*

locale *left-commutative* =
fixes f :: $'a$ => $'b$ => $'b$
assumes *left-commute*: $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$
begin

lemma *fold-msetG-determ*:
$\mathit{fold\text{-}msetG}\ f\ z\ A\ x \Longrightarrow \mathit{fold\text{-}msetG}\ f\ z\ A\ y \Longrightarrow y = x$
proof (*induct arbitrary*: $x\ y\ z$ *rule*: *full-multiset-induct*)
case (*less* $M\ x_1\ x_2\ Z$)
have *IH*: $\forall A.\ A \subset\# M \longrightarrow$
$(\forall x\ x'\ x''.\ \mathit{fold\text{-}msetG}\ f\ x''\ A\ x \longrightarrow \mathit{fold\text{-}msetG}\ f\ x''\ A\ x'$
$\longrightarrow x' = x)$ **by** *fact*
have $\mathit{Mfold}x_1$: $\mathit{fold\text{-}msetG}\ f\ Z\ M\ x_1$ **and** $\mathit{Mfold}x_2$: $\mathit{fold\text{-}msetG}\ f\ Z\ M\ x_2$ **by** *fact+*
show *?case*
proof (*rule fold-msetG.cases* [*OF* $\mathit{Mfold}x_1$])
assume $M = \{\#\}$ **and** $x_1 = Z$
then show *?case* **using** $\mathit{Mfold}x_2$ **by** *auto*
next
fix $B\ b\ u$
assume $M = B + \{\#b\#\}$ **and** $x_1 = f\ b\ u$ **and** Bu: $\mathit{fold\text{-}msetG}\ f\ Z\ B\ u$
then have *MBb*: $M = B + \{\#b\#\}$ **and** x_1: $x_1 = f\ b\ u$ **by** *auto*
show *?case*
proof (*rule fold-msetG.cases* [*OF* $\mathit{Mfold}x_2$])
assume $M = \{\#\}\ x_2 = Z$
then show *?case* **using** $\mathit{Mfold}x_1$ **by** *auto*
next
fix $C\ c\ v$
assume $M = C + \{\#c\#\}$ **and** $x_2 = f\ c\ v$ **and** Cv: $\mathit{fold\text{-}msetG}\ f\ Z\ C\ v$
then have *MCc*: $M = C + \{\#c\#\}$ **and** x_2: $x_2 = f\ c\ v$ **by** *auto*
then have *CsubM*: $C \subset\# M$ **by** *simp*
from *MBb* **have** *BsubM*: $B \subset\# M$ **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 $B - \{\#c\#\} \subset\# B$ **using** *cinB* **by** (*rule mset-less-diff-self*)
then have *DsubM*: *?D* $\subset\# M$ **using** *BsubM* **by** (*blast intro*: *mset-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: left-commute)
    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 union-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 left-commute* [*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*:*union-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 ... = *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 *left-commutative g*
shows (⋀*x y*. *h* (*g x y*) = *f x* (*h y*)) ⟹ *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A* (**is** *PROP ?P*)
proof −
interpret *left-commutative g* **by** *fact*
show *PROP ?P* **by** (*induct A*) *auto*
qed

lemma *fold-mset-rec*:
assumes *a* ∈# *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.

46.9 Image

definition [*code del*]:

image-mset f = fold-mset (op + o single o f) {#}

interpretation *image-left-comm*: *left-commutative op + o single o f*
proof qed (*simp add*:*union-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*: *union-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#}#}*.

46.10 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 (Multiset.multiset, [T]);

  fun mk-mset T [] = Const (@{const-name 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 =
     FundefLib.regroup-conv @{const-name Multiset.Mempty} @{const-name plus}
     (map (fn t => t RS eq-reflection) (@{thms union-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
⟩⟩

end

47 Nat-Infinity: Natural numbers with infinity

theory *Nat-Infinity*
imports *Main*
begin

47.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

datatype *inat* = *Fin nat* | *Infty*

notation (*xsymbols*)
Infty (∞)

notation (*HTML* **output**)
Infty (∞)

47.2 Constructors and numbers

instantiation *inat* :: {*zero*, *one*, *number*}
begin

definition
0 = *Fin 0*

definition
[*code inline*]: *1* = *Fin 1*

definition
[*code inline*, *code del*]: *number-of k* = *Fin* (*number-of k*)

instance ..

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*]: $\infty \neq 0$
by (*simp add*: *zero-inat-def*)

lemma *i0-ne-Infty* [*simp*]: $0 \neq \infty$
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*]: $\infty \neq 1$
by (*simp add*: *one-inat-def*)

lemma *i1-ne-Infty* [*simp*]: $1 \neq \infty$
by (*simp add*: *one-inat-def*)

lemma *Infty-ne-number* [*simp*]: $\infty \neq$ *number-of k*
by (*simp add*: *number-of-inat-def*)

lemma *number-ne-Infty* [*simp*]: *number-of k* $\neq \infty$
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* $\infty = \infty$
by (*simp add*: *iSuc-def*)

lemma *iSuc-ne-0* [*simp*]: *iSuc n* $\neq$ *0*
by (*simp add*: *iSuc-def zero-inat-def split*: *inat.splits*)

lemma *zero-ne-iSuc* [*simp*]: *0* $\neq$ *iSuc n*
by (*rule iSuc-ne-0* [*symmetric*])

lemma *iSuc-inject* [*simp*]: *iSuc m* = *iSuc n* $\longleftrightarrow$ *m* = *n*
by (*simp add*: *iSuc-def split*: *inat.splits*)

lemma *number-of-inat-inject* [*simp*]:
(*number-of k* :: *inat*) = *number-of l* $\longleftrightarrow$ (*number-of k* :: *nat*) = *number-of l*
by (*simp add*: *number-of-inat-def*)

47.3 Addition

instantiation *inat* :: *comm-monoid-add*
begin

definition
[*code del*]: *m* + *n* = (*case m of* $\infty \Rightarrow \infty$ | *Fin m* $\Rightarrow$ (*case n of* $\infty \Rightarrow \infty$ | *Fin n* $\Rightarrow$ *Fin* (*m* + *n*)))

lemma *plus-inat-simps* [*simp*, *code*]:
Fin m + *Fin n* = *Fin* (*m* + *n*)
∞ + *q* = ∞
q + ∞ = ∞
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*)

47.4 Multiplication

instantiation *inat* :: *comm-semiring-1*
begin

definition
times-inat-def [*code del*]:
m ∗ *n* = (*case m of* ∞ ⇒ *if n* = *0 then 0 else* ∞ | *Fin m* ⇒
(*case n of* ∞ ⇒ *if m* = *0 then 0 else* ∞ | *Fin n* ⇒ *Fin* (*m* ∗ *n*)))

lemma *times-inat-simps* [*simp*, *code*]:
Fin m ∗ *Fin n* = *Fin* (*m* ∗ *n*)
∞ ∗ ∞ = ∞
∞ ∗ *Fin n* = (*if n* = *0 then 0 else* ∞)
Fin m ∗ ∞ = (*if m* = *0 then 0 else* ∞)
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 $(0::inat) \neq 1$
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*: $of\text{-}nat\ n = Fin\ n$
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*)

47.5 Ordering

instantiation *inat* :: *ordered-ab-semigroup-add*
begin

definition
[*code del*]: $m \leq n = (case\ n\ of\ Fin\ n1 \Rightarrow (case\ m\ of\ Fin\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow False) \mid \infty \Rightarrow True)$

definition
[*code del*]: $m < n = (case\ m\ of\ Fin\ m1 \Rightarrow (case\ n\ of\ Fin\ n1 \Rightarrow m1 < n1 \mid \infty \Rightarrow True) \mid \infty \Rightarrow False)$

lemma *inat-ord-simps* [*simp*]:
$Fin\ m \leq Fin\ n \longleftrightarrow m \leq n$

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* :: *pordered-comm-semiring*
proof
fix *a b c* :: *inat*
assume *a* ≤ *b* **and** *0* ≤ *c*
thus *c* ∗ *a* ≤ *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*) ≤ *number-of n* ⟷ (*number-of m* :: *nat*) ≤ *number-of n*
(*number-of m* :: *inat*) < *number-of n* ⟷ (*number-of m* :: *nat*) < *number-of n*
by (*simp-all add*: *number-of-inat-def*)

lemma *i0-lb* [*simp*]: (*0*::*inat*) ≤ *n*
by (*simp add*: *zero-inat-def less-eq-inat-def split*: *inat.splits*)

lemma *i0-neq* [*simp*]: *n* ≤ (*0*::*inat*) ⟷ *n* = *0*
by (*simp add*: *zero-inat-def less-eq-inat-def split*: *inat.splits*)

lemma *Infty-ileE* [*elim*!]: ∞ ≤ *Fin m* ⟹ *R*
by (*simp add*: *zero-inat-def less-eq-inat-def split*: *inat.splits*)

lemma *Infty-ilessE* [*elim*!]: ∞ < *Fin m* ⟹ *R*
by *simp*

lemma *not-ilessi0* [*simp*]: ¬ *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

47.6 Well-ordering

lemma *less-FinE*:
[| $n < Fin\ m$; !!k. $n = Fin\ k \Longrightarrow k < m \Longrightarrow P$ |] $\Longrightarrow P$
by (*induct n*) *auto*

lemma *less-InftyE*:
[| $n < Infty$; !!k. $n = Fin\ k \Longrightarrow P$ |] $\Longrightarrow P$
by (*induct n*) *auto*

lemma *inat-less-induct*:
assumes *prem*: !!n. $\forall m$::*inat*. $m < n \longrightarrow P\ m \Longrightarrow P\ n$ **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
```

47.7 Traditional theorem names

```
lemmas inat-defs = zero-inat-def one-inat-def number-of-inat-def iSuc-def
  plus-inat-def less-eq-inat-def less-inat-def

lemmas inat-splits = inat.splits

end
```

48 Nested-Environment: Nested environments

```
theory Nested-Environment
imports Main
begin
```

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

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

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

48.1 The lookup operation

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

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

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

hide const lookup-option
```

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

```
theorem lookup-nil: lookup e [] = Some e
  by (cases e) simp-all

theorem lookup-val-cons: lookup (Val a) (x # xs) = None
  by simp

theorem lookup-env-cons:
  lookup (Env b es) (x # xs) =
    (case es x of
      None => None
    | Some e => lookup e xs)
  by (cases es x) simp-all

lemmas lookup.simps [simp del]
  and lookup-simps [simp] = lookup-nil lookup-val-cons lookup-env-cons

theorem lookup-eq:
  lookup env xs =
    (case xs of
      [] => Some env
    | x # xs =>
      (case env of
        Val a => None
      | Env b es =>
          (case es x of
```

```
        None => None
      | Some e => lookup e xs)))
  by (simp split: list.split env.split)
```

Displaced *lookup* operations, relative to a certain base path prefix, may be reduced as follows. There are two cases, depending whether the environment actually extends far enough to follow the base path.

```
theorem lookup-append-none:
  assumes lookup env xs = None
  shows lookup env (xs @ ys) = None
  using assms
proof (induct xs arbitrary: env)
  case Nil
  then have False by simp
  then show ?case ..
next
  case (Cons x xs)
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with Env show ?thesis by simp
    next
      case (Some e)
      note es = ‹es x = Some e›
      show ?thesis
      proof (cases lookup e xs)
        case None
        then have lookup e (xs @ ys) = None by (rule Cons.hyps)
        with Env Some show ?thesis by simp
      next
        case Some
        with Env es have False using Cons.prems by simp
        then show ?thesis ..
      qed
    qed
  qed
qed

theorem lookup-append-some:
  assumes lookup env xs = Some e
  shows lookup env (xs @ ys) = lookup e ys
  using assms
proof (induct xs arbitrary: env e)
```

```
  case Nil
  then have env = e by simp
  then show lookup env ([] @ ys) = lookup e ys by simp
next
  case (Cons x xs)
  note asm = ‹lookup env (x # xs) = Some e›
  show lookup env ((x # xs) @ ys) = lookup e ys
  proof (cases env)
    case (Val a)
    with asm have False by simp
    then show ?thesis ..
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with asm Env have False by simp
      then show ?thesis ..
    next
      case (Some e')
      note es = ‹es x = Some e'›
      show ?thesis
      proof (cases lookup e' xs)
        case None
        with asm Env es have False by simp
        then show ?thesis ..
      next
        case Some
        with asm Env es have lookup e' xs = Some e
          by simp
        then have lookup e' (xs @ ys) = lookup e ys by (rule Cons.hyps)
        with Env es show ?thesis by simp
      qed
    qed
  qed
qed
```

Successful *lookup* deeper down an environment structure means we are able to peek further up as well. Note that this is basically just the contrapositive statement of *lookup-append-none* above.

```
theorem lookup-some-append:
  assumes lookup env (xs @ ys) = Some e
  shows ∃ e. lookup env xs = Some e
proof −
  from assms have lookup env (xs @ ys) ≠ None by simp
  then have lookup env xs ≠ None
    by (rule contrapos-nn) (simp only: lookup-append-none)
  then show ?thesis by (simp)
qed
```

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

theorem *lookup-some-upper*:
  assumes *lookup env* (*xs* @ *y* # *ys*) = *Some e*
  shows ∃ *b′ es′ env′*.
    lookup env xs = *Some* (*Env b′ es′*) ∧
    es′ y = *Some env′* ∧
    lookup env′ ys = *Some e*
  using *assms*
proof (*induct xs arbitrary*: *env e*)
  case *Nil*
  from *Nil.prems* **have** *lookup env* (*y* # *ys*) = *Some e*
    by *simp*
  then obtain *b′ es′ env′* **where**
      env: *env* = *Env b′ es′* **and**
      es′: *es′ y* = *Some env′* **and**
      look′: *lookup env′ ys* = *Some e*
    by (*auto simp add*: *lookup-eq split*: *option.splits env.splits*)
  from *env* **have** *lookup env* [] = *Some* (*Env b′ es′*) **by** *simp*
  with *es′ look′* **show** *?case* **by** *blast*
next
  case (*Cons x xs*)
  from *Cons.prems*
  obtain *b′ es′ env′* **where**
      env: *env* = *Env b′ es′* **and**
      es′: *es′ x* = *Some env′* **and**
      look′: *lookup env′* (*xs* @ *y* # *ys*) = *Some e*
    by (*auto simp add*: *lookup-eq split*: *option.splits env.splits*)
  from *Cons.hyps* [*OF look′*] **obtain** *b″ es″ env″* **where**
      upper′: *lookup env′ xs* = *Some* (*Env b″ es″*) **and**
      es″: *es″ y* = *Some env″* **and**
      look″: *lookup env″ ys* = *Some e*
    by *blast*
  from *env es′ upper′* **have** *lookup env* (*x* # *xs*) = *Some* (*Env b″ es″*)
    by *simp*
  with *es″ look″* **show** *?case* **by** *blast*
qed

48.2 The update operation

Update at a certain position in a nested environment may either delete an existing entry, or overwrite an existing one. Note that update at undefined positions is simple absorbed, i.e. the environment is left unchanged.

consts
  update :: *′c list* => (*′a*, *′b*, *′c*) *env option*
    => (*′a*, *′b*, *′c*) *env* => (*′a*, *′b*, *′c*) *env*
  update-option :: *′c list* => (*′a*, *′b*, *′c*) *env option*

```
  => ('a, 'b, 'c) env option => ('a, 'b, 'c) env option

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

hide const update-option
```

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

```
theorem update-nil-none: update [] None env = env
  by (cases env) simp-all

theorem update-nil-some: update [] (Some e) env = e
  by (cases env) simp-all

theorem update-cons-val: update (x # xs) opt (Val a) = Val a
  by simp

theorem update-cons-nil-env:
    update [x] opt (Env b es) = Env b (es (x := opt))
  by (cases es x) simp-all

theorem update-cons-cons-env:
  update (x # y # ys) opt (Env b es) =
    Env b (es (x :=
      (case es x of
        None => None
      | Some e => Some (update (y # ys) opt e))))
  by (cases es x) simp-all

lemmas update.simps [simp del]
  and update-simps [simp] = update-nil-none update-nil-some
    update-cons-val update-cons-nil-env update-cons-cons-env

lemma update-eq:
  update xs opt env =
    (case xs of
      [] =>
        (case opt of
```

```
        None => env
      | Some e => e)
  | x # xs =>
      (case env of
        Val a => Val a
      | Env b es =>
          (case xs of
            [] => Env b (es (x := opt))
          | y # ys =>
              Env b (es (x :=
                (case es x of
                  None => None
                | Some e => Some (update (y # ys) opt e)))))))
 by (simp split: list.split env.split option.split)
```

The most basic correspondence of *lookup* and *update* states that after *update* at a defined position, subsequent *lookup* operations would yield the new value.

```
theorem lookup-update-some:
  assumes lookup env xs = Some e
  shows lookup (update xs (Some env') env) xs = Some env'
  using assms
proof (induct xs arbitrary: env e)
  case Nil
  then have env = e by simp
  then show ?case by simp
next
  case (Cons x xs)
  note hyp = Cons.hyps
    and asm = ‹lookup env (x # xs) = Some e›
  show ?case
  proof (cases env)
    case (Val a)
    with asm have False by simp
    then show ?thesis ..
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with asm Env have False by simp
      then show ?thesis ..
    next
      case (Some e')
      note es = ‹es x = Some e'›
      show ?thesis
      proof (cases xs)
        case Nil
        with Env show ?thesis by simp
```

```
      next
        case (Cons x' xs')
        from asm Env es have lookup e' xs = Some e by simp
       then have lookup (update xs (Some env') e') xs = Some env' by (rule hyp)
        with Env es Cons show ?thesis by simp
      qed
    qed
  qed
qed
```

The properties of displaced *update* operations are analogous to those of *lookup* above. There are two cases: below an undefined position *update* is absorbed altogether, and below a defined positions *update* affects subsequent *lookup* operations in the obvious way.

```
theorem update-append-none:
  assumes lookup env xs = None
  shows update (xs @ y # ys) opt env = env
  using assms
proof (induct xs arbitrary: env)
  case Nil
  then have False by simp
  then show ?case ..
next
  case (Cons x xs)
  note hyp = Cons.hyps
    and asm = ‹lookup env (x # xs) = None›
  show update ((x # xs) @ y # ys) opt env = env
  proof (cases env)
    case (Val a)
    then show ?thesis by simp
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      note es = ‹es x = None›
      show ?thesis
        by (cases xs) (simp-all add: es Env fun-upd-idem-iff)
    next
      case (Some e)
      note es = ‹es x = Some e›
      show ?thesis
      proof (cases xs)
        case Nil
        with asm Env Some have False by simp
        then show ?thesis ..
      next
        case (Cons x' xs')
        from asm Env es have lookup e xs = None by simp
```

```
      then have update (xs @ y # ys) opt e = e by (rule hyp)
      with Env es Cons show update ((x # xs) @ y # ys) opt env = env
        by (simp add: fun-upd-idem-iff)
    qed
  qed
 qed
qed

theorem update-append-some:
  assumes lookup env xs = Some e
  shows lookup (update (xs @ y # ys) opt env) xs = Some (update (y # ys) opt
e)
  using assms
proof (induct xs arbitrary: env e)
  case Nil
  then have env = e by simp
  then show ?case by simp
next
  case (Cons x xs)
  note hyp = Cons.hyps
    and asm = ‹lookup env (x # xs) = Some e›
  show lookup (update ((x # xs) @ y # ys) opt env) (x # xs) =
      Some (update (y # ys) opt e)
  proof (cases env)
    case (Val a)
    with asm have False by simp
    then show ?thesis ..
  next
    case (Env b es)
    show ?thesis
    proof (cases es x)
      case None
      with asm Env have False by simp
      then show ?thesis ..
    next
      case (Some e')
      note es = ‹es x = Some e'›
      show ?thesis
      proof (cases xs)
        case Nil
        with asm Env es have e = e' by simp
        with Env es Nil show ?thesis by simp
      next
        case (Cons x' xs')
        from asm Env es have lookup e' xs = Some e by simp
        then have lookup (update (xs @ y # ys) opt e') xs =
          Some (update (y # ys) opt e) by (rule hyp)
        with Env es Cons show ?thesis by simp
      qed
```

```
    qed
  qed
qed
```

Apparently, *update* does not affect the result of subsequent *lookup* operations at independent positions, i.e. in case that the paths for *update* and *lookup* fork at a certain point.

```
theorem lookup-update-other:
  assumes neq: y ≠ (z::'c)
  shows lookup (update (xs @ z # zs) opt env) (xs @ y # ys) =
    lookup env (xs @ y # ys)
proof (induct xs arbitrary: env)
  case Nil
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case Env
    show ?thesis
    proof (cases zs)
      case Nil
      with neq Env show ?thesis by simp
    next
      case Cons
      with neq Env show ?thesis by simp
    qed
  qed
next
  case (Cons x xs)
  note hyp = Cons.hyps
  show ?case
  proof (cases env)
    case Val
    then show ?thesis by simp
  next
    case (Env y es)
    show ?thesis
    proof (cases xs)
      case Nil
      show ?thesis
      proof (cases es x)
        case None
        with Env Nil show ?thesis by simp
      next
        case Some
        with neq hyp and Env Nil show ?thesis by simp
      qed
    next
```

```
      case (Cons x' xs')
      show ?thesis
      proof (cases es x)
        case None
        with Env Cons show ?thesis by simp
      next
        case Some
        with neq hyp and Env Cons show ?thesis by simp
      qed
    qed
  qed
qed
```

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 ⟷
```

$x = y \wedge (\forall z{\in}UNIV.\ case\ f\ z$
$\quad of\ None \Rightarrow (case\ g\ z$
$\quad\quad of\ None \Rightarrow True \mid Some\ \text{-} \Rightarrow False)$
$\quad \mid Some\ a \Rightarrow (case\ g\ z$
$\quad\quad of\ None \Rightarrow False \mid Some\ b \Rightarrow a = b))$ **by** *simp*
qed *simp-all*

lemma [*code*, *code del*]:
$(Code\text{-}Eval.term\text{-}of :: ('a{::}\{term\text{-}of, type\}, 'b{::}\{term\text{-}of, type\}, 'c{::}\{term\text{-}of, type\}) env \Rightarrow term) = Code\text{-}Eval.term\text{-}of$ **..**

end

49 Option-ord: Canonical order on option type

theory *Option-ord*
imports *Option Main*
begin

instantiation *option* :: (*preorder*) *preorder*
begin

definition *less-eq-option* **where**
[*code del*]: $x \leq y \longleftrightarrow (case\ x\ of\ None \Rightarrow True \mid Some\ x \Rightarrow (case\ y\ of\ None \Rightarrow False \mid Some\ y \Rightarrow x \leq y))$

definition *less-option* **where**
[*code del*]: $x < y \longleftrightarrow (case\ y\ of\ None \Rightarrow False \mid Some\ y \Rightarrow (case\ x\ of\ None \Rightarrow True \mid Some\ x \Rightarrow x < y))$

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 $P :: 'a\ option \Rightarrow bool$ **and** $z :: 'a\ option$
assume H: $\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x$

```
  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
```

50 Permutation: Permutations

```
theory Permutation
imports Main Multiset
begin

inductive
  perm :: 'a list => 'a list => bool  (- <~~> -  [50, 50] 50)
  where
    Nil  [intro!]: [] <~~> []
  | swap [intro!]: y # x # l <~~> x # y # l
  | Cons [intro!]: xs <~~> ys ==> z # xs <~~> z # ys
  | trans [intro]: xs <~~> ys ==> ys <~~> zs ==> xs <~~> zs

lemma perm-refl [iff]: l <~~> l
  by (induct l) auto
```

50.1 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*

50.2 Ways of making new permutations

We can insert the head anywhere in the list.

lemma *perm-append-Cons*: *a* # *xs* @ *ys* <~~> *xs* @ *a* # *ys*
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*)

50.3 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*)

50.4 Removing elements

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

lemma *perm-remove*: *x* ∈ *set ys* ==> *ys* <~~> *x* # *remove x ys*
by (*induct ys*) *auto*

lemma *remove-commute*: *remove x* (*remove y l*) = *remove y* (*remove x l*)
by (*induct l*) *auto*

lemma *multiset-of-remove* [*simp*]:
multiset-of (*remove a x*) = *multiset-of x* − {#*a*#}
apply (*induct x*)
apply (*auto simp*: *multiset-eq-conv-count-eq*)
done

Congruence rule

lemma *perm-remove-perm*: *xs* <~~> *ys* ==> *remove z xs* <~~> *remove z ys*
by (*induct pred*: *perm*) *auto*

lemma *remove-hd* [*simp*]: *remove z* (*z* # *xs*) = *xs*
by *auto*

lemma *cons-perm-imp-perm*: *z* # *xs* <~~> *z* # *ys* ==> *xs* <~~> *ys*
by (*drule-tac z* = *z* **in** *perm-remove-perm*) *auto*

lemma *cons-perm-eq* [*iff*]: (*z*#*xs* <~~> *z*#*ys*) = (*xs* <~~> *ys*)
by (*blast intro*: *cons-perm-imp-perm*)

lemma *append-perm-imp-perm*: *zs* @ *xs* <~~> *zs* @ *ys* ==> *xs* <~~> *ys*
apply (*induct zs arbitrary*: *xs ys rule*: *rev-induct*)
apply (*simp-all* (*no-asm-use*))
apply *blast*
done

lemma *perm-append1-eq* [*iff*]: (*zs* @ *xs* <~~> *zs* @ *ys*) = (*xs* <~~> *ys*)
by (*blast intro*: *append-perm-imp-perm perm-append1*)

lemma *perm-append2-eq* [*iff*]: (*xs* @ *zs* <~~> *ys* @ *zs*) = (*xs* <~~> *ys*)
apply (*safe intro*!: *perm-append2*)
apply (*rule append-perm-imp-perm*)
apply (*rule perm-append-swap* [*THEN perm.trans*])
— the previous step helps this *blast* call succeed quickly
apply (*blast intro*: *perm-append-swap*)
done

lemma *multiset-of-eq-perm*: (*multiset-of xs* = *multiset-of ys*) = (*xs* <~~> *ys*)
apply (*rule iffI*)
apply (*erule-tac* [*2*] *perm.induct*, *simp-all add*: *union-ac*)
apply (*erule rev-mp*, *rule-tac x*=*ys* **in** *spec*)
apply (*induct-tac xs*, *auto*)
apply (*erule-tac x* = *remove a x* **in** *allE*, *drule sym*, *simp*)
apply (*subgoal-tac a* ∈ *set x*)
apply (*drule-tac z*=*a* **in** *perm.Cons*)
apply (*erule perm.trans*, *rule perm-sym*, *erule perm-remove*)
apply (*drule-tac f*=*set-of* **in** *arg-cong*, *simp*)
done

lemma *multiset-of-le-perm-append*:
(*multiset-of xs* ≤# *multiset-of ys*) = (∃*zs*. *xs* @ *zs* <~~> *ys*)
apply (*auto simp*: *multiset-of-eq-perm*[*THEN sym*] *mset-le-exists-conv*)
apply (*insert surj-multiset-of*, *drule surjD*)
apply (*blast intro*: *sym*)+
done

lemma *perm-set-eq*: *xs* <~~> *ys* ==> *set xs* = *set ys*
by (*metis multiset-of-eq-perm multiset-of-eq-setD*)

lemma *perm-distinct-iff*: *xs* <~~> *ys* ==> *distinct xs* = *distinct ys*
apply (*induct pred*: *perm*)
apply *simp-all*
apply *fastsimp*
apply (*metis perm-set-eq*)
done

lemma *eq-set-perm-remdups*: *set xs* = *set ys* ==> *remdups xs* <~~> *remdups ys*
apply (*induct xs arbitrary*: *ys rule*: *length-induct*)
apply (*case-tac remdups xs*, *simp*, *simp*)
apply (*subgoal-tac a* : *set* (*remdups ys*))
prefer *2* **apply** (*metis set.simps*(*2*) *insert-iff set-remdups*)
apply (*drule split-list*) **apply**(*elim exE conjE*)
apply (*drule-tac x*=*list* **in** *spec*) **apply**(*erule impE*) **prefer** *2*
apply (*drule-tac x*=*ysa*@*zs* **in** *spec*) **apply**(*erule impE*) **prefer** *2*
apply *simp*
apply (*subgoal-tac a*#*list* <~~> *a*#*ysa*@*zs*)
apply (*metis Cons-eq-appendI perm-append-Cons trans*)
apply (*metis Cons Cons-eq-appendI distinct.simps*(*2*)

distinct-remdups distinct-remdups-id perm-append-swap perm-distinct-iff)
apply (*subgoal-tac set* (*a#list*) = *set* (*ysa@a#zs*) & *distinct* (*a#list*) & *distinct* (*ysa@a#zs*))
apply (*fastsimp simp add*: *insert-ident*)
apply (*metis distinct-remdups set-remdups*)
apply (*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

51 Primes: Primality on nat

theory *Primes*
imports *Complex-Main*
begin

definition
coprime :: *nat* => *nat* => *bool* **where**
coprime m n ⟷ *gcd m n* = *1*

definition
prime :: *nat* ⇒ *bool* **where**
[*code del*]: *prime p* ⟷ (*1* < *p* ∧ (∀ *m*. *m dvd p* −−> *m* = *1* ∨ *m* = *p*))

lemma *two-is-prime*: *prime 2*
apply (*auto simp add*: *prime-def*)
apply (*case-tac m*)
apply (*auto dest*!: *dvd-imp-le*)
done

lemma *prime-imp-relprime*: *prime p* ==> ¬ *p dvd n* ==> *gcd p n* = *1*
apply (*auto simp add*: *prime-def*)
apply (*metis One-nat-def gcd-dvd1 gcd-dvd2*)
done

This theorem leads immediately to a proof of the uniqueness of factorization. If *p* divides a product of primes then it is one of those primes.

lemma *prime-dvd-mult*: *prime p* ==> *p dvd m* * *n* ==> *p dvd m* ∨ *p dvd n*
by (*blast intro*: *relprime-dvd-mult prime-imp-relprime*)

lemma *prime-dvd-square*: *prime p ==> p dvd m^Suc (Suc 0) ==> p dvd m*
by (*auto dest*: *prime-dvd-mult*)

lemma *prime-dvd-power-two*: *prime p ==> p dvd* m^2 *==> p dvd m*
by (*rule prime-dvd-square*) (*simp-all add*: *power2-eq-square*)

lemma *exp-eq-1*:(*x*::*nat*) *^n = 1* $\longleftrightarrow$ *x = 1* $\vee$ *n = 0*
by (*induct n*, *auto*)

lemma *exp-mono-lt*: (*x*::*nat*) *^ (Suc n) < y ^ (Suc n)* $\longleftrightarrow$ *x < y*
by(*metis linorder-not-less not-less0 power-le-imp-le-base power-less-imp-less-base*)

lemma *exp-mono-le*: (*x*::*nat*) *^ (Suc n)* $\leq$ *y ^ (Suc n)* $\longleftrightarrow$ *x* $\leq$ *y*
by (*simp only*: *linorder-not-less*[*symmetric*] *exp-mono-lt*)

lemma *exp-mono-eq*: (*x*::*nat*) *^ Suc n = y ^ Suc n* $\longleftrightarrow$ *x = y*
using *power-inject-base*[*of x n y*] **by** *auto*

lemma *even-square*: **assumes** *e*: *even* (*n*::*nat*) **shows** $\exists$ *x. n ^ 2 = 4*x*
proof−
from *e* **have** *2 dvd n* **by** *presburger*
then obtain *k* **where** *k*: *n = 2*k* **using** *dvd-def* **by** *auto*
hence *n^2 = 4* (k^2)* **by** (*simp add*: *power2-eq-square*)
thus *?thesis* **by** *blast*
qed

lemma *odd-square*: **assumes** *e*: *odd* (*n*::*nat*) **shows** $\exists$ *x. n ^ 2 = 4*x + 1*
proof−
from *e* **have** *np*: *n > 0* **by** *presburger*
from *e* **have** *2 dvd (n − 1)* **by** *presburger*
then obtain *k* **where** *n − 1 = 2*k* **using** *dvd-def* **by** *auto*
hence *k*: *n = 2*k + 1* **using** *e* **by** *presburger*
hence *n^2 = 4* (k^2 + k) + 1* **by** *algebra*
thus *?thesis* **by** *blast*
qed

lemma *diff-square*: (*x*::*nat*) *^2 − y^2 = (x+y)*(x − y)*
proof−
have *x* $\leq$ *y* $\vee$ *y* $\leq$ *x* **by** (*rule nat-le-linear*)
moreover
{assume *le*: *x* $\leq$ *y*
hence *x ^2* $\leq$ *y^2* **by** (*simp only*: *numeral-2-eq-2 exp-mono-le Let-def*)
with *le* **have** *?thesis* **by** *simp* **}**
moreover
{assume *le*: *y* $\leq$ *x*
hence *le2*: *y ^2* $\leq$ *x^2* **by** (*simp only*: *numeral-2-eq-2 exp-mono-le Let-def*)
from *le* **have** $\exists$ *z. y + z = x* **by** *presburger*

then obtain *z* **where** *z*: $x = y + z$ **by** *blast*
from *le2* **have** $\exists z.\ x\hat{}2 = y\hat{}2 + z$ **by** *presburger*
then obtain *z2* **where** *z2*: $x\hat{}2 = y\hat{}2 + z2$ **by** *blast*
from *z z2* **have** *?thesis* **apply** *simp* **by** *algebra* **}**
ultimately show *?thesis* **by** *blast*
qed

Elementary theory of divisibility

lemma *divides-ge*: (*a*::*nat*) *dvd b* $\Longrightarrow$ $b = 0 \vee a \leq b$ **unfolding** *dvd-def* **by** *auto*
lemma *divides-antisym*: (*x*::*nat*) *dvd y* $\wedge$ *y dvd x* $\longleftrightarrow$ $x = y$
using *dvd-anti-sym*[*of x y*] **by** *auto*

lemma *divides-add-revr*: **assumes** *da*: (*d*::*nat*) *dvd a* **and** *dab*:*d dvd* $(a + b)$
shows *d dvd b*
proof−
from *da* **obtain** *k* **where** *k*:$a = d*k$ **by** (*auto simp add*: *dvd-def*)
from *dab* **obtain** k' **where** k': $a + b = d*k'$ **by** (*auto simp add*: *dvd-def*)
from *k* k' **have** $b = d*(k' - k)$ **by** (*simp add* : *diff-mult-distrib2*)
thus *?thesis* **unfolding** *dvd-def* **by** *blast*
qed

declare *nat-mult-dvd-cancel-disj*[*presburger*]
lemma *nat-mult-dvd-cancel-disj'*[*presburger*]:
(*m*::*nat*)$*k$ *dvd* $n*k$ $\longleftrightarrow$ $k = 0 \vee$ *m dvd n* **unfolding** *mult-commute*[*of m k*] *mult-commute*[*of n k*] **by** *presburger*

lemma *divides-mul-l*: (*a*::*nat*) *dvd b* ==> $(c * a)$ *dvd* $(c * b)$
by *presburger*

lemma *divides-mul-r*: (*a*::*nat*) *dvd b* ==> $(a * c)$ *dvd* $(b * c)$ **by** *presburger*
lemma *divides-cases*: (*n*::*nat*) *dvd m* ==> $m = 0 \vee m = n \vee 2 * n <= m$
by (*auto simp add*: *dvd-def*)

lemma *divides-div-not*: (*x*::*nat*) = $(q * n) + r$ $\Longrightarrow$ $0 < r$ $\Longrightarrow$ $r < n$ ==> ~(*n dvd x*)
proof(*auto simp add*: *dvd-def*)
fix *k* **assume** *H*: $0 < r\ r < n\ q * n + r = n * k$
from *H*(*3*) **have** *r*: $r = n*(k - q)$ **by**(*simp add*: *diff-mult-distrib2 mult-commute*)
{assume $k - q = 0$ **with** *r H*(*1*) **have** *False* **by** *simp***}**
moreover
{assume $k - q \neq 0$ **with** *r* **have** $r \geq n$ **by** *auto*
with *H*(*2*) **have** *False* **by** *simp***}**
ultimately show *False* **by** *blast*
qed
lemma *divides-exp*: (*x*::*nat*) *dvd y* ==> *x* ^ *n dvd y* ^ *n*
by (*auto simp add*: *power-mult-distrib dvd-def*)

lemma *divides-exp2*: $n \neq 0$ $\Longrightarrow$ (*x*::*nat*) ^ *n dvd y* $\Longrightarrow$ *x dvd y*
by (*induct n* ,*auto simp add*: *dvd-def*)

fun *fact* :: *nat* ⇒ *nat* **where**
  fact 0 = *1*
| *fact* (*Suc n*) = *Suc n* ∗ *fact n*

lemma *fact-lt*: *0* < *fact n* **by**(*induct n*, *simp-all*)
lemma *fact-le*: *fact n* ≥ *1* **using** *fact-lt*[*of n*] **by** *simp*
lemma *fact-mono*: **assumes** *le*: *m* ≤ *n* **shows** *fact m* ≤ *fact n*
proof−
  from *le* **have** ∃ *i*. *n* = *m*+*i* **by** *presburger*
  then obtain *i* **where** *i*: *n* = *m*+*i* **by** *blast*
  have *fact m* ≤ *fact* (*m* + *i*)
  proof(*induct m*)
    case *0* **thus** *?case* **using** *fact-le*[*of i*] **by** *simp*
  next
    case (*Suc m*)
    have *fact* (*Suc m*) = *Suc m* ∗ *fact m* **by** *simp*
    have *th1*: *Suc m* ≤ *Suc* (*m* + *i*) **by** *simp*
    from *mult-le-mono*[*of Suc m Suc* (*m*+*i*) *fact m fact* (*m*+*i*), *OF th1 Suc.hyps*]
    show *?case* **by** *simp*
  qed
  thus *?thesis* **using** *i* **by** *simp*
qed

lemma *divides-fact*: *1* <= *p* ⟹ *p* <= *n* ==> *p dvd fact n*
proof(*induct n arbitrary*: *p*)
  case *0* **thus** *?case* **by** *simp*
next
  case (*Suc n p*)
  from *Suc.prems* **have** *p* = *Suc n* ∨ *p* ≤ *n* **by** *presburger*
  moreover
  {assume *p* = *Suc n* **hence** *?case* **by** (*simp only*: *fact.simps dvd-triv-left*)**}**
  moreover
  {assume *p* ≤ *n*
    with *Suc.prems*(*1*) *Suc.hyps* **have** *th*: *p dvd fact n* **by** *simp*
    from *dvd-mult*[*OF th*] **have** *?case* **by** (*simp only*: *fact.simps*) **}**
  ultimately show *?case* **by** *blast*
qed

declare *dvd-triv-left*[*presburger*]
declare *dvd-triv-right*[*presburger*]
lemma *divides-rexp*:
  x dvd y ⟹ (*x*::*nat*) *dvd* (*y*^(*Suc n*)) **by** (*simp add*: *dvd-mult2*[*of x y*])

Coprimality

lemma *coprime*: *coprime a b* ⟷ (∀ *d*. *d dvd a* ∧ *d dvd b* ⟷ *d* = *1*)
using *gcd-unique*[*of 1 a b*, *simplified*] **by** (*auto simp add*: *coprime-def*)
lemma *coprime-commute*: *coprime a b* ⟷ *coprime b a* **by** (*simp add*: *coprime-def gcd-commute*)

lemma *coprime-bezout*: *coprime a b* $\longleftrightarrow$ ($\exists$ *x y*. *a* $*$ *x* $-$ *b* $*$ *y* $=$ *1* $\vee$ *b* $*$ *x* $-$ *a* $*$ *y* $=$ *1*)
using *coprime-def gcd-bezout* **by** *auto*

lemma *coprime-divprod*: *d dvd a* $*$ *b* $\Longrightarrow$ *coprime d a* $\Longrightarrow$ *d dvd b*
using *relprime-dvd-mult-iff*[*of d a b*] **by** (*auto simp add*: *coprime-def mult-commute*)

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

lemma *gcd-coprime*:
assumes *z*: *gcd a b* $\neq$ *0* **and** *a*: *a* $=$ *a'* $*$ *gcd a b* **and** *b*: *b* $=$ *b'* $*$ *gcd a b*
shows *coprime a' b'*
proof$-$
let *?g* $=$ *gcd a b*
{assume *bz*: *a* $=$ *0* **from** *b bz z a* **have** *?thesis* **by** (*simp add*: *gcd-zero coprime-def*)**}**
moreover
{assume *az*: *a*$\neq$ *0*
from *z* **have** *z'*: *?g* $>$ *0* **by** *simp*
from *bezout-gcd-strong*[*OF az*, *of b*]
obtain *x y* **where** *xy*: *a*$*$*x* $=$ *b*$*$*y* $+$ *?g* **by** *blast*
from *xy a b* **have** *?g* $*$ *a'*$*$*x* $=$ *?g* $*$ (*b'*$*$*y* $+$ *1*) **by** (*simp add*: *algebra-simps*)
hence *?g* $*$ (*a'*$*$*x*) $=$ *?g* $*$ (*b'*$*$*y* $+$ *1*) **by** (*simp add*: *mult-assoc*)
hence *a'*$*$*x* $=$ (*b'*$*$*y* $+$ *1*)
by (*simp only*: *nat-mult-eq-cancel1*[*OF z'*])
hence *a'*$*$*x* $-$ *b'*$*$*y* $=$ *1* **by** *simp*
with *coprime-bezout*[*of a' b'*] **have** *?thesis* **by** *auto***}**
ultimately show *?thesis* **by** *blast*
qed
lemma *coprime-0*: *coprime d 0* $\longleftrightarrow$ *d* $=$ *1* **by** (*simp add*: *coprime-def*)
lemma *coprime-mul*: **assumes** *da*: *coprime d a* **and** *db*: *coprime d b*
shows *coprime d* (*a* $*$ *b*)
proof$-$
from *da* **have** *th*: *gcd a d* $=$ *1* **by** (*simp add*: *coprime-def gcd-commute*)
from *gcd-mult-cancel*[*of a d b*, *OF th*] *db*[*unfolded coprime-def*] **have** *gcd d* (*a*$*$*b*) $=$ *1*
by (*simp add*: *gcd-commute*)
thus *?thesis* **unfolding** *coprime-def* **.**
qed
lemma *coprime-lmul2*: **assumes** *dab*: *coprime d* (*a* $*$ *b*) **shows** *coprime d b*
using *prems* **unfolding** *coprime-bezout*
apply *clarsimp*
apply (*case-tac d* $*$ *x* $-$ *a* $*$ *b* $*$ *y* $=$ *Suc 0* , *simp-all*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*rule-tac x*=*a*$*$*y* **in** *exI*)
apply (*simp add*: *mult-ac*)

apply (*rule-tac x=a∗x* **in** *exI*)
apply (*rule-tac x=y* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma *coprime-rmul2*: *coprime d* (*a* ∗ *b*) ⟹ *coprime d a*
unfolding *coprime-bezout*
apply *clarsimp*
apply (*case-tac d* ∗ *x* − *a* ∗ *b* ∗ *y* = *Suc 0* , *simp-all*)
apply (*rule-tac x=x* **in** *exI*)
apply (*rule-tac x=b∗y* **in** *exI*)
apply (*simp add*: *mult-ac*)
apply (*rule-tac x=b∗x* **in** *exI*)
apply (*rule-tac x=y* **in** *exI*)
apply (*simp add*: *mult-ac*)
done
lemma *coprime-mul-eq*: *coprime d* (*a* ∗ *b*) ⟷ *coprime d a* ∧ *coprime d b*
using *coprime-rmul2*[*of d a b*] *coprime-lmul2*[*of d a b*] *coprime-mul*[*of d a b*]
by *blast*

lemma *gcd-coprime-exists*:
assumes *nz*: *gcd a b* ≠ *0*
shows ∃ *a′ b′*. *a* = *a′* ∗ *gcd a b* ∧ *b* = *b′* ∗ *gcd a b* ∧ *coprime a′ b′*
proof−
let *?g* = *gcd a b*
from *gcd-dvd1*[*of a b*] *gcd-dvd2*[*of a b*]
obtain *a′ b′* **where** *a* = *?g∗a′* *b* = *?g∗b′* **unfolding** *dvd-def* **by** *blast*
hence *ab′*: *a* = *a′∗?g b* = *b′∗?g* **by** *algebra+*
from *ab′ gcd-coprime*[*OF nz ab′*] **show** *?thesis* **by** *blast*
qed

lemma *coprime-exp*: *coprime d a* ==> *coprime d* (*a*^*n*)
by(*induct n*, *simp-all add*: *coprime-mul*)

lemma *coprime-exp-imp*: *coprime a b* ==> *coprime* (*a* ^*n*) (*b* ^*n*)
by (*induct n*, *simp-all add*: *coprime-mul-eq coprime-commute coprime-exp*)
lemma *coprime-refl*[*simp*]: *coprime n n* ⟷ *n* = *1* **by** (*simp add*: *coprime-def*)
lemma *coprime-plus1*[*simp*]: *coprime* (*n* + *1*) *n*
apply (*simp add*: *coprime-bezout*)
apply (*rule exI*[**where** *x=1*])
apply (*rule exI*[**where** *x=1*])
apply *simp*
done
lemma *coprime-minus1*: *n* ≠ *0* ==> *coprime* (*n* − *1*) *n*
using *coprime-plus1*[*of n* − *1*] *coprime-commute*[*of n* − *1 n*] **by** *auto*

lemma *bezout-gcd-pow*: ∃ *x y*. *a* ^*n* ∗ *x* − *b* ^ *n* ∗ *y* = *gcd a b* ^ *n* ∨ *b* ^ *n* ∗ *x* − *a* ^ *n* ∗ *y* = *gcd a b* ^ *n*
proof−

let *?g = gcd a b*
{assume *z*: *?g = 0* **hence** *?thesis*
apply (*cases n*, *simp*)
apply *arith*
apply (*simp only*: *z power-0-Suc*)
apply (*rule exI*[**where** *x=0*])
apply (*rule exI*[**where** *x=0*])
by *simp***}**
moreover
{assume *z*: *?g ≠ 0*
from *gcd-dvd1*[*of a b*] *gcd-dvd2*[*of a b*] **obtain** *a' b'* **where**
ab': *a = a'*?g b = b'*?g* **unfolding** *dvd-def* **by** (*auto simp add*: *mult-ac*)
hence *ab''*: *?g*a' = a ?g * b' = b* **by** *algebra+*
from *coprime-exp-imp*[*OF gcd-coprime*[*OF z ab'*], *unfolded coprime-bezout*, *of n*]
obtain *x y* **where** *a'^n * x − b'^n * y = 1 ∨ b'^n * x − a'^n * y = 1* **by** *blast*
hence *?g^n * (a'^n * x − b'^n * y) = ?g^n ∨ ?g^n*(b'^n * x − a'^n * y) = ?g^n*
using *z* **by** *auto*
then have *a^n * x − b^n * y = ?g^n ∨ b^n * x − a^n * y = ?g^n*
using *z ab''* **by** (*simp only*: *power-mult-distrib*[*symmetric*] *diff-mult-distrib2 mult-assoc*[*symmetric*])
hence *?thesis* **by** *blast* **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *gcd-exp*: *gcd (a^n) (b^n) = gcd a b^n*
proof−
let *?g = gcd (a^n) (b^n)*
let *?gn = gcd a b^n*
{fix *e* **assume** *H*: *e dvd a^n e dvd b^n*
from *bezout-gcd-pow*[*of a n b*] **obtain** *x y*
where *xy*: *a ^ n * x − b ^ n * y = ?gn ∨ b ^ n * x − a ^ n * y = ?gn* **by** *blast*
from *nat-dvd-diff* [*OF dvd-mult2*[*OF H(1)*, *of x*] *dvd-mult2*[*OF H(2)*, *of y*]]
nat-dvd-diff [*OF dvd-mult2*[*OF H(2)*, *of x*] *dvd-mult2*[*OF H(1)*, *of y*]] *xy*
have *e dvd ?gn* **by** (*cases a ^ n * x − b ^ n * y = gcd a b ^ n*, *simp-all*)**}**
hence *th*: *∀ e. e dvd a^n ∧ e dvd b^n ⟶ e dvd ?gn* **by** *blast*
from *divides-exp*[*OF gcd-dvd1*[*of a b*], *of n*] *divides-exp*[*OF gcd-dvd2*[*of a b*], *of n*] *th*
gcd-unique **have** *?gn = ?g* **by** *blast* **thus** *?thesis* **by** *simp*
qed

lemma *coprime-exp2*: *coprime (a ^ Suc n) (b^ Suc n) ⟷ coprime a b*
by (*simp only*: *coprime-def gcd-exp exp-eq-1*) *simp*

lemma *division-decomp*: **assumes** *dc*: (*a*::*nat*) *dvd b * c*
shows *∃ b' c'. a = b' * c' ∧ b' dvd b ∧ c' dvd c*

proof−
let *?g = gcd a b*
{assume *?g = 0* **with** *dc* **have** *?thesis* **apply** (*simp add*: *gcd-zero*)
apply (*rule exI*[**where** *x=0*])
by (*rule exI*[**where** *x=c*], *simp*)**}**
moreover
{assume *z*: *?g ≠ 0*
from *gcd-coprime-exists*[*OF z*]
obtain *a' b'* **where** *ab'*: *a = a' * ?g b = b' * ?g coprime a' b'* **by** *blast*
from *gcd-dvd2*[*of a b*] **have** *thb*: *?g dvd b* **.**
from *ab'*(*1*) **have** *a' dvd a* **unfolding** *dvd-def* **by** *blast*
with *dc* **have** *th0*: *a' dvd b*c* **using** *dvd-trans*[*of a' a b*c*] **by** *simp*
from *dc ab'*(*1*,*2*) **have** *a'*?g dvd (b'*?g) *c* **by** *auto*
hence *?g*a' dvd ?g * (b' * c)* **by** (*simp add*: *mult-assoc*)
with *z* **have** *th-1*: *a' dvd b'*c* **by** *simp*
from *coprime-divprod*[*OF th-1 ab'*(*3*)] **have** *thc*: *a' dvd c* **.**
from *ab'* **have** *a = ?g*a'* **by** *algebra*
with *thb thc* **have** *?thesis* **by** *blast* **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *nat-power-eq-0-iff*: (*m*::*nat*) ^ *n = 0* ⟷ *n ≠ 0 ∧ m = 0* **by** (*induct n*, *auto*)

lemma *divides-rev*: **assumes** *ab*: (*a*::*nat*) ^ *n dvd b* ^*n* **and** *n*:*n ≠ 0* **shows** *a dvd b*
proof−
let *?g = gcd a b*
from *n* **obtain** *m* **where** *m*: *n = Suc m* **by** (*cases n*, *simp-all*)
{assume *?g = 0* **with** *ab n* **have** *?thesis* **by** (*simp add*: *gcd-zero*)**}**
moreover
{assume *z*: *?g ≠ 0*
hence *zn*: *?g* ^ *n ≠ 0* **using** *n* **by** (*simp add*: *neq0-conv*)
from *gcd-coprime-exists*[*OF z*]
obtain *a' b'* **where** *ab'*: *a = a' * ?g b = b' * ?g coprime a' b'* **by** *blast*
from *ab* **have** (*a' * ?g*) ^ *n dvd* (*b' * ?g*) ^*n* **by** (*simp add*: *ab'*(*1*,*2*)[*symmetric*])
hence *?g^n*a'^n dvd ?g^n *b'^n* **by** (*simp only*: *power-mult-distrib mult-commute*)
with *zn z n* **have** *th0*:*a'^n dvd b'^n* **by** (*auto simp add*: *nat-power-eq-0-iff*)
have *a' dvd a'^n* **by** (*simp add*: *m*)
with *th0* **have** *a' dvd b'^n* **using** *dvd-trans*[*of a' a'^n b'^n*] **by** *simp*
hence *th1*: *a' dvd b'^m * b'* **by** (*simp add*: *m mult-commute*)
from *coprime-divprod*[*OF th1 coprime-exp*[*OF ab'*(*3*), *of m*]]
have *a' dvd b'* **.**
hence *a'*?g dvd b'*?g* **by** *simp*
with *ab'*(*1*,*2*) **have** *?thesis* **by** *simp* **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *divides-mul*: **assumes** *mr*: *m dvd r* **and** *nr*: *n dvd r* **and** *mn*:*coprime m*

n
shows *m * n dvd r*
proof−
from *mr nr* **obtain** *m' n'* **where** *m'*: *r = m*m'* **and** *n'*: *r = n*n'*
unfolding *dvd-def* **by** *blast*
from *mr n'* **have** *m dvd n'*n* **by** (*simp add*: *mult-commute*)
hence *m dvd n'* **using** *relprime-dvd-mult-iff*[*OF mn*[*unfolded coprime-def*]] **by** *simp*
then obtain *k* **where** *k*: *n' = m*k* **unfolding** *dvd-def* **by** *blast*
from *n' k* **show** *?thesis* **unfolding** *dvd-def* **by** *auto*
qed

A binary form of the Chinese Remainder Theorem.

lemma *chinese-remainder*: **assumes** *ab*: *coprime a b* **and** *a*:*a ≠ 0* **and** *b*:*b ≠ 0*
shows ∃ *x q1 q2*. *x = u + q1 * a ∧ x = v + q2 * b*
proof−
from *bezout-add-strong*[*OF a*, *of b*] *bezout-add-strong*[*OF b*, *of a*]
obtain *d1 x1 y1 d2 x2 y2* **where** *dxy1*: *d1 dvd a d1 dvd b a * x1 = b * y1 + d1*
and *dxy2*: *d2 dvd b d2 dvd a b * x2 = a * y2 + d2* **by** *blast*
from *gcd-unique*[*of 1 a b*, *simplified ab*[*unfolded coprime-def*], *simplified*]
dxy1(*1*,*2*) *dxy2*(*1*,*2*) **have** *d12*: *d1 = 1 d2 =1* **by** *auto*
let *?x = v * a * x1 + u * b * x2*
let *?q1 = v * x1 + u * y2*
let *?q2 = v * y1 + u * x2*
from *dxy2*(*3*)[*simplified d12*] *dxy1*(*3*)[*simplified d12*]
have *?x = u + ?q1 * a ?x = v + ?q2 * b* **by** *algebra*+
thus *?thesis* **by** *blast*
qed

Primality

A few useful theorems about primes

lemma *prime-0*[*simp*]: ~*prime 0* **by** (*simp add*: *prime-def*)
lemma *prime-1*[*simp*]: ~ *prime 1* **by** (*simp add*: *prime-def*)
lemma *prime-Suc0*[*simp*]: ~ *prime* (*Suc 0*) **by** (*simp add*: *prime-def*)

lemma *prime-ge-2*: *prime p ==> p ≥ 2* **by** (*simp add*: *prime-def*)
lemma *prime-factor*: **assumes** *n*: *n ≠ 1* **shows** ∃ *p*. *prime p ∧ p dvd n*
using *n*
proof(*induct n rule*: *nat-less-induct*)
fix *n*
assume *H*: ∀ *m*<*n*. *m ≠ 1* ⟶ (∃ *p*. *prime p ∧ p dvd m*) *n ≠ 1*
let *?ths* = ∃ *p*. *prime p ∧ p dvd n*
{assume *n=0* **hence** *?ths* **using** *two-is-prime* **by** *auto***}**
moreover
{assume *nz*: *n≠0*
{assume *prime n* **hence** *?ths* **by** − (*rule exI*[**where** *x=n*], *simp*)**}**
moreover
{assume *n*: ¬ *prime n*
with *nz H*(*2*)

obtain *k* **where** *k*:*k dvd n k* ≠ *1 k* ≠ *n* **by** (*auto simp add*: *prime-def*)
from *dvd-imp-le*[*OF k*(*1*)] *nz k*(*3*) **have** *kn*: *k* < *n* **by** *simp*
from *H*(*1*)[*rule-format*, *OF kn k*(*2*)] **obtain** *p* **where** *p*: *prime p p dvd k* **by** *blast*
from *dvd-trans*[*OF p*(*2*) *k*(*1*)] *p*(*1*) **have** *?ths* **by** *blast*}
ultimately have *?ths* **by** *blast*}
ultimately show *?ths* **by** *blast*
qed

lemma *prime-factor-lt*: **assumes** *p*: *prime p* **and** *n*: *n* ≠ *0* **and** *npm*:*n* = *p* ∗ *m*
shows *m* < *n*
proof−
{**assume** *m*=*0* **with** *n* **have** *?thesis* **by** *simp*}
moreover
{**assume** *m*: *m* ≠ *0*
from *npm* **have** *mn*: *m dvd n* **unfolding** *dvd-def* **by** *auto*
from *npm m* **have** *n* ≠ *m* **using** *p* **by** *auto*
with *dvd-imp-le*[*OF mn*] *n* **have** *?thesis* **by** *simp*}
ultimately show *?thesis* **by** *blast*
qed

lemma *euclid-bound*: ∃ *p*. *prime p* ∧ *n* < *p* ∧ *p* <= *Suc* (*fact n*)
proof−
have *f1*: *fact n* + *1* ≠ *1* **using** *fact-le*[*of n*] **by** *arith*
from *prime-factor*[*OF f1*] **obtain** *p* **where** *p*: *prime p p dvd fact n* + *1* **by** *blast*
from *dvd-imp-le*[*OF p*(*2*)] **have** *pfn*: *p* ≤ *fact n* + *1* **by** *simp*
{**assume** *np*: *p* ≤ *n*
from *p*(*1*) **have** *p1*: *p* ≥ *1* **by** (*cases p*, *simp-all*)
from *divides-fact*[*OF p1 np*] **have** *pfn′*: *p dvd fact n* **.**
from *divides-add-revr*[*OF pfn′ p*(*2*)] *p*(*1*) **have** *False* **by** *simp*}
hence *n* < *p* **by** *arith*
with *p*(*1*) *pfn* **show** *?thesis* **by** *auto*
qed

lemma *euclid*: ∃ *p*. *prime p* ∧ *p* > *n* **using** *euclid-bound* **by** *auto*
lemma *primes-infinite*: ¬ (*finite* {*p*. *prime p*})
proof (*auto simp add*: *finite-conv-nat-seg-image*)
fix *n f*
assume *H*: *Collect prime* = *f* ‘ {*i*. *i* < (*n*::*nat*)}
let *?P* = *Collect prime*
let *?m* = *Max ?P*
have *P0*: *?P* ≠ {} **using** *two-is-prime* **by** *auto*
from *H* **have** *fP*: *finite ?P* **using** *finite-conv-nat-seg-image* **by** *blast*
from *Max-in* [*OF fP P0*] **have** *?m* ∈ *?P* **.**
from *Max-ge* [*OF fP*] **have** *contr*: ∀ *p*. *prime p* ⟶ *p* ≤ *?m* **by** *blast*
from *euclid* [*of ?m*] **obtain** *q* **where** *q*: *prime q q* > *?m* **by** *blast*
with *contr* **show** *False* **by** *auto*
qed

lemma *coprime-prime*: **assumes** *ab*: *coprime a b*
shows ~(*prime p* ∧ *p dvd a* ∧ *p dvd b*)
proof
assume *prime p* ∧ *p dvd a* ∧ *p dvd b*
thus *False* **using** *ab gcd-greatest*[*of p a b*] **by** (*simp add*: *coprime-def*)
qed
lemma *coprime-prime-eq*: *coprime a b* ⟷ (∀ *p*. ~(*prime p* ∧ *p dvd a* ∧ *p dvd b*))

(**is** *?lhs* = *?rhs*)
proof−
{assume *?lhs* **with** *coprime-prime* **have** *?rhs* **by** *blast***}**
moreover
{assume *r*: *?rhs* **and** *c*: ¬ *?lhs*
then obtain *g* **where** *g*: *g*≠*1 g dvd a g dvd b* **unfolding** *coprime-def* **by** *blast*
from *prime-factor*[*OF g*(*1*)] **obtain** *p* **where** *p*: *prime p p dvd g* **by** *blast*
from *dvd-trans* [*OF p*(*2*) *g*(*2*)] *dvd-trans* [*OF p*(*2*) *g*(*3*)]
have *p dvd a p dvd b* **. with** *p*(*1*) *r* **have** *False* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *prime-coprime*: **assumes** *p*: *prime p*
shows *n* = *1* ∨ *p dvd n* ∨ *coprime p n*
using *p prime-imp-relprime*[*of p n*] **by** (*auto simp add*: *coprime-def*)

lemma *prime-coprime-strong*: *prime p* ⟹ *p dvd n* ∨ *coprime p n*
using *prime-coprime*[*of p n*] **by** *auto*

declare *coprime-0*[*simp*]

lemma *coprime-0'*[*simp*]: *coprime 0 d* ⟷ *d* = *1* **by** (*simp add*: *coprime-commute*[*of 0 d*])
lemma *coprime-bezout-strong*: **assumes** *ab*: *coprime a b* **and** *b*: *b* ≠ *1*
shows ∃ *x y*. *a* ∗ *x* = *b* ∗ *y* + *1*
proof−
from *ab b* **have** *az*: *a* ≠ *0* **by** − (*rule ccontr*, *auto*)
from *bezout-gcd-strong*[*OF az*, *of b*] *ab*[*unfolded coprime-def*]
show *?thesis* **by** *auto*
qed

lemma *bezout-prime*: **assumes** *p*: *prime p* **and** *pa*: ¬ *p dvd a*
shows ∃ *x y*. *a*∗*x* = *p*∗*y* + *1*
proof−
from *p* **have** *p1*: *p* ≠ *1* **using** *prime-1* **by** *blast*
from *prime-coprime*[*OF p*, *of a*] *p1 pa* **have** *ap*: *coprime a p*
by (*auto simp add*: *coprime-commute*)
from *coprime-bezout-strong*[*OF ap p1*] **show** *?thesis* **.**
qed
lemma *prime-divprod*: **assumes** *p*: *prime p* **and** *pab*: *p dvd a*∗*b*
shows *p dvd a* ∨ *p dvd b*

proof−
{assume *a=1* **hence** *?thesis* **using** *pab* **by** *simp* **}**
moreover
{assume *p dvd a* **hence** *?thesis* **by** *blast***}**
moreover
{assume *pa*: *coprime p a* **from** *coprime-divprod*[*OF pab pa*] **have** *?thesis* **.. }**
ultimately show *?thesis* **using** *prime-coprime*[*OF p*, *of a*] **by** *blast*
qed

lemma *prime-divprod-eq*: **assumes** *p*: *prime p*
shows *p dvd a*∗*b* ⟷ *p dvd a* ∨ *p dvd b*
using *p prime-divprod dvd-mult dvd-mult2* **by** *auto*

lemma *prime-divexp*: **assumes** *p*:*prime p* **and** *px*: *p dvd x*^*n*
shows *p dvd x*
using *px*
proof(*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
hence *th*: *p dvd x*∗*x*^*n* **by** *simp*
{assume *H*: *p dvd x*^*n*
from *Suc.hyps*[*OF H*] **have** *?case* **.}**
with *prime-divprod*[*OF p th*] **show** *?case* **by** *blast*
qed

lemma *prime-divexp-n*: *prime p* ⟹ *p dvd x*^*n* ⟹ *p*^*n dvd x*^*n*
using *prime-divexp*[*of p x n*] *divides-exp*[*of p x n*] **by** *blast*

lemma *coprime-prime-dvd-ex*: **assumes** *xy*: ¬*coprime x y*
shows ∃*p*. *prime p* ∧ *p dvd x* ∧ *p dvd y*
proof−
from *xy*[*unfolded coprime-def*] **obtain** *g* **where** *g*: *g* ≠ *1 g dvd x g dvd y*
by *blast*
from *prime-factor*[*OF g(1)*] **obtain** *p* **where** *p*: *prime p p dvd g* **by** *blast*
from *g*(*2*,*3*) *dvd-trans*[*OF p*(*2*)] *p*(*1*) **show** *?thesis* **by** *auto*
qed
lemma *coprime-sos*: **assumes** *xy*: *coprime x y*
shows *coprime* (*x* ∗ *y*) (*x*^*2* + *y*^*2*)
proof−
{assume *c*: ¬ *coprime* (*x* ∗ *y*) (*x*^*2* + *y*^*2*)
from *coprime-prime-dvd-ex*[*OF c*] **obtain** *p*
where *p*: *prime p p dvd x*∗*y p dvd x*^*2* + *y*^*2* **by** *blast*
{assume *px*: *p dvd x*
from *dvd-mult*[*OF px*, *of x*] *p*(*3*)
obtain *r s* **where** *x* ∗ *x* = *p* ∗ *r* **and** *x*^*2* + *y*^*2* = *p* ∗ *s*
by (*auto elim*!: *dvdE*)
then have *y*^*2* = *p* ∗ (*s* − *r*)
by (*auto simp add*: *power2-eq-square diff-mult-distrib2*)

```
      then have p dvd y^2 ..
    with prime-divexp[OF p(1), of y 2] have py: p dvd y .
    from p(1) px py xy[unfolded coprime, rule-format, of p] prime-1
    have False by simp }
  moreover
  {assume py: p dvd y
    from dvd-mult[OF py, of y] p(3)
    obtain r s where y * y = p * r and x^2 + y^2 = p * s
      by (auto elim!: dvdE)
    then have x^2 = p * (s - r)
      by (auto simp add: power2-eq-square diff-mult-distrib2)
    then have p dvd x^2 ..
    with prime-divexp[OF p(1), of x 2] have px: p dvd x .
    from p(1) px py xy[unfolded coprime, rule-format, of p] prime-1
    have False by simp }
  ultimately have False using prime-divprod[OF p(1,2)] by blast}
 thus ?thesis by blast
qed

lemma distinct-prime-coprime: prime p ⟹ prime q ⟹ p ≠ q ⟹ coprime p q
 unfolding prime-def coprime-prime-eq by blast

lemma prime-coprime-lt: assumes p: prime p and x: 0 < x and xp: x < p
 shows coprime x p
proof−
 {assume c: ¬ coprime x p
   then obtain g where g: g ≠ 1 g dvd x g dvd p unfolding coprime-def by
blast
 from dvd-imp-le[OF g(2)] x xp have gp: g < p by arith
 from g(2) x have g ≠ 0 by − (rule ccontr, simp)
 with g gp p[unfolded prime-def] have False by blast}
thus ?thesis by blast
qed

lemma even-dvd[simp]: even (n::nat) ⟷ 2 dvd n by presburger
lemma prime-odd: prime p ⟹ p = 2 ∨ odd p unfolding prime-def by auto
```

One property of coprimality is easier to prove via prime factors.

```
lemma prime-divprod-pow:
 assumes p: prime p and ab: coprime a b and pab: p^n dvd a * b
 shows p^n dvd a ∨ p^n dvd b
proof−
 {assume n = 0 ∨ a = 1 ∨ b = 1 with pab have ?thesis
    apply (cases n=0, simp-all)
    apply (cases a=1, simp-all) done}
 moreover
 {assume n: n ≠ 0 and a: a≠1 and b: b≠1
  then obtain m where m: n = Suc m by (cases n, auto)
  from divides-exp2[OF n pab] have pab': p dvd a*b .
```

```
    from prime-divprod[OF p pab'] 
    have p dvd a ∨ p dvd b .
    moreover
    {assume pa: p dvd a
     have pnba: p^n dvd b*a using pab by (simp add: mult-commute)
     from coprime-prime[OF ab, of p] p pa have ¬ p dvd b by blast
     with prime-coprime[OF p, of b] b
     have cpb: coprime b p using coprime-commute by blast
     from coprime-exp[OF cpb] have pnb: coprime (p^n) b
       by (simp add: coprime-commute)
     from coprime-divprod[OF pnba pnb] have ?thesis by blast }
    moreover
    {assume pb: p dvd b
     have pnba: p^n dvd b*a using pab by (simp add: mult-commute)
     from coprime-prime[OF ab, of p] p pb have ¬ p dvd a by blast
     with prime-coprime[OF p, of a] a
     have cpb: coprime a p using coprime-commute by blast
     from coprime-exp[OF cpb] have pnb: coprime (p^n) a
       by (simp add: coprime-commute)
     from coprime-divprod[OF pab pnb] have ?thesis by blast }
    ultimately have ?thesis by blast}
  ultimately show ?thesis by blast
qed

lemma nat-mult-eq-one: (n::nat) * m = 1 ⟷ n = 1 ∧ m = 1 (is ?lhs ⟷
?rhs)
proof
  assume H: ?lhs
  hence n dvd 1 m dvd 1 unfolding dvd-def by (auto simp add: mult-commute)
  thus ?rhs by auto
next
  assume ?rhs then show ?lhs by auto
qed

lemma power-Suc0[simp]: Suc 0 ^ n = Suc 0
  unfolding One-nat-def[symmetric] power-one ..
lemma coprime-pow: assumes ab: coprime a b and abcn: a * b = c ^n
  shows ∃ r s. a = r^n ∧ b = s ^n
  using ab abcn
proof(induct c arbitrary: a b rule: nat-less-induct)
  fix c a b
  assume H: ∀ m<c. ∀ a b. coprime a b ⟶ a * b = m ^ n ⟶ (∃ r s. a = r ^ n
∧ b = s ^ n) coprime a b a * b = c ^ n
  let ?ths = ∃ r s. a = r^n ∧ b = s ^n
  {assume n: n = 0
    with H(3) power-one have a*b = 1 by simp
    hence a = 1 ∧ b = 1 by simp
    hence ?ths
      apply −
```

```
    apply (rule exI[where x=1])
    apply (rule exI[where x=1])
    using power-one[of n]
    by simp}
 moreover
 {assume n: n ≠ 0 then obtain m where m: n = Suc m by (cases n, auto)
  {assume c: c = 0
   with H(3) m H(2) have ?ths apply simp
    apply (cases a=0, simp-all)
    apply (rule exI[where x=0], simp)
    apply (rule exI[where x=0], simp)
    done}
  moreover
  {assume c=1 with H(3) power-one have a*b = 1 by simp
    hence a = 1 ∧ b = 1 by simp
    hence ?ths
   apply −
   apply (rule exI[where x=1])
   apply (rule exI[where x=1])
   using power-one[of n]
   by simp}
 moreover
 {assume c: c≠1 c ≠ 0
  from prime-factor[OF c(1)] obtain p where p: prime p p dvd c by blast
   from prime-divprod-pow[OF p(1) H(2), unfolded H(3), OF divides-exp[OF
p(2), of n]]
   have pnab: p ^ n dvd a ∨ p^n dvd b .
   from p(2) obtain l where l: c = p*l unfolding dvd-def by blast
   have pn0: p^n ≠ 0 using n prime-ge-2 [OF p(1)] by (simp add: neq0-conv)
   {assume pa: p^n dvd a
    then obtain k where k: a = p^n * k unfolding dvd-def by blast
    from l have l dvd c by auto
    with dvd-imp-le[of l c] c have l ≤ c by auto
   moreover {assume l = c with l c have p = 1 by simp with p have False
by simp}
    ultimately have lc: l < c by arith
    from coprime-lmul2 [OF H(2)[unfolded k coprime-commute[of p^n*k b]]]
    have kb: coprime k b by (simp add: coprime-commute)
    from H(3) l k pn0 have kbln: k * b = l ^ n
     by (auto simp add: power-mult-distrib)
    from H(1)[rule-format, OF lc kb kbln]
    obtain r s where rs: k = r ^n b = s^n by blast
    from k rs(1) have a = (p*r) ^n by (simp add: power-mult-distrib)
    with rs(2) have ?ths by blast }
  moreover
  {assume pb: p^n dvd b
   then obtain k where k: b = p^n * k unfolding dvd-def by blast
   from l have l dvd c by auto
   with dvd-imp-le[of l c] c have l ≤ c by auto
```

moreover {assume $l = c$ **with** $l\ c$ **have** $p = 1$ **by** *simp* **with** p **have** *False*
by *simp***}**
ultimately have *lc*: $l < c$ **by** *arith*
from *coprime-lmul2* [*OF H*(*2*)[*unfolded k coprime-commute*[*of p ^ n*∗*k a*]]]
have *kb*: *coprime k a* **by** (*simp add*: *coprime-commute*)
from *H*(*3*) *l k pn0 n* **have** *kbln*: $k * a = l\ \hat{}\ n$
by (*simp add*: *power-mult-distrib mult-commute*)
from *H*(*1*)[*rule-format*, *OF lc kb kbln*]
obtain *r s* **where** *rs*: $k = r\ \hat{}n\ a = s\hat{}n$ **by** *blast*
from *k rs*(*1*) **have** $b = (p*r)\hat{}n$ **by** (*simp add*: *power-mult-distrib*)
with *rs*(*2*) **have** *?ths* **by** *blast* **}**
ultimately have *?ths* **using** *pnab* **by** *blast***}**
ultimately have *?ths* **by** *blast***}**
ultimately show *?ths* **by** *blast*
qed

More useful lemmas.

lemma *prime-product*:
assumes *prime* $(p * q)$
shows $p = 1 \vee q = 1$
proof −
from *assms* **have**
$1 < p * q$ **and** *P*: $\bigwedge m.\ m\ dvd\ p * q \Longrightarrow m = 1 \vee m = p * q$
unfolding *prime-def* **by** *auto*
from ‹$1 < p * q$› **have** $p \neq 0$ **by** (*cases p*) *auto*
then have *Q*: $p = p * q \longleftrightarrow q = 1$ **by** *auto*
have $p\ dvd\ p * q$ **by** *simp*
then have $p = 1 \vee p = p * q$ **by** (*rule P*)
then show *?thesis* **by** (*simp add*: *Q*)
qed

lemma *prime-exp*: *prime* $(p\hat{}n) \longleftrightarrow$ *prime* $p \wedge n = 1$
proof(*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
{assume $p = 0$ **hence** *?case* **by** *simp***}**
moreover
{assume $p{=}1$ **hence** *?case* **by** *simp***}**
moreover
{assume *p*: $p \neq 0\ p{\neq}1$
{assume *pp*: *prime* $(p\hat{}Suc\ n)$
hence $p = 1 \vee p\hat{}n = 1$ **using** *prime-product*[*of p p ^ n*] **by** *simp*
with *p* **have** *n*: $n = 0$
by (*simp only*: *exp-eq-1*) *simp*
with *pp* **have** *prime p* ∧ *Suc n* = *1* **by** *simp***}**
moreover
{assume *n*: *prime p* ∧ *Suc n* = *1* **hence** *prime* $(p\hat{}Suc\ n)$ **by** *simp***}**
ultimately have *?case* **by** *blast***}**

```
  ultimately show ?case by blast
qed

lemma prime-power-mult:
  assumes p: prime p and xy: x * y = p ^ k
  shows ∃ i j. x = p ^i ∧ y = p ^ j
  using xy
proof(induct k arbitrary: x y)
  case 0 thus ?case apply simp by (rule exI[where x=0], simp)
next
  case (Suc k x y)
  from Suc.prems have pxy: p dvd x*y by auto
  from prime-divprod[OF p pxy] have pxyc: p dvd x ∨ p dvd y .
  from p have p0: p ≠ 0 by − (rule ccontr, simp)
  {assume px: p dvd x
    then obtain d where d: x = p*d unfolding dvd-def by blast
    from Suc.prems d  have p*d*y = p^Suc k by simp
    hence th: d*y = p^k using p0 by simp
    from Suc.hyps[OF th] obtain i j where ij: d = p^i y = p^j by blast
    with d have x = p^Suc i by simp
    with ij(2) have ?case by blast}
  moreover
  {assume px: p dvd y
    then obtain d where d: y = p*d unfolding dvd-def by blast
    from Suc.prems d  have p*d*x = p^Suc k by (simp add: mult-commute)
    hence th: d*x = p^k using p0 by simp
    from Suc.hyps[OF th] obtain i j where ij: d = p^i x = p^j by blast
    with d have y = p^Suc i by simp
    with ij(2) have ?case by blast}
  ultimately show ?case  using pxyc by blast
qed

lemma prime-power-exp: assumes p: prime p and n:n ≠ 0
  and xn: x^n = p^k shows ∃ i. x = p^i
  using n xn
proof(induct n arbitrary: k)
  case 0 thus ?case by simp
next
  case (Suc n k) hence th: x*x^n = p^k by simp
  {assume n = 0 with prems have ?case apply simp
     by (rule exI[where x=k],simp)}
  moreover
  {assume n: n ≠ 0
    from prime-power-mult[OF p th]
    obtain i j where ij: x = p^i x^n = p^jby blast
    from Suc.hyps[OF n ij(2)] have ?case .}
  ultimately show ?case by blast
qed
```

lemma *divides-primepow*: **assumes** *p*: *prime p*
shows *d dvd p^k* $\longleftrightarrow$ $(\exists\ i.\ i \leq k \wedge d = p\ \hat{}\ i)$
proof
assume *H*: *d dvd p^k* **then obtain** *e* **where** *e*: $d*e = p\hat{}k$
unfolding *dvd-def* **apply** (*auto simp add*: *mult-commute*) **by** *blast*
from *prime-power-mult*[*OF p e*] **obtain** *i j* **where** *ij*: $d = p\hat{}i\ e{=}p\hat{}j$ **by** *blast*
from *prime-ge-2*[*OF p*] **have** *p1*: $p > 1$ **by** *arith*
from *e ij* **have** $p\hat{}(i + j) = p\hat{}k$ **by** (*simp add*: *power-add*)
hence $i + j = k$ **using** *power-inject-exp*[*of p i+j k*, *OF p1*] **by** *simp*
hence $i \leq k$ **by** *arith*
with *ij*(*1*) **show** $\exists i{\leq}k.\ d = p\ \hat{}\ i$ **by** *blast*
next
{fix *i* **assume** *H*: $i \leq k\ d = p\hat{}i$
hence $\exists j.\ k = i + j$ **by** *arith*
then obtain *j* **where** *j*: $k = i + j$ **by** *blast*
hence $p\hat{}k = p\hat{}j*d$ **using** *H*(*2*) **by** (*simp add*: *power-add*)
hence *d dvd p^k* **unfolding** *dvd-def* **by** *auto***}**
thus $\exists i{\leq}k.\ d = p\ \hat{}\ i \Longrightarrow$ *d dvd p ^ k* **by** *blast*
qed

lemma *coprime-divisors*: *d dvd a* $\Longrightarrow$ *e dvd b* $\Longrightarrow$ *coprime a b* $\Longrightarrow$ *coprime d e*
by (*auto simp add*: *dvd-def coprime*)

declare *power-Suc0*[*simp del*]
declare *even-dvd*[*simp del*]

end

52 Pocklington: Pocklington's Theorem for Primes

theory *Pocklington*
imports *Main Primes*
begin

definition *modeq*:: *nat => nat => nat => bool* ((1[- = -] '(*mod* -')))
where $[a = b]\ (mod\ p) == ((a\ mod\ p) = (b\ mod\ p))$

definition *modneq*:: *nat => nat => nat => bool* ((1[- $\neq$ -] '(*mod* -')))
where $[a \neq b]\ (mod\ p) == ((a\ mod\ p) \neq (b\ mod\ p))$

lemma *modeq-trans*:
$[\![\ [a = b]\ (mod\ p);\ [b = c]\ (mod\ p)\ ]\!] \Longrightarrow [a = c]\ (mod\ p)$
by (*simp add*:*modeq-def*)

lemma *nat-mod-lemma*: **assumes** *xyn*: $[x = y]\ (mod\ n)$ **and** *xy*:$y \leq x$
shows $\exists q.\ x = y + n * q$
using *xyn xy* **unfolding** *modeq-def* **using** *nat-mod-eq-lemma* **by** *blast*

lemma *nat-mod*[*algebra*]: [*x* = *y*] (*mod n*) ⟷ (∃ *q1 q2*. *x* + *n* ∗ *q1* = *y* + *n* ∗ *q2*)
unfolding *modeq-def nat-mod-eq-iff* **..**

lemma *prime*: *prime p* ⟷ *p* ≠ *0* ∧ *p*≠*1* ∧ (∀ *m*. *0* < *m* ∧ *m* < *p* ⟶ *coprime p m*)
(**is** *?lhs* ⟷ *?rhs*)
proof−
{assume *p=0* ∨ *p=1* **hence** *?thesis* **using** *prime-0 prime-1* **by** (*cases p=0*, *simp-all*)**}**
moreover
{assume *p0*: *p*≠*0 p*≠*1*
{assume *H*: *?lhs*
{fix *m* **assume** *m*: *m* > *0 m* < *p*
{assume *m=1* **hence** *coprime p m* **by** *simp***}**
moreover
{assume *p dvd m* **hence** *p* ≤ *m* **using** *dvd-imp-le m* **by** *blast* **with** *m*(*2*)
have *coprime p m* **by** *simp***}**
ultimately have *coprime p m* **using** *prime-coprime*[*OF H*, *of m*] **by** *blast***}**
hence *?rhs* **using** *p0* **by** *auto***}**
moreover
{ **assume** *H*: ∀ *m*. *0* < *m* ∧ *m* < *p* ⟶ *coprime p m*
from *prime-factor*[*OF p0*(*2*)] **obtain** *q* **where** *q*: *prime q q dvd p* **by** *blast*
from *prime-ge-2*[*OF q*(*1*)] **have** *q0*: *q* > *0* **by** *arith*
from *dvd-imp-le*[*OF q*(*2*)] *p0* **have** *qp*: *q* ≤ *p* **by** *arith*
{assume *q* = *p* **hence** *?lhs* **using** *q*(*1*) **by** *blast***}**
moreover
{assume *q*≠*p* **with** *qp* **have** *qplt*: *q* < *p* **by** *arith*
from *H*[*rule-format*, *of q*] *qplt q0* **have** *coprime p q* **by** *arith*
with *coprime-prime*[*of p q q*] *q* **have** *False* **by** *simp* **hence** *?lhs* **by** *blast***}**
ultimately have *?lhs* **by** *blast***}**
ultimately have *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** (*casesp=0* ∨ *p=1*, *auto*)
qed

lemma *finite-number-segment*: *card* { *m*. *0* < *m* ∧ *m* < *n* } = *n* − *1*
proof−
have { *m*. *0* < *m* ∧ *m* < *n* } = {*1*..<*n*} **by** *auto*
thus *?thesis* **by** *simp*
qed

lemma *coprime-mod*: **assumes** *n*: *n* ≠ *0* **shows** *coprime* (*a mod n*) *n* ⟷ *coprime a n*
using *n dvd-mod-iff*[*of* - *n a*] **by** (*auto simp add*: *coprime*)

lemma *cong-mod-01*[*simp*,*presburger*]:
[*x* = *y*] (*mod* 0) ⟷ *x* = *y* [*x* = *y*] (*mod* 1) [*x* = 0] (*mod n*) ⟷ *n dvd x*
by (*simp-all add*: *modeq-def*, *presburger*)

lemma *cong-sub-cases*:
[*x* = *y*] (*mod n*) ⟷ (*if x* <= *y then* [*y* − *x* = 0] (*mod n*) *else* [*x* − *y* = 0] (*mod n*))
apply (*auto simp add*: *nat-mod*)
apply (*rule-tac x*=*q2* **in** *exI*)
apply (*rule-tac x*=*q1* **in** *exI*, *simp*)
apply (*rule-tac x*=*q2* **in** *exI*)
apply (*rule-tac x*=*q1* **in** *exI*, *simp*)
apply (*rule-tac x*=*q1* **in** *exI*)
apply (*rule-tac x*=*q2* **in** *exI*, *simp*)
apply (*rule-tac x*=*q1* **in** *exI*)
apply (*rule-tac x*=*q2* **in** *exI*, *simp*)
done

lemma *cong-mult-lcancel*: **assumes** *an*: *coprime a n* **and** *axy*:[*a* ∗ *x* = *a* ∗ *y*] (*mod n*)
shows [*x* = *y*] (*mod n*)
proof−
{assume *a* = 0 **with** *an axy coprime-0'*[*of n*] **have** *?thesis* **by** (*simp add*: *modeq-def*) **}**
moreover
{assume *az*: *a*≠*0*
{assume *xy*: *x* ≤ *y* **hence** *axy'*: *a*∗*x* ≤ *a*∗*y* **by** *simp*
with *axy cong-sub-cases*[*of a*∗*x a*∗*y n*] **have** [*a*∗(*y* − *x*) = 0] (*mod n*)
by (*simp only*: *if-True diff-mult-distrib2*)
hence *th*: *n dvd a*∗(*y* −*x*) **by** *simp*
from *coprime-divprod*[*OF th*] *an* **have** *n dvd y* − *x*
by (*simp add*: *coprime-commute*)
hence *?thesis* **using** *xy cong-sub-cases*[*of x y n*] **by** *simp***}**
moreover
{assume *H*: ¬*x* ≤ *y* **hence** *xy*: *y* ≤ *x* **by** *arith*
from *H az* **have** *axy'*: ¬ *a*∗*x* ≤ *a*∗*y* **by** *auto*
with *axy H cong-sub-cases*[*of a*∗*x a*∗*y n*] **have** [*a*∗(*x* − *y*) = 0] (*mod n*)
by (*simp only*: *if-False diff-mult-distrib2*)
hence *th*: *n dvd a*∗(*x* − *y*) **by** *simp*
from *coprime-divprod*[*OF th*] *an* **have** *n dvd x* − *y*
by (*simp add*: *coprime-commute*)
hence *?thesis* **using** *xy cong-sub-cases*[*of x y n*] **by** *simp***}**
ultimately have *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *cong-mult-rcancel*: **assumes** *an*: *coprime a n* **and** *axy*:[*x*∗*a* = *y*∗*a*] (*mod n*)

shows $[x = y]\ (mod\ n)$
using *cong-mult-lcancel*[*OF an axy*[*unfolded mult-commute*[*of -a*]]] **.**

lemma *cong-refl*: $[x = x]\ (mod\ n)$ **by** (*simp add*: *modeq-def*)

lemma *eq-imp-cong*: $a = b \Longrightarrow [a = b]\ (mod\ n)$ **by** (*simp add*: *cong-refl*)

lemma *cong-commute*: $[x = y]\ (mod\ n) \longleftrightarrow [y = x]\ (mod\ n)$
by (*auto simp add*: *modeq-def*)

lemma *cong-trans*[*trans*]: $[x = y]\ (mod\ n) \Longrightarrow [y = z]\ (mod\ n) \Longrightarrow [x = z]\ (mod\ n)$
by (*simp add*: *modeq-def*)

lemma *cong-add*: **assumes** *xx'*: $[x = x']\ (mod\ n)$ **and** *yy'*:$[y = y']\ (mod\ n)$
shows $[x + y = x' + y']\ (mod\ n)$
proof−
have $(x + y)\ mod\ n = (x\ mod\ n + y\ mod\ n)\ mod\ n$
by (*simp add*: *mod-add-left-eq*[*of x y n*] *mod-add-right-eq*[*of x mod n y n*])
also have $\ldots = (x'\ mod\ n + y'\ mod\ n)\ mod\ n$ **using** *xx' yy' modeq-def* **by** *simp*
also have $\ldots = (x' + y')\ mod\ n$
by (*simp add*: *mod-add-left-eq*[*of x' y' n*] *mod-add-right-eq*[*of x' mod n y' n*])
finally show *?thesis* **unfolding** *modeq-def* **.**
qed

lemma *cong-mult*: **assumes** *xx'*: $[x = x']\ (mod\ n)$ **and** *yy'*:$[y = y']\ (mod\ n)$
shows $[x * y = x' * y']\ (mod\ n)$
proof−
have $(x * y)\ mod\ n = (x\ mod\ n) * (y\ mod\ n)\ mod\ n$
by (*simp add*: *mod-mult-left-eq*[*of x y n*] *mod-mult-right-eq*[*of x mod n y n*])
also have $\ldots = (x'\ mod\ n) * (y'\ mod\ n)\ mod\ n$ **using** *xx'*[*unfolded modeq-def*] *yy'*[*unfolded modeq-def*] **by** *simp*
also have $\ldots = (x' * y')\ mod\ n$
by (*simp add*: *mod-mult-left-eq*[*of x' y' n*] *mod-mult-right-eq*[*of x' mod n y' n*])
finally show *?thesis* **unfolding** *modeq-def* **.**
qed

lemma *cong-exp*: $[x = y]\ (mod\ n) \Longrightarrow [x\hat{}k = y\hat{}k]\ (mod\ n)$
by (*induct k*, *auto simp add*: *cong-refl cong-mult*)
lemma *cong-sub*: **assumes** *xx'*: $[x = x']\ (mod\ n)$ **and** *yy'*: $[y = y']\ (mod\ n)$
and *yx*: $y <= x$ **and** *yx'*: $y' <= x'$
shows $[x - y = x' - y']\ (mod\ n)$
proof−
{ fix *x a x' a' y b y' b'*
have $(x::nat) + a = x' + a' \Longrightarrow y + b = y' + b' \Longrightarrow y <= x \Longrightarrow y' <= x' \Longrightarrow (x - y) + (a + b') = (x' - y') + (a' + b)$ **by** *arith***}**
note *th* = *this*
from *xx' yy'* **obtain** *q1 q2 q1' q2'* **where** *q12*: $x + n*q1 = x'+n*q2$
and *q12'*: $y + n*q1' = y'+n*q2'$ **unfolding** *nat-mod* **by** *blast+*

from *th*[*OF q12 q12' yx yx'*]
have $(x - y) + n*(q1 + q2') = (x' - y') + n*(q2 + q1')$
by (*simp add*: *right-distrib*)
thus *?thesis* **unfolding** *nat-mod* **by** *blast*
qed

lemma *cong-mult-lcancel-eq*: **assumes** *an*: *coprime a n*
shows $[a * x = a * y]$ (*mod n*) $\longleftrightarrow$ $[x = y]$ (*mod n*) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *H*: *?rhs* **from** *cong-mult*[*OF cong-refl*[*of a n*] *H*] **show** *?lhs* **.**
next
assume *H*: *?lhs* **hence** *H'*: $[x*a = y*a]$ (*mod n*) **by** (*simp add*: *mult-commute*)
from *cong-mult-rcancel*[*OF an H'*] **show** *?rhs* **.**
qed

lemma *cong-mult-rcancel-eq*: **assumes** *an*: *coprime a n*
shows $[x * a = y * a]$ (*mod n*) $\longleftrightarrow$ $[x = y]$ (*mod n*)
using *cong-mult-lcancel-eq*[*OF an*, *of x y*] **by** (*simp add*: *mult-commute*)

lemma *cong-add-lcancel-eq*: $[a + x = a + y]$ (*mod n*) $\longleftrightarrow$ $[x = y]$ (*mod n*)
by (*simp add*: *nat-mod*)

lemma *cong-add-rcancel-eq*: $[x + a = y + a]$ (*mod n*) $\longleftrightarrow$ $[x = y]$ (*mod n*)
by (*simp add*: *nat-mod*)

lemma *cong-add-rcancel*: $[x + a = y + a]$ (*mod n*) $\Longrightarrow$ $[x = y]$ (*mod n*)
by (*simp add*: *nat-mod*)

lemma *cong-add-lcancel*: $[a + x = a + y]$ (*mod n*) $\Longrightarrow$ $[x = y]$ (*mod n*)
by (*simp add*: *nat-mod*)

lemma *cong-add-lcancel-eq-0*: $[a + x = a]$ (*mod n*) $\longleftrightarrow$ $[x = 0]$ (*mod n*)
by (*simp add*: *nat-mod*)

lemma *cong-add-rcancel-eq-0*: $[x + a = a]$ (*mod n*) $\longleftrightarrow$ $[x = 0]$ (*mod n*)
by (*simp add*: *nat-mod*)

lemma *cong-imp-eq*: **assumes** *xn*: $x < n$ **and** *yn*: $y < n$ **and** *xy*: $[x = y]$ (*mod n*)
shows $x = y$
using *xy*[*unfolded modeq-def mod-less*[*OF xn*] *mod-less*[*OF yn*]] **.**

lemma *cong-divides-modulus*: $[x = y]$ (*mod m*) $\Longrightarrow$ *n dvd m* ==> $[x = y]$ (*mod n*)
apply (*auto simp add*: *nat-mod dvd-def*)
apply (*rule-tac x=k*q1* **in** *exI*)
apply (*rule-tac x=k*q2* **in** *exI*)
by *simp*

lemma *cong-0-divides*: $[x = 0]\ (mod\ n) \longleftrightarrow n\ dvd\ x$ **by** *simp*

lemma *cong-1-divides*:$[x = 1]\ (mod\ n) ==> n\ dvd\ x - 1$
apply (*cases* $x \leq 1$, *simp-all*)
using *cong-sub-cases*[*of x 1 n*] **by** *auto*

lemma *cong-divides*: $[x = y]\ (mod\ n) \Longrightarrow n\ dvd\ x \longleftrightarrow n\ dvd\ y$
apply (*auto simp add*: *nat-mod dvd-def*)
apply (*rule-tac* $x{=}k + q1 - q2$ **in** *exI*, *simp add*: *add-mult-distrib2 diff-mult-distrib2*)
apply (*rule-tac* $x{=}k + q2 - q1$ **in** *exI*, *simp add*: *add-mult-distrib2 diff-mult-distrib2*)
done

lemma *cong-coprime*: **assumes** *xy*: $[x = y]\ (mod\ n)$
shows *coprime n x* $\longleftrightarrow$ *coprime n y*
proof−
{assume $n{=}0$ **hence** *?thesis* **using** *xy* **by** *simp***}**
moreover
{assume *nz*: $n \neq 0$
have *coprime n x* $\longleftrightarrow$ *coprime* $(x\ mod\ n)\ n$
by (*simp add*: *coprime-mod*[*OF nz*, *of x*] *coprime-commute*[*of n x*])
also have ... $\longleftrightarrow$ *coprime* $(y\ mod\ n)\ n$ **using** *xy*[*unfolded modeq-def*] **by** *simp*
also have ... $\longleftrightarrow$ *coprime y n* **by** (*simp add*: *coprime-mod*[*OF nz*, *of y*])
finally have *?thesis* **by** (*simp add*: *coprime-commute*) **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *cong-mod*: $\sim(n = 0) \Longrightarrow [a\ mod\ n = a]\ (mod\ n)$ **by** (*simp add*: *modeq-def*)

lemma *mod-mult-cong*: $\sim(a = 0) \Longrightarrow \sim(b = 0)$
$\Longrightarrow [x\ mod\ (a * b) = y]\ (mod\ a) \longleftrightarrow [x = y]\ (mod\ a)$
by (*simp add*: *modeq-def mod-mult2-eq mod-add-left-eq*)

lemma *cong-mod-mult*: $[x = y]\ (mod\ n) \Longrightarrow m\ dvd\ n \Longrightarrow [x = y]\ (mod\ m)$
apply (*auto simp add*: *nat-mod dvd-def*)
apply (*rule-tac* $x{=}k{*}q1$ **in** *exI*)
apply (*rule-tac* $x{=}k{*}q2$ **in** *exI*, *simp*)
done

lemma *cong-le*: $y <= x \Longrightarrow [x = y]\ (mod\ n) \longleftrightarrow (\exists\, q.\ x = q * n + y)$
using *nat-mod-lemma*[*of x y n*]
apply *auto*
apply (*simp add*: *nat-mod*)
apply (*rule-tac* $x{=}q$ **in** *exI*)
apply (*rule-tac* $x{=}q + q$ **in** *exI*)
by (*auto simp*: *algebra-simps*)

lemma *cong-to-1*: $[a = 1]\ (mod\ n) \longleftrightarrow a = 0 \wedge n = 1 \vee (\exists\, m.\ a = 1 + m * n)$

proof−
{assume $n = 0 \vee n = 1 \vee a = 0 \vee a = 1$ **hence** *?thesis*
apply (*cases n=0*, *simp-all add*: *cong-commute*)
apply (*cases n=1*, *simp-all add*: *cong-commute modeq-def*)
apply *arith*
by (*cases a=1*, *simp-all add*: *modeq-def cong-commute*)**}**
moreover
{assume *n*: $n \neq 0$ $n \neq 1$ **and** *a*:$a \neq 0$ $a \neq 1$ **hence** a': $a \geq 1$ **by** *simp*
hence *?thesis* **using** *cong-le*[*OF* a', *of n*] **by** *auto* **}**
ultimately show *?thesis* **by** *auto*
qed

lemma *cong-solve*: **assumes** *an*: *coprime a n* **shows** $\exists x.\ [a * x = b]\ (mod\ n)$
proof−
{assume *a=0* **hence** *?thesis* **using** *an* **by** (*simp add*: *modeq-def*)**}**
moreover
{assume *az*: $a \neq 0$
from *bezout-add-strong*[*OF az*, *of n*]
obtain *d x y* **where** *dxy*: *d dvd a d dvd n* $a*x = n*y + d$ **by** *blast*
from *an*[*unfolded coprime*, *rule-format*, *of d*] *dxy*(*1*,*2*) **have** *d1*: $d = 1$ **by** *blast*
hence $a*x*b = (n*y + 1)*b$ **using** *dxy*(*3*) **by** *simp*
hence $a*(x*b) = n*(y*b) + b$ **by** *algebra*
hence $a*(x*b)\ mod\ n = (n*(y*b) + b)\ mod\ n$ **by** *simp*
hence $a*(x*b)\ mod\ n = b\ mod\ n$ **by** (*simp add*: *mod-add-left-eq*)
hence $[a*(x*b) = b]\ (mod\ n)$ **unfolding** *modeq-def* **.**
hence *?thesis* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *cong-solve-unique*: **assumes** *an*: *coprime a n* **and** *nz*: $n \neq 0$
shows $\exists!x.\ x < n \wedge [a * x = b]\ (mod\ n)$
proof−
let *?P* = $\lambda x.\ x < n \wedge [a * x = b]\ (mod\ n)$
from *cong-solve*[*OF an*] **obtain** *x* **where** *x*: $[a*x = b]\ (mod\ n)$ **by** *blast*
let *?x* = *x mod n*
from *x* **have** *th*: $[a * ?x = b]\ (mod\ n)$
by (*simp add*: *modeq-def mod-mult-right-eq*[*of a x n*])
from *mod-less-divisor*[*of n x*] *nz th* **have** *Px*: *?P ?x* **by** *simp*
{fix *y* **assume** *Py*: $y < n$ $[a * y = b]\ (mod\ n)$
from *Py*(*2*) *th* **have** $[a * y = a*?x]\ (mod\ n)$ **by** (*simp add*: *modeq-def*)
hence $[y = ?x]\ (mod\ n)$ **by** (*simp add*: *cong-mult-lcancel-eq*[*OF an*])
with *mod-less*[*OF Py*(*1*)] *mod-less-divisor*[*of n x*] *nz*
have *y* = *?x* **by** (*simp add*: *modeq-def*)**}**
with *Px* **show** *?thesis* **by** *blast*
qed

lemma *cong-solve-unique-nontrivial*:
assumes *p*: *prime p* **and** *pa*: *coprime p a* **and** *x0*: $0 < x$ **and** *xp*: $x < p$
shows $\exists!y.\ 0 < y \wedge y < p \wedge [x * y = a]\ (mod\ p)$
proof−
from *p* **have** *p1*: $p > 1$ **using** *prime-ge-2*[*OF p*] **by** *arith*
hence *p01*: $p \neq 0\ p \neq 1$ **by** *arith*+
from *pa* **have** *ap*: *coprime a p* **by** (*simp add*: *coprime-commute*)
from *prime-coprime*[*OF p*, *of x*] *dvd-imp-le*[*of p x*] *x0 xp* **have** *px*:*coprime x p*
by (*auto simp add*: *coprime-commute*)
from *cong-solve-unique*[*OF px p01*(*1*)]
obtain *y* **where** *y*: $y < p\ [x * y = a]\ (mod\ p)\ \forall z.\ z < p \wedge [x * z = a]\ (mod\ p) \longrightarrow z = y$ **by** *blast*
{assume *y0*: $y = 0$
with *y*(*2*) **have** *th*: *p dvd a* **by** (*simp add*: *cong-commute*[*of 0 a p*])
with *p coprime-prime*[*OF pa*, *of p*] **have** *False* **by** *simp***}**
with *y* **show** *?thesis* **unfolding** *Ex1-def* **using** *neq0-conv* **by** *blast*
qed
lemma *cong-unique-inverse-prime*:
assumes *p*: *prime p* **and** *x0*: $0 < x$ **and** *xp*: $x < p$
shows $\exists!y.\ 0 < y \wedge y < p \wedge [x * y = 1]\ (mod\ p)$
using *cong-solve-unique-nontrivial*[*OF p coprime-1*[*of p*] *x0 xp*] **.**

lemma *cong-chinese*:
assumes *ab*: *coprime a b* **and** *xya*: $[x = y]\ (mod\ a)$
and *xyb*: $[x = y]\ (mod\ b)$
shows $[x = y]\ (mod\ a*b)$
using *ab xya xyb*
by (*simp add*: *cong-sub-cases*[*of x y a*] *cong-sub-cases*[*of x y b*]
cong-sub-cases[*of x y a*∗*b*])
(*cases* $x \leq y$, *simp-all add*: *divides-mul*[*of a - b*])

lemma *chinese-remainder-unique*:
assumes *ab*: *coprime a b* **and** *az*: $a \neq 0$ **and** *bz*: $b \neq 0$
shows $\exists!x.\ x < a * b \wedge [x = m]\ (mod\ a) \wedge [x = n]\ (mod\ b)$
proof−
from *az bz* **have** *abpos*: $a*b > 0$ **by** *simp*
from *chinese-remainder*[*OF ab az bz*] **obtain** *x q1 q2* **where**
xq12: $x = m + q1 * a\ x = n + q2 * b$ **by** *blast*
let *?w* = *x mod* (*a*∗*b*)
have *wab*: $?w < a*b$ **by** (*simp add*: *mod-less-divisor*[*OF abpos*])
from *xq12*(*1*) **have** *?w mod a* = ((*m* + *q1* ∗ *a*) *mod* (*a*∗*b*)) *mod a* **by** *simp*
also have ... = *m mod a* **apply** (*simp add*: *mod-mult2-eq*)
apply (*subst mod-add-left-eq*)
by *simp*
finally have *th1*: $[?w = m]\ (mod\ a)$ **by** (*simp add*: *modeq-def*)
from *xq12*(*2*) **have** *?w mod b* = ((*n* + *q2* ∗ *b*) *mod* (*a*∗*b*)) *mod b* **by** *simp*
also have ... = ((*n* + *q2* ∗ *b*) *mod* (*b*∗*a*)) *mod b* **by** (*simp add*: *mult-commute*)

also have ... = *n mod b* **apply** (*simp add*: *mod-mult2-eq*)
apply (*subst mod-add-left-eq*)
by *simp*
finally have *th2*: [*?w* = *n*] (*mod b*) **by** (*simp add*: *modeq-def*)
{fix *y* **assume** *H*: *y* < *a*∗*b* [*y* = *m*] (*mod a*) [*y* = *n*] (*mod b*)
with *th1 th2* **have** *H'*: [*y* = *?w*] (*mod a*) [*y* = *?w*] (*mod b*)
by (*simp-all add*: *modeq-def*)
from *cong-chinese*[*OF ab H'*] *mod-less*[*OF H*(*1*)] *mod-less*[*OF wab*]
have *y* = *?w* **by** (*simp add*: *modeq-def*)**}**
with *th1 th2 wab* **show** *?thesis* **by** *blast*
qed

lemma *chinese-remainder-coprime-unique*:
assumes *ab*: *coprime a b* **and** *az*: *a* ≠ *0* **and** *bz*: *b* ≠ *0*
and *ma*: *coprime m a* **and** *nb*: *coprime n b*
shows ∃!*x*. *coprime x* (*a* ∗ *b*) ∧ *x* < *a* ∗ *b* ∧ [*x* = *m*] (*mod a*) ∧ [*x* = *n*] (*mod b*)
proof−
let *?P* = λ*x*. *x* < *a* ∗ *b* ∧ [*x* = *m*] (*mod a*) ∧ [*x* = *n*] (*mod b*)
from *chinese-remainder-unique*[*OF ab az bz*]
obtain *x* **where** *x*: *x* < *a* ∗ *b* [*x* = *m*] (*mod a*) [*x* = *n*] (*mod b*)
∀ *y*. *?P y* ⟶ *y* = *x* **by** *blast*
from *ma nb cong-coprime*[*OF x*(*2*)] *cong-coprime*[*OF x*(*3*)]
have *coprime x a coprime x b* **by** (*simp-all add*: *coprime-commute*)
with *coprime-mul*[*of x a b*] **have** *coprime x* (*a*∗*b*) **by** *simp*
with *x* **show** *?thesis* **by** *blast*
qed

definition *phi-def*: φ *n* = *card* { *m*. *0* < *m* ∧ *m* <= *n* ∧ *coprime m n* }
lemma *phi-0*[*simp*]: φ *0* = *0*
unfolding *phi-def* **by** (*auto simp add*: *card-eq-0-iff*)

lemma *phi-finite*[*simp*]: *finite* ({ *m*. *0* < *m* ∧ *m* <= *n* ∧ *coprime m n* })
proof−
have { *m*. *0* < *m* ∧ *m* <= *n* ∧ *coprime m n* } ⊆ {*0*..*n*} **by** *auto*
thus *?thesis* **by** (*auto intro*: *finite-subset*)
qed

declare *coprime-1*[*presburger*]
lemma *phi-1*[*simp*]: φ *1* = *1*
proof−
{fix *m*
have *0* < *m* ∧ *m* <= *1* ∧ *coprime m 1* ⟷ *m* = *1* **by** *presburger* **}**
thus *?thesis* **by** (*simp add*: *phi-def*)
qed

lemma [*simp*]: φ (*Suc 0*) = *Suc 0* **using** *phi-1* **by** *simp*

lemma *phi-alt*: $\varphi(n) = card\ \{\ m.\ coprime\ m\ n \wedge m < n\}$
proof−
{assume *n=0* ∨ *n=1* **hence** *?thesis* **by** (*cases n=0*, *simp-all*)**}**
moreover
{assume *n*: *n≠0 n≠1*
{fix *m*
from *n* **have** $0 < m \wedge m <= n \wedge coprime\ m\ n \longleftrightarrow coprime\ m\ n \wedge m < n$
apply (*cases m = 0*, *simp-all*)
apply (*cases m = 1*, *simp-all*)
apply (*cases m = n*, *auto*)
done }
hence *?thesis* **unfolding** *phi-def* **by** *simp***}**
ultimately show *?thesis* **by** *auto*
qed

lemma *phi-finite-lemma*[*simp*]: *finite* {*m*. *coprime m n* ∧ *m* < *n*} (**is** *finite ?S*)
by (*rule finite-subset*[*of ?S* {*0*..*n*}], *auto*)

lemma *phi-another*: **assumes** *n*: *n≠1*
shows $\varphi\ n = card\ \{m.\ 0 < m \wedge m < n \wedge coprime\ m\ n\ \}$
proof−
{fix *m*
from *n* **have** $0 < m \wedge m < n \wedge coprime\ m\ n \longleftrightarrow coprime\ m\ n \wedge m < n$
by (*cases m=0*, *auto*)**}**
thus *?thesis* **unfolding** *phi-alt* **by** *auto*
qed

lemma *phi-limit*: $\varphi\ n \leq n$
proof−
have $\{\ m.\ coprime\ m\ n \wedge m < n\} \subseteq \{0\ ..<n\}$ **by** *auto*
with *card-mono*[*of* {*0* ..<*n*} { *m*. *coprime m n* ∧ *m* < *n*}]
show *?thesis* **unfolding** *phi-alt* **by** *auto*
qed

lemma *stupid*[*simp*]: $\{m.\ (0::nat) < m \wedge m < n\} = \{1..<n\}$
by *auto*

lemma *phi-limit-strong*: **assumes** *n*: *n≠1*
shows $\varphi(n) \leq n - 1$
proof−
show *?thesis*
unfolding *phi-another*[*OF n*] *finite-number-segment*[*of n*, *symmetric*]
by (*rule card-mono*[*of* {*m*. *0* < *m* ∧ *m* < *n*} {*m*. *0* < *m* ∧ *m* < *n* ∧ *coprime m n*}], *auto*)
qed

lemma *phi-lowerbound-1-strong*: **assumes** *n*: $n \geq 1$
shows $\varphi(n) \geq 1$

```
proof−
  let ?S = { m. 0 < m ∧ m <= n ∧ coprime m n }
  from card-0-eq[of ?S] n have φ n ≠ 0 unfolding phi-alt
    apply auto
    apply (cases n=1, simp-all)
    apply (rule exI[where x=1], simp)
    done
  thus ?thesis by arith
qed

lemma phi-lowerbound-1: 2 <= n ==> 1 <= φ(n)
  using phi-lowerbound-1-strong[of n] by auto

lemma phi-lowerbound-2: assumes n: 3 <= n shows 2 <= φ (n)
proof−
  let ?S = { m. 0 < m ∧ m <= n ∧ coprime m n }
  have inS: {1, n − 1} ⊆ ?S using n coprime-plus1[of n − 1]
    by (auto simp add: coprime-commute)
  from n have c2: card {1, n − 1} = 2 by (auto simp add: card-insert-if)
  from card-mono[of ?S {1, n − 1}, simplified inS c2] show ?thesis
    unfolding phi-def by auto
qed

lemma phi-prime: φ n = n − 1 ∧ n≠0 ∧ n≠1 ⟷ prime n
proof−
  {assume n=0 ∨ n=1 hence ?thesis by (cases n=1, simp-all)}
  moreover
  {assume n: n≠0 n≠1
    let ?S = {m. 0 < m ∧ m < n}
    have fS: finite ?S by simp
    let ?S' = {m. 0 < m ∧ m < n ∧ coprime m n}
    have fS':finite ?S' apply (rule finite-subset[of ?S' ?S]) by auto
    {assume H: φ n = n − 1 ∧ n≠0 ∧ n≠1
      hence ceq: card ?S' = card ?S
      using n finite-number-segment[of n] phi-another[OF n(2)] by simp
      {fix m assume m: 0 < m m < n ¬ coprime m n
        hence mS': m ∉ ?S' by auto
        have insert m ?S' ≤ ?S using m by auto
        from m have card (insert m ?S') ≤ card ?S
          by − (rule card-mono[of ?S insert m ?S'], auto)
        hence False
          unfolding card-insert-disjoint[of ?S' m, OF fS' mS'] ceq
          by simp }
      hence ∀ m. 0 <m ∧ m < n ⟶ coprime m n by blast
     hence prime n unfolding prime using n by (simp add: coprime-commute)}
    moreover
    {assume H: prime n
      hence ?S = ?S' unfolding prime using n
        by (auto simp add: coprime-commute)
```

```
    hence card ?S = card ?S' by simp
    hence φ n = n - 1 unfolding phi-another[OF n(2)] by simp}
   ultimately have ?thesis using n by blast}
  ultimately show ?thesis by (cases n=0) blast+
qed



lemma phi-multiplicative: assumes ab: coprime a b
  shows φ (a * b) = φ a * φ b
proof-
  {assume a = 0 ∨ b = 0 ∨ a = 1 ∨ b = 1
    hence ?thesis
      by (cases a=0, simp, cases b=0, simp, casesa=1, simp-all) }
  moreover
  {assume a: a≠0 a≠1 and b: b≠0 b≠1
    hence ab0: a*b ≠ 0 by simp
    let ?S = λk. {m. coprime m k ∧ m < k}
    let ?f = λx. (x mod a, x mod b)
    have eq: ?f ` (?S (a*b)) = (?S a × ?S b)
    proof-
      {fix x assume x:x ∈ ?S (a*b)
        hence x': coprime x (a*b) x < a*b by simp-all
        hence xab: coprime x a coprime x b by (simp-all add: coprime-mul-eq)
        from mod-less-divisor a b have xab':x mod a < a x mod b < b by auto
        from xab xab' have ?f x ∈ (?S a × ?S b)
          by (simp add: coprime-mod[OF a(1)] coprime-mod[OF b(1)])}
      moreover
      {fix x y assume x: x ∈ ?S a and y: y ∈ ?S b
        hence x': coprime x a x < a and y': coprime y b y < b by simp-all
        from chinese-remainder-coprime-unique[OF ab a(1) b(1) x'(1) y'(1)]
        obtain z where z: coprime z (a * b) z < a * b [z = x] (mod a)
          [z = y] (mod b) by blast
        hence (x,y) ∈ ?f ` (?S (a*b))
          using y'(2) mod-less-divisor[of b y] x'(2) mod-less-divisor[of a x]
          by (auto simp add: image-iff modeq-def)}
      ultimately show ?thesis by auto
    qed
    have finj: inj-on ?f (?S (a*b))
      unfolding inj-on-def
    proof(clarify)
      fix x y assume H: coprime x (a * b) x < a * b coprime y (a * b)
        y < a * b x mod a = y mod a x mod b = y mod b
      hence cp: coprime x a coprime x b coprime y a coprime y b
        by (simp-all add: coprime-mul-eq)
      from chinese-remainder-coprime-unique[OF ab a(1) b(1) cp(3,4)] H
      show x = y unfolding modeq-def by blast
    qed
    from card-image[OF finj, unfolded eq] have ?thesis
```

```
    unfolding phi-alt by simp }
  ultimately show ?thesis by auto
qed



lemma nproduct-mod:
  assumes fS: finite S and n0: n ≠ 0
  shows [setprod (λm. a(m) mod n) S = setprod a S] (mod n)
proof−
  have th1:[1 = 1] (mod n) by (simp add: modeq-def)
  from cong-mult
  have th3:∀ x1 y1 x2 y2.
    [x1 = x2] (mod n) ∧ [y1 = y2] (mod n) ⟶ [x1 * y1 = x2 * y2] (mod n)
    by blast
  have th4:∀ x∈S. [a x mod n = a x] (mod n) by (simp add: modeq-def)
  from fold-image-related[where h=(λm. a(m) mod n) and g=a, OF th1 th3 fS,
OF th4] show ?thesis unfolding setprod-def by (simp add: fS)
qed

lemma nproduct-cmul:
  assumes fS:finite S
 shows setprod (λm. (c::'a::{comm-monoid-mult,recpower})* a(m)) S = c ^ (card
S) * setprod a S
unfolding setprod-timesf setprod-constant[OF fS, of c] ..

lemma coprime-nproduct:
  assumes fS: finite S and Sn: ∀ x∈S. coprime n (a x)
  shows coprime n (setprod a S)
  using fS unfolding setprod-def by (rule finite-subset-induct)
    (insert Sn, auto simp add: coprime-mul)

lemma fermat-little: assumes an: coprime a n
  shows [a ^ (φ n) = 1] (mod n)
proof−
  {assume n=0 hence ?thesis by simp}
  moreover
  {assume n=1 hence ?thesis by (simp add: modeq-def)}
  moreover
  {assume nz: n ≠ 0 and n1: n ≠ 1
    let ?S = {m. coprime m n ∧ m < n}
    let ?P = ∏ ?S
    have fS: finite ?S by simp
    have cardfS: φ n = card ?S unfolding phi-alt ..
    {fix m assume m: m ∈ ?S
      hence coprime m n by simp
      with coprime-mul[of n a m] an have coprime (a*m) n
        by (simp add: coprime-commute)}
```

hence *Sn*: $\forall\, m{\in}\ ?S.$ *coprime* $(a{*}m)\ n$ **by** *blast*
from *coprime-nproduct*[*OF fS*, *of n* $\lambda m.\ m$] **have** *nP*:*coprime ?P n*
by (*simp add*: *coprime-commute*)
have *Paphi*: [*?P*$a\hat{}$ $(\varphi\ n) = ?P{*}1$] (*mod n*)
proof−
let *?h* = $\lambda m.\ m$ *mod n*
{fix *m* **assume** *mS*: $m{\in}\ ?S$
hence *?h m* $\in$ *?S* **by** *simp***}**
hence *hS*: *?h* ' *?S* = *?S***by** (*auto simp add*: *image-iff*)
have $a{\neq}0$ **using** *an n1 nz* **apply**− **apply** (*rule ccontr*) **by** *simp*
hence *inj*: *inj-on* (*op* $*$ *a*) *?S* **unfolding** *inj-on-def* **by** *simp*

have *eq0*: *fold-image op* $*$ (*?h* $\circ$ *op* $*$ *a*) *1* $\{m.\ coprime\ m\ n \wedge m < n\}$ =
fold-image op $*$ ($\lambda m.\ m$) *1* $\{m.\ coprime\ m\ n \wedge m < n\}$
proof (*rule fold-image-eq-general*[**where** *h*=*?h o* (*op* $*$ *a*)])
show *finite ?S* **using** *fS* **.**
next
{fix *y* **assume** *yS*: $y \in ?S$ **hence** *y*: *coprime y n* $y < n$ **by** *simp-all*
from *cong-solve-unique*[*OF an nz*, *of y*]
obtain *x* **where** *x*:$x < n$ $[a * x = y]$ (*mod n*) $\forall z.\ z < n \wedge [a * z = y]$
(*mod n*) $\longrightarrow$ *z*=*x* **by** *blast*
from *cong-coprime*[*OF x*(*2*)] *y*(*1*)
have *xm*: *coprime x n* **by** (*simp add*: *coprime-mul-eq coprime-commute*)
{fix *z* **assume** $z \in ?S$ (*?h* $\circ$ *op* $*$ *a*) $z = y$
hence *z*: *coprime z n z* $< n$ (*?h* $\circ$ *op* $*$ *a*) $z = y$ **by** *simp-all*
from *x*(*3*)[*rule-format*, *of z*] *z*(*2*,*3*) **have** *z*=*x*
unfolding *modeq-def mod-less*[*OF y*(*2*)] **by** *simp***}**
with *xm x*(*1*,*2*) **have** $\exists!x.\ x \in ?S \wedge$ (*?h* $\circ$ *op* $*$ *a*) $x = y$
unfolding *modeq-def mod-less*[*OF y*(*2*)] **by** *auto* **}**
thus $\forall\, y{\in}\{m.\ coprime\ m\ n \wedge m < n\}.$
$\exists!x.\ x \in \{m.\ coprime\ m\ n \wedge m < n\} \wedge ((\lambda m.\ m\ mod\ n) \circ op * a)\ x = y$
by *blast*
next
{fix *x* **assume** *xS*: $x{\in}\ ?S$
hence *x*: *coprime x n x* $< n$ **by** *simp-all*
with *an* **have** *coprime* $(a{*}x)\ n$
by (*simp add*: *coprime-mul-eq*[*of n a x*] *coprime-commute*)
hence *?h* $(a{*}x) \in ?S$ **using** *nz*
by (*simp add*: *coprime-mod*[*OF nz*] *mod-less-divisor*)**}**
thus $\forall\, x{\in}\{m.\ coprime\ m\ n \wedge m < n\}.$
$((\lambda m.\ m\ mod\ n) \circ op * a)\ x \in \{m.\ coprime\ m\ n \wedge m < n\} \wedge$
$((\lambda m.\ m\ mod\ n) \circ op * a)\ x = ((\lambda m.\ m\ mod\ n) \circ op * a)\ x$ **by** *simp*
qed
from *nproduct-mod*[*OF fS nz*, *of op* $*$ *a*]
have [(*setprod* (*op* $*a$) *?S*) = (*setprod* (*?h o* (*op* $*$ *a*)) *?S*)] (*mod n*)
unfolding *o-def*
by (*simp add*: *cong-commute*)
also have [*setprod* (*?h o* (*op* $*$ *a*)) *?S* = *?P*] (*mod n*)
using *eq0 fS an* **by** (*simp add*: *setprod-def modeq-def o-def*)

```
    finally show [?P*a^ (φ n) = ?P*1] (mod n)
      unfolding cardfS mult-commute[of ?P a^ (card ?S)]
        nproduct-cmul[OF fS, symmetric] mult-1-right by simp
  qed
  from cong-mult-lcancel[OF nP Paphi] have ?thesis . }
 ultimately show ?thesis by blast
qed

lemma fermat-little-prime: assumes p: prime p and ap: coprime a p
 shows [a^ (p - 1) = 1] (mod p)
 using fermat-little[OF ap] p[unfolded phi-prime[symmetric]]
by simp



lemma lucas-coprime-lemma:
 assumes m: m≠0 and am: [a^m = 1] (mod n)
 shows coprime a n
proof−
 {assume n=1 hence ?thesis by simp}
 moreover
 {assume n = 0 hence ?thesis using am m exp-eq-1[of a m] by simp}
 moreover
 {assume n: n≠0 n≠1
  from m obtain m' where m': m = Suc m' by (cases m, blast+)
  {fix d
   assume d: d dvd a d dvd n
   from n have n1: 1 < n by arith
   from am mod-less[OF n1] have am1: a^m mod n = 1 unfolding modeq-def
by simp
   from dvd-mult2[OF d(1), of a^m'] have dam:d dvd a^m by (simp add: m')
   from dvd-mod-iff[OF d(2), of a^m] dam am1
   have d = 1 by simp }
  hence ?thesis unfolding coprime by auto
 }
 ultimately show ?thesis by blast
qed

lemma lucas-weak:
 assumes n: n ≥ 2 and an:[a^(n - 1) = 1] (mod n)
 and nm: ∀ m. 0 <m ∧ m < n - 1 ⟶ ¬ [a^m = 1] (mod n)
 shows prime n
proof−
 from n have n1: n ≠ 1 n≠0 n - 1 ≠ 0 n - 1 > 0 n - 1 < n by arith+
 from lucas-coprime-lemma[OF n1(3) an] have can: coprime a n .
 from fermat-little[OF can] have afn: [a ^ φ n = 1] (mod n) .
 {assume φ n ≠ n - 1
  with phi-limit-strong[OF n1(1)] phi-lowerbound-1[OF n]
```

```
    have c:φ n > 0 ∧ φ n < n − 1 by arith
    from nm[rule-format, OF c] afn have False ..}
  hence φ n = n − 1 by blast
  with phi-prime[of n] n1(1,2) show ?thesis by simp
qed

lemma nat-exists-least-iff: (∃(n::nat). P n) ⟷ (∃n. P n ∧ (∀m < n. ¬ P m))
  (is ?lhs ⟷ ?rhs)
proof
  assume ?rhs thus ?lhs by blast
next
  assume H: ?lhs then obtain n where n: P n by blast
  let ?x = Least P
  {fix m assume m: m < ?x
    from not-less-Least[OF m] have ¬ P m .}
  with LeastI-ex[OF H] show ?rhs by blast
qed

lemma nat-exists-least-iff': (∃(n::nat). P n) ⟷ (P (Least P) ∧ (∀m < (Least
P). ¬ P m))
  (is ?lhs ⟷ ?rhs)
proof−
  {assume ?rhs hence ?lhs by blast}
  moreover
  { assume H: ?lhs then obtain n where n: P n by blast
    let ?x = Least P
    {fix m assume m: m < ?x
      from not-less-Least[OF m] have ¬ P m .}
    with LeastI-ex[OF H] have ?rhs by blast}
  ultimately show ?thesis by blast
qed

lemma power-mod: ((x::nat) mod m) ^n mod m = x^n mod m
proof(induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  have (x mod m) ^(Suc n) mod m = ((x mod m) * (((x mod m) ^ n) mod m))
mod m
    by (simp add: mod-mult-right-eq[symmetric])
  also have ... = ((x mod m) * (x^n mod m)) mod m using Suc.hyps by simp
  also have ... = x^(Suc n) mod m
    by (simp add: mod-mult-left-eq[symmetric] mod-mult-right-eq[symmetric])
  finally show ?case .
qed

lemma lucas:
  assumes n2: n ≥ 2 and an1: [a^(n − 1) = 1] (mod n)
  and pn: ∀p. prime p ∧ p dvd n − 1 ⟶ ¬ [a^((n − 1) div p) = 1] (mod n)
```

```
  shows prime n
proof−
  from n2 have n01: n≠0 n≠1 n − 1 ≠ 0 by arith+
  from mod-less-divisor[of n 1] n01 have onen: 1 mod n = 1 by simp
  from lucas-coprime-lemma[OF n01(3) an1] cong-coprime[OF an1]
 have an: coprime a n coprime (a^(n − 1)) n by (simp-all add: coprime-commute)
  {assume H0: ∃m. 0 < m ∧ m < n − 1 ∧ [a ^ m = 1] (mod n) (is EX m. ?P
m)
    from H0[unfolded nat-exists-least-iff[of ?P]] obtain m where
      m: 0 < m m < n − 1 [a ^ m = 1] (mod n) ∀k <m. ¬?P k by blast
    {assume nm1: (n − 1) mod m > 0
      from mod-less-divisor[OF m(1)] have th0:(n − 1) mod m < m by blast
      let ?y = a^ ((n − 1) div m * m)
      note mdeq = mod-div-equality[of (n − 1) m]
      from coprime-exp[OF an(1)[unfolded coprime-commute[of a n]],
        of (n − 1) div m * m]
      have yn: coprime ?y n by (simp add: coprime-commute)
      have ?y mod n = (a^m)^((n − 1) div m) mod n
        by (simp add: algebra-simps power-mult)
      also have ... = (a^m mod n)^((n − 1) div m) mod n
        using power-mod[of a^m n (n − 1) div m] by simp
      also have ... = 1 using m(3)[unfolded modeq-def onen] onen
        by (simp add: power-Suc0)
      finally have th3: ?y mod n = 1 .
      have th2: [?y * a ^ ((n − 1) mod m) = ?y* 1] (mod n)
        using an1[unfolded modeq-def onen] onen
          mod-div-equality[of (n − 1) m, symmetric]
        by (simp add:power-add[symmetric] modeq-def th3 del: One-nat-def)
      from cong-mult-lcancel[of ?y n a^((n − 1) mod m) 1, OF yn th2]
      have th1: [a ^ ((n − 1) mod m) = 1] (mod n) .
      from m(4)[rule-format, OF th0] nm1
        less-trans[OF mod-less-divisor[OF m(1), of n − 1] m(2)] th1
      have False by blast }
    hence (n − 1) mod m = 0 by auto
    then have mn: m dvd n − 1 by presburger
    then obtain r where r: n − 1 = m*r unfolding dvd-def by blast
    from n01 r m(2) have r01: r≠0 r≠1 by − (rule ccontr, simp)+
    from prime-factor[OF r01(2)] obtain p where p: prime p p dvd r by blast
    hence th: prime p ∧ p dvd n − 1 unfolding r by (auto intro: dvd-mult)
    have (a ^ ((n − 1) div p)) mod n = (a^(m*r div p)) mod n using r
      by (simp add: power-mult)
      also have ... = (a^(m*(r div p))) mod n using div-mult1-eq[of m r p]
p(2)[unfolded dvd-eq-mod-eq-0] by simp
    also have ... = ((a^m)^(r div p)) mod n by (simp add: power-mult)
    also have ... = ((a^m mod n)^(r div p)) mod n using power-mod[of a^m n r
div p ] ..
    also have ... = 1 using m(3) onen by (simp add: modeq-def power-Suc0)
    finally have [(a ^ ((n − 1) div p))= 1] (mod n)
      using onen by (simp add: modeq-def)
```

with *pn*[*rule-format*, *OF th*] **have** *False* **by** *blast*}
hence *th*: ∀ *m*. *0* < *m* ∧ *m* < *n* − *1* ⟶ ¬ [*a* ^ *m* = *1*] (*mod n*) **by** *blast*
from *lucas-weak*[*OF n2 an1 th*] **show** *?thesis* **.**
qed

definition *ord n a* = (*if coprime n a then Least* (λ*d*. *d* > *0* ∧ [*a* ^*d* = *1*] (*mod n*)) *else 0*)

lemma *coprime-ord*:
assumes *na*: *coprime n a*
shows *ord n a* > *0* ∧ [*a* ^(*ord n a*) = *1*] (*mod n*) ∧ (∀ *m*. *0* < *m* ∧ *m* < *ord n a* ⟶ ¬ [*a*^ *m* = *1*] (*mod n*))
proof−
let *?P* = λ*d*. *0* < *d* ∧ [*a* ^ *d* = *1*] (*mod n*)
from *euclid*[*of a*] **obtain** *p* **where** *p*: *prime p a* < *p* **by** *blast*
from *na* **have** *o*: *ord n a* = *Least ?P* **by** (*simp add*: *ord-def*)
{assume *n*=*0* ∨ *n*=*1* **with** *na* **have** ∃ *m*>*0*. *?P m* **apply** *auto* **apply** (*rule exI*[**where** *x*=*1*]) **by** (*simp add*: *modeq-def*)**}**
moreover
{assume *n*≠*0* ∧ *n*≠*1* **hence** *n2*:*n* ≥ *2* **by** *arith*
from *na* **have** *na'*: *coprime a n* **by** (*simp add*: *coprime-commute*)
from *phi-lowerbound-1*[*OF n2*] *fermat-little*[*OF na'*]
have *ex*: ∃ *m*>*0*. *?P m* **by** − (*rule exI*[**where** *x*=φ *n*], *auto*) **}**
ultimately have *ex*: ∃ *m*>*0*. *?P m* **by** *blast*
from *nat-exists-least-iff'*[*of ?P*] *ex na* **show** *?thesis*
unfolding *o*[*symmetric*] **by** *auto*
qed

lemma *ord-works*:
[*a* ^ (*ord n a*) = *1*] (*mod n*) ∧ (∀ *m*. *0* < *m* ∧ *m* < *ord n a* ⟶ ~[*a*^ *m* = *1*] (*mod n*))
apply (*cases coprime n a*)
using *coprime-ord*[*of n a*]
by (*blast*, *simp add*: *ord-def modeq-def*)

lemma *ord*: [*a*^(*ord n a*) = *1*] (*mod n*) **using** *ord-works* **by** *blast*
lemma *ord-minimal*: *0* < *m* ⟹ *m* < *ord n a* ⟹ ~[*a*^*m* = *1*] (*mod n*)
using *ord-works* **by** *blast*
lemma *ord-eq-0*: *ord n a* = *0* ⟷ ~*coprime n a*
by (*cases coprime n a*, *simp add*: *neq0-conv coprime-ord*, *simp add*: *neq0-conv ord-def*)

lemma *ord-divides*:
[*a* ^ *d* = *1*] (*mod n*) ⟷ *ord n a dvd d* (**is** *?lhs* ⟷ *?rhs*)
proof

```
  assume rh: ?rhs
  then obtain k where d = ord n a * k unfolding dvd-def by blast
  hence [a ^ d = (a ^ (ord n a) mod n) ^k] (mod n)
    by (simp add : modeq-def power-mult power-mod)
  also have [(a ^ (ord n a) mod n) ^k = 1] (mod n)
    using ord[of a n, unfolded modeq-def]
    by (simp add: modeq-def power-mod power-Suc0)
  finally  show ?lhs .
next
  assume lh: ?lhs
  { assume H: ¬ coprime n a
    hence o: ord n a = 0 by (simp add: ord-def)
    {assume d: d=0 with o H have ?rhs by (simp add: modeq-def)}
    moreover
    {assume d0: d≠0 then obtain d' where d': d = Suc d' by (cases d, auto)
      from H[unfolded coprime]
      obtain p where p: p dvd n p dvd a p ≠ 1 by auto
      from lh[unfolded nat-mod]
      obtain q1 q2 where q12:a ^ d + n * q1 = 1 + n * q2 by blast
      hence a ^ d + n * q1 − n * q2 = 1 by simp
     with nat-dvd-diff [OF dvd-add [OF divides-rexp[OF p(2), of d'] dvd-mult2[OF
p(1), of q1]] dvd-mult2[OF p(1), of q2]] d' have p dvd 1 by simp
      with p(3) have False by simp
      hence ?rhs ..}
    ultimately have ?rhs by blast}
  moreover
  {assume H: coprime n a
    let ?o = ord n a
    let ?q = d div ord n a
    let ?r = d mod ord n a
    from cong-exp[OF ord[of a n], of ?q]
    have eqo: [(a^?o) ^?q = 1] (mod n)  by (simp add: modeq-def power-Suc0)
    from H have onz: ?o ≠ 0 by (simp add: ord-eq-0)
    hence op: ?o > 0 by simp
    from mod-div-equality[of d ord n a] lh
    have [a^(?o*?q + ?r) = 1] (mod n) by (simp add: modeq-def mult-commute)
    hence [(a^?o) ^?q * (a^?r) = 1] (mod n)
      by (simp add: modeq-def power-mult[symmetric] power-add[symmetric])
    hence th: [a^?r = 1] (mod n)
      using eqo mod-mult-left-eq[of (a^?o) ^?q a^?r n]
      apply (simp add: modeq-def del: One-nat-def)
      by (simp add: mod-mult-left-eq[symmetric])
    {assume r: ?r = 0 hence ?rhs by (simp add: dvd-eq-mod-eq-0)}
    moreover
    {assume r: ?r ≠ 0
      with mod-less-divisor[OF op, of d] have r0o:?r >0 ∧ ?r < ?o by simp
      from conjunct2[OF ord-works[of a n], rule-format, OF r0o] th
      have ?rhs by blast}
    ultimately have ?rhs by blast}
```

ultimately show *?rhs* **by** *blast*
qed

lemma *order-divides-phi*: *coprime n a* $\Longrightarrow$ *ord n a dvd* φ *n*
using *ord-divides fermat-little coprime-commute* **by** *simp*
lemma *order-divides-expdiff*:
assumes *na*: *coprime n a*
shows $[a\hat{}d = a\hat{}e]$ (*mod n*) $\longleftrightarrow$ $[d = e]$ (*mod* (*ord n a*))
proof−
{fix *n a d e*
assume *na*: *coprime n a* **and** *ed*: (*e*::*nat*) $\leq$ *d*
hence $\exists\, c.\ d = e + c$ **by** *arith*
then obtain *c* **where** *c*: $d = e + c$ **by** *arith*
from *na* **have** *an*: *coprime a n* **by** (*simp add*: *coprime-commute*)
from *coprime-exp*[*OF na*, *of e*]
have *aen*: *coprime* $(a\hat{}e)$ *n* **by** (*simp add*: *coprime-commute*)
from *coprime-exp*[*OF na*, *of c*]
have *acn*: *coprime* $(a\hat{}c)$ *n* **by** (*simp add*: *coprime-commute*)
have $[a\hat{}d = a\hat{}e]$ (*mod n*) $\longleftrightarrow$ $[a\hat{}(e + c) = a\hat{}(e + 0)]$ (*mod n*)
using *c* **by** *simp*
also have ... $\longleftrightarrow$ $[a\hat{}e* a\hat{}c = a\hat{}e *a\hat{}0]$ (*mod n*) **by** (*simp add*: *power-add*)
also have ... $\longleftrightarrow$ $[a\ \hat{}\ c = 1]$ (*mod n*)
using *cong-mult-lcancel-eq*[*OF aen*, *of a*ˆ*c a*ˆ*0*] **by** *simp*
also have ... $\longleftrightarrow$ *ord n a dvd c* **by** (*simp only*: *ord-divides*)
also have ... $\longleftrightarrow$ $[e + c = e + 0]$ (*mod ord n a*)
using *cong-add-lcancel-eq*[*of e c 0 ord n a*, *simplified cong-0-divides*]
by *simp*
finally have $[a\hat{}d = a\hat{}e]$ (*mod n*) $\longleftrightarrow$ $[d = e]$ (*mod* (*ord n a*))
using *c* **by** *simp* **}**
note *th* = *this*
have $e \leq d \vee d \leq e$ **by** *arith*
moreover
{assume *ed*: $e \leq d$ **from** *th*[*OF na ed*] **have** *?thesis* **.}**
moreover
{assume *de*: $d \leq e$
from *th*[*OF na de*] **have** *?thesis* **by** (*simp add*: *cong-commute*) **}**
ultimately show *?thesis* **by** *blast*
qed

lemma *prime-prime-factor*:
prime n $\longleftrightarrow$ $n \neq 1 \wedge$ $(\forall\, p.$ *prime p* $\wedge$ *p dvd n* $\longrightarrow$ $p = n)$
proof−
{assume *n*: *n=0* $\vee$ *n=1* **hence** *?thesis* **using** *prime-0 two-is-prime* **by** *auto***}**
moreover
{assume *n*: *n≠0 n≠1*
{assume *pn*: *prime n*

```
    from pn[unfolded prime-def] have ∀p. prime p ∧ p dvd n ⟶ p = n
      using n
      apply (cases n = 0 ∨ n=1,simp)
      by (clarsimp, erule-tac x=p in allE, auto)}
   moreover
   {assume H: ∀p. prime p ∧ p dvd n ⟶ p = n
    from n have n1: n > 1 by arith
    {fix m assume m: m dvd n m≠1
     from prime-factor[OF m(2)] obtain p where
       p: prime p p dvd m by blast
     from dvd-trans[OF p(2) m(1)] p(1) H have p = n by blast
     with p(2) have n dvd m  by simp
     hence m=n  using dvd-anti-sym[OF m(1)] by simp }
    with n1 have prime n  unfolding prime-def by auto }
   ultimately have ?thesis using n by blast}
  ultimately      show ?thesis by auto
qed

lemma prime-divisor-sqrt:
  prime n ⟷ n ≠ 1 ∧ (∀d. d dvd n ∧ d^2 ≤ n ⟶ d = 1)
proof−
  {assume n=0 ∨ n=1 hence ?thesis using prime-0 prime-1
    by (auto simp add: nat-power-eq-0-iff)}
  moreover
  {assume n: n≠0 n≠1
    hence np: n > 1 by arith
    {fix d assume d: d dvd n d^2 ≤ n and H: ∀m. m dvd n ⟶ m=1 ∨ m=n
     from H d have d1n: d = 1 ∨ d=n by blast
     {assume dn: d=n
      have n^2 > n*1 using n
        by (simp add: power2-eq-square mult-less-cancel1)
      with dn d(2) have d=1 by simp}
     with d1n have d = 1 by blast  }
    moreover
    {fix d assume d: d dvd n and H: ∀d'. d' dvd n ∧ d'^2 ≤ n ⟶ d' = 1
     from d n have d ≠ 0 apply − apply (rule ccontr) by simp
     hence dp: d > 0 by simp
     from d[unfolded dvd-def] obtain e where e: n= d*e by blast
     from n dp e have ep:e > 0 by simp
     have d^2 ≤ n ∨ e^2 ≤ n using dp ep
       by (auto simp add: e power2-eq-square mult-le-cancel-left)
     moreover
     {assume h: d^2 ≤ n
      from H[rule-format, of d] h d have d = 1 by blast}
     moreover
     {assume h: e^2 ≤ n
      from e have e dvd n unfolding dvd-def by (simp add: mult-commute)
      with H[rule-format, of e] h have e=1 by simp
      with e have d = n by simp}
```

```
    ultimately have d=1 ∨ d=n  by blast}
   ultimately have ?thesis unfolding prime-def using np n(2) by blast}
  ultimately show ?thesis by auto
qed
lemma prime-prime-factor-sqrt:
  prime n ⟷ n ≠ 0 ∧ n ≠ 1 ∧ ¬ (∃p. prime p ∧ p dvd n ∧ p^2 ≤ n)
  (is ?lhs ⟷?rhs)
proof−
  {assume n=0 ∨ n=1 hence ?thesis using prime-0 prime-1 by auto}
  moreover
  {assume n: n≠0 n≠1
    {assume H: ?lhs
     from H[unfolded prime-divisor-sqrt] n
     have ?rhs  apply clarsimp by (erule-tac x=p in allE, simp add: prime-1)
    }
    moreover
    {assume H: ?rhs
     {fix d assume d: d dvd n d^2 ≤ n d≠1
      from prime-factor[OF d(3)]
      obtain p where p: prime p p dvd d by blast
      from n have np: n > 0 by arith
      from d(1) n have d ≠ 0 by − (rule ccontr, auto)
      hence dp: d > 0 by arith
      from mult-mono[OF dvd-imp-le[OF p(2) dp] dvd-imp-le[OF p(2) dp]] d(2)
      have p^2 ≤ n unfolding power2-eq-square by arith
      with H n p(1) dvd-trans[OF p(2) d(1)] have False  by blast}
     with n prime-divisor-sqrt  have ?lhs by auto}
    ultimately have ?thesis by blast }
  ultimately show ?thesis by (cases n=0 ∨ n=1, auto)
qed


lemma pocklington-lemma:
  assumes n: n ≥ 2 and nqr: n − 1 = q∗r and an: [a^ (n − 1) = 1] (mod n)
  and aq:∀p. prime p ∧ p dvd q ⟶ coprime (a^ ((n − 1) div p) − 1) n
  and pp: prime p and pn: p dvd n
  shows [p = 1] (mod q)
proof−
  from pp prime-0 prime-1 have p01: p ≠ 0 p ≠ 1 by − (rule ccontr, simp)+
  from cong-1-divides[OF an, unfolded nqr, unfolded dvd-def]
  obtain k where k: a ^ (q ∗ r) − 1 = n∗k by blast
  from pn[unfolded dvd-def] obtain l where l: n = p∗l by blast
  {assume a0: a = 0
    hence a^ (n − 1) = 0 using n by (simp add: power-0-left)
   with n an mod-less[of 1 n]  have False by (simp add: power-0-left modeq-def)}
  hence a0: a≠0 ..
  from n nqr have aqr0: a ^ (q ∗ r) ≠ 0 using a0 by (simp add: neq0-conv)
  hence (a ^ (q ∗ r) − 1) + 1  = a ^ (q ∗ r) by simp
  with k l have a ^ (q ∗ r) = p∗l∗k + 1 by simp
```

hence $a \hat{} (r * q) + p * 0 = 1 + p * (l*k)$ **by** (*simp add*: *mult-ac*)
hence *odq*: *ord p* $(a \hat{} r)$ *dvd q*
unfolding *ord-divides*[*symmetric*] *power-mult*[*symmetric*] *nat-mod* **by** *blast*
from *odq*[*unfolded dvd-def*] **obtain** *d* **where** *d*: $q = ord\ p\ (a \hat{} r) * d$ **by** *blast*
{assume *d1*: $d \neq 1$
from *prime-factor*[*OF d1*] **obtain** *P* **where** *P*: *prime P P dvd d* **by** *blast*
from *d dvd-mult*[*OF P*(2), *of ord p* $(a \hat{} r)$] **have** *Pq*: *P dvd q* **by** *simp*
from *aq P*(1) *Pq* **have** *caP*:*coprime* $(a \hat{} ((n - 1)\ div\ P) - 1)\ n$ **by** *blast*
from *Pq* **obtain** *s* **where** *s*: $q = P*s$ **unfolding** *dvd-def* **by** *blast*
have *P0*: $P \neq 0$ **using** *P*(1) *prime-0* **by** − (*rule ccontr*, *simp*)
from *P*(2) **obtain** *t* **where** *t*: $d = P*t$ **unfolding** *dvd-def* **by** *blast*
from *d s t P0* **have** *s'*: $ord\ p\ (a \hat{} r) * t = s$ **by** *algebra*
have $ord\ p\ (a \hat{} r) * t*r = r * ord\ p\ (a \hat{} r) * t$ **by** *algebra*
hence *exps*: $a \hat{} (ord\ p\ (a \hat{} r) * t*r) = ((a \hat{}\ r) \hat{}\ ord\ p\ (a \hat{} r)) \hat{}\ t$
by (*simp only*: *power-mult*)
have $[((a \hat{}\ r) \hat{}\ ord\ p\ (a \hat{} r)) \hat{}\ t = 1 \hat{} t]\ (mod\ p)$
by (*rule cong-exp*, *rule ord*)
then have *th*: $[((a \hat{}\ r) \hat{}\ ord\ p\ (a \hat{} r)) \hat{}\ t = 1]\ (mod\ p)$
by (*simp add*: *power-Suc0*)
from *cong-1-divides*[*OF th*] *exps* **have** *pd0*: $p\ dvd\ a \hat{} (ord\ p\ (a \hat{} r) * t*r) - 1$
by *simp*
from *nqr s s'* **have** $(n - 1)\ div\ P = ord\ p\ (a \hat{} r) * t*r$ **using** *P0* **by** *simp*
with *caP* **have** *coprime* $(a \hat{} (ord\ p\ (a \hat{} r) * t*r) - 1)\ n$ **by** *simp*
with *p01 pn pd0* **have** *False* **unfolding** *coprime* **by** *auto*}
hence *d1*: $d = 1$ **by** *blast*
hence *o*: *ord p* $(a \hat{} r) = q$ **using** *d* **by** *simp*
from *pp phi-prime*[*of p*] **have** *phip*: $\varphi\ p = p - 1$ **by** *simp*
{fix *d* **assume** *d*: *d dvd p d dvd a d* $\neq 1$
from *pp*[*unfolded prime-def*] *d* **have** *dp*: $d = p$ **by** *blast*
from *n* **have** *n12*:$Suc\ (n - 2) = n - 1$ **by** *arith*
with *divides-rexp*[*OF d*(2)[*unfolded dp*], *of n − 2*]
have *th0*: *p dvd a* $\hat{}\ (n - 1)$ **by** *simp*
from *n* **have** *n0*: $n \neq 0$ **by** *simp*
from *d*(2) *an n12*[*symmetric*] **have** *a0*: $a \neq 0$
by − (*rule ccontr*, *simp add*: *modeq-def*)
have *th1*: $a \hat{}\ (n - 1) \neq 0$ **using** *n d*(2) *dp a0* **by** (*auto simp add*: *neq0-conv*)
from *coprime-minus1*[*OF th1*, *unfolded coprime*]
dvd-trans[*OF pn cong-1-divides*[*OF an*]] *th0 d*(3) *dp*
have *False* **by** *auto*}
hence *cpa*: *coprime p a* **using** *coprime* **by** *auto*
from *coprime-exp*[*OF cpa*, *of r*] *coprime-commute*
have *arp*: *coprime* $(a \hat{} r)\ p$ **by** *blast*
from *fermat-little*[*OF arp*, *simplified ord-divides*] *o phip*
have *q dvd* $(p - 1)$ **by** *simp*
then obtain *d* **where** *d*:$p - 1 = q * d$ **unfolding** *dvd-def* **by** *blast*
from *prime-0 pp* **have** *p0*:$p \neq 0$ **by** − (*rule ccontr*, *auto*)
from *p0 d* **have** $p + q * 0 = 1 + q * d$ **by** *simp*
with *nat-mod*[*of p 1 q*, *symmetric*]
show *?thesis* **by** *blast*

qed

lemma *pocklington*:
assumes *n*: $n \geq 2$ **and** *nqr*: $n - 1 = q*r$ **and** *sqr*: $n \leq q\hat{}2$
and *an*: $[a\hat{}\ (n - 1) = 1]\ (mod\ n)$
and *aq*:$\forall p.\ prime\ p \wedge p\ dvd\ q \longrightarrow coprime\ (a\hat{}\ ((n - 1)\ div\ p) - 1)\ n$
shows *prime n*
unfolding *prime-prime-factor-sqrt*[*of n*]
proof−
let *?ths* = $n \neq 0 \wedge n \neq 1 \wedge \neg\ (\exists p.\ prime\ p \wedge p\ dvd\ n \wedge p^2 \leq n)$
from *n* **have** *n01*: *n≠0 n≠1* **by** *arith+*
{fix *p* **assume** *p*: *prime p p dvd n p^2* ≤ *n*
from *p*(*3*) *sqr* **have** *p^*(*Suc 1*) ≤ *q^*(*Suc 1*) **by** (*simp add*: *power2-eq-square*)
hence *pq*: *p* ≤ *q* **unfolding** *exp-mono-le* **.**
from *pocklington-lemma*[*OF n nqr an aq p*(*1*,*2*)] *cong-1-divides*
have *th*: *q dvd p − 1* **by** *blast*
have *p − 1 ≠ 0***using** *prime-ge-2*[*OF p*(*1*)] **by** *arith*
with *divides-ge*[*OF th*] *pq* **have** *False* **by** *arith* **}**
with *n01* **show** *?ths* **by** *blast*
qed

lemma *pocklington-alt*:
assumes *n*: $n \geq 2$ **and** *nqr*: $n - 1 = q*r$ **and** *sqr*: $n \leq q\hat{}2$
and *an*: $[a\hat{}\ (n - 1) = 1]\ (mod\ n)$
and *aq*:$\forall p.\ prime\ p \wedge p\ dvd\ q \longrightarrow (\exists b.\ [a\hat{}((n - 1)\ div\ p) = b]\ (mod\ n) \wedge$
coprime $(b - 1)\ n$)
shows *prime n*
proof−
{fix *p* **assume** *p*: *prime p p dvd q*
from *aq*[*rule-format*] *p* **obtain** *b* **where**
b: $[a\hat{}((n - 1)\ div\ p) = b]\ (mod\ n)$ *coprime* $(b - 1)\ n$ **by** *blast*
{assume *a0*: *a=0*
from *n an* **have** $[0 = 1]\ (mod\ n)$ **unfolding** *a0 power-0-left* **by** *auto*
hence *False* **using** *n* **by** (*simp add*: *modeq-def dvd-eq-mod-eq-0*[*symmetric*])**}**
hence *a0*: *a≠ 0* **..**
hence *a1*: $a \geq 1$ **by** *arith*
from *one-le-power*[*OF a1*] **have** *ath*: $1 \leq a\ \hat{}\ ((n - 1)\ div\ p)$ **.**
{assume *b0*: *b = 0*
from *p*(*2*) *nqr* **have** $(n - 1)\ mod\ p = 0$
apply (*simp only*: *dvd-eq-mod-eq-0*[*symmetric*]) **by** (*rule dvd-mult2*, *simp*)
with *mod-div-equality*[*of n − 1 p*]
have $(n - 1)\ div\ p * p = n - 1$ **by** *auto*
hence *eq*: $(a\hat{}((n - 1)\ div\ p))\ \hat{}p = a\hat{}(n - 1)$
by (*simp only*: *power-mult*[*symmetric*])
from *prime-ge-2*[*OF p*(*1*)] **have** *pS*: *Suc* $(p - 1) = p$ **by** *arith*
from *b*(*1*) **have** *d*: *n dvd a^*((*n − 1*) *div p*) **unfolding** *b0 cong-0-divides* **.**
from *divides-rexp*[*OF d*, *of p − 1*] *pS eq cong-divides*[*OF an*] *n*
have *False* **by** *simp***}**

```
    then have b0: b ≠ 0 ..
    hence b1: b ≥ 1 by arith
    from cong-coprime[OF cong-sub[OF b(1) cong-refl[of 1] ath b1]] b(2) nqr
    have coprime (a ^ ((n − 1) div p) − 1) n by (simp add: coprime-commute)}
  hence th: ∀ p. prime p ∧ p dvd q ⟶ coprime (a ^ ((n − 1) div p) − 1) n
    by blast
  from pocklington[OF n nqr sqr an th] show ?thesis .
qed



definition primefact ps n = (foldr op ∗ ps  1 = n ∧ (∀ p∈ set ps. prime p))

lemma primefact: assumes n: n ≠ 0
  shows ∃ ps. primefact ps n
using n
proof(induct n rule: nat-less-induct)
  fix n assume H: ∀ m<n. m ≠ 0 ⟶ (∃ ps. primefact ps m) and n: n≠0
  let ?ths = ∃ ps. primefact ps n
  {assume n = 1
    hence primefact [] n by (simp add: primefact-def)
    hence ?ths by blast }
  moreover
  {assume n1: n ≠ 1
    with n have n2: n ≥ 2 by arith
    from prime-factor[OF n1] obtain p where p: prime p p dvd n by blast
    from p(2) obtain m where m: n = p∗m unfolding dvd-def by blast
    from n m have m0: m > 0 m≠0 by auto
    from prime-ge-2[OF p(1)] have 1 < p by arith
    with m0 m have mn: m < n by auto
    from H[rule-format, OF mn m0(2)] obtain ps where ps: primefact ps m ..
    from ps m p(1) have primefact (p#ps) n by (simp add: primefact-def)
    hence ?ths by blast}
  ultimately show ?ths by blast
qed

lemma primefact-contains:
  assumes pf: primefact ps n and p: prime p and pn: p dvd n
  shows p ∈ set ps
  using pf p pn
proof(induct ps arbitrary: p n)
  case Nil thus ?case by (auto simp add: primefact-def)
next
  case (Cons q qs p n)
  from Cons.prems[unfolded primefact-def]
  have q: prime q q ∗ foldr op ∗ qs 1 = n ∀ p ∈set qs. prime p  and p: prime p p
dvd q ∗ foldr op ∗ qs 1 by simp-all
  {assume p dvd q
    with p(1) q(1) have p = q unfolding prime-def by auto
```

hence *?case* **by** *simp*}
moreover
{ **assume** *h*: *p dvd foldr op ∗ qs 1*
from *q*(*3*) **have** *pqs*: *primefact qs* (*foldr op ∗ qs 1*)
by (*simp add*: *primefact-def*)
from *Cons.hyps*[*OF pqs p*(*1*) *h*] **have** *?case* **by** *simp*}
ultimately show *?case* **using** *prime-divprod*[*OF p*] **by** *blast*
qed

lemma *primefact-variant*: *primefact ps n* ⟷ *foldr op ∗ ps 1* = *n* ∧ *list-all prime ps* **by** (*auto simp add*: *primefact-def list-all-iff*)

lemma *lucas-primefact*:
assumes *n*: *n* ≥ *2* **and** *an*: [*a*^(*n* − *1*) = *1*] (*mod n*)
and *psn*: *foldr op ∗ ps 1* = *n* − *1*
and *psp*: *list-all* (λ*p*. *prime p* ∧ ¬ [*a*^((*n* − *1*) *div p*) = *1*] (*mod n*)) *ps*
shows *prime n*
proof−
{**fix** *p* **assume** *p*: *prime p p dvd n* − *1* [*a* ^ ((*n* − *1*) *div p*) = *1*] (*mod n*)
from *psn psp* **have** *psn1*: *primefact ps* (*n* − *1*)
by (*auto simp add*: *list-all-iff primefact-variant*)
from *p*(*3*) *primefact-contains*[*OF psn1 p*(*1*,*2*)] *psp*
have *False* **by** (*induct ps*, *auto*)}
with *lucas*[*OF n an*] **show** *?thesis* **by** *blast*
qed

lemma *mod-le*: **assumes** *n*: *n* ≠ (*0*::*nat*) **shows** *m mod n* ≤ *m*
proof−
from *mod-div-equality*[*of m n*]
have ∃ *x*. *x* + *m mod n* = *m* **by** *blast*
then show *?thesis* **by** *auto*
qed

lemma *pocklington-primefact*:
assumes *n*: *n* ≥ *2* **and** *qrn*: *q*∗*r* = *n* − *1* **and** *nq2*: *n* ≤ *q*^*2*
and *arnb*: (*a*^*r*) *mod n* = *b* **and** *psq*: *foldr op ∗ ps 1* = *q*
and *bqn*: (*b*^*q*) *mod n* = *1*
and *psp*: *list-all* (λ*p*. *prime p* ∧ *coprime* ((*b*^(*q div p*)) *mod n* − *1*) *n*) *ps*
shows *prime n*
proof−
from *bqn psp qrn*
have *bqn*: *a* ^ (*n* − *1*) *mod n* = *1*
and *psp*: *list-all* (λ*p*. *prime p* ∧ *coprime* (*a*^(*r* ∗(*q div p*)) *mod n* − *1*) *n*) *ps*
unfolding *arnb*[*symmetric*] *power-mod*

```
    by (simp-all add: power-mult[symmetric] algebra-simps)
  from n  have n0: n > 0 by arith
  from mod-div-equality[of a^(n - 1) n]
    mod-less-divisor[OF n0, of a^(n - 1)]
  have an1: [a ^ (n - 1) = 1] (mod n)
    unfolding nat-mod bqn
    apply −
    apply (rule exI[where x=0])
    apply (rule exI[where x=a^(n - 1) div n])
    by (simp add: algebra-simps)
  {fix p assume p: prime p p dvd q
    from psp psq have pfpsq: primefact ps q
      by (auto simp add: primefact-variant list-all-iff)
    from psp primefact-contains[OF pfpsq p]
    have p': coprime (a ^ (r * (q div p)) mod n - 1) n
      by (simp add: list-all-iff)
    from prime-ge-2[OF p(1)] have p01: p ≠ 0 p ≠ 1 p =Suc(p - 1) by arith+
    from div-mult1-eq[of r q p] p(2)
    have eq1: r* (q div p) = (n - 1) div p
      unfolding qrn[symmetric] dvd-eq-mod-eq-0 by (simp add: mult-commute)
    have ath: ⋀a (b::nat). a <= b ⟹ a ≠ 0 ==> 1 <= a ∧ 1 <= b by arith
    from n0 have n00: n ≠ 0 by arith
    from mod-le[OF n00]
    have th10: a ^ ((n - 1) div p) mod n ≤ a ^ ((n - 1) div p) .
    {assume a ^ ((n - 1) div p) mod n = 0
      then obtain s where s: a ^ ((n - 1) div p) = n*s
        unfolding mod-eq-0-iff by blast
      hence eq0: (a^((n - 1) div p))^p = (n*s)^p by simp
      from qrn[symmetric] have qn1: q dvd n - 1 unfolding dvd-def by auto
      from dvd-trans[OF p(2) qn1] div-mod-equality'[of n - 1 p]
      have npp: (n - 1) div p * p = n - 1 by (simp add: dvd-eq-mod-eq-0)
      with eq0 have a^(n - 1) = (n*s)^p
        by (simp add: power-mult[symmetric])
      hence 1 = (n*s)^(Suc (p - 1)) mod n using bqn p01 by simp
      also have ... = 0 by (simp add: mult-assoc)
      finally have False by simp }
      then have th11: a ^ ((n - 1) div p) mod n ≠ 0 by auto
    have th1: [a ^ ((n - 1) div p) mod n = a ^ ((n - 1) div p)] (mod n)
      unfolding modeq-def by simp
    from cong-sub[OF th1 cong-refl[of 1]]  ath[OF th10 th11]
    have th: [a ^ ((n - 1) div p) mod n - 1 = a ^ ((n - 1) div p) - 1] (mod n)
      by blast
    from cong-coprime[OF th] p'[unfolded eq1]
    have coprime (a ^ ((n - 1) div p) - 1) n by (simp add: coprime-commute) }
  with pocklington[OF n qrn[symmetric] nq2 an1]
  show ?thesis by blast
qed

end
```

53 Poly-Deriv: Polynomials and Differentiation

theory *Poly-Deriv*
imports *Deriv Polynomial*
begin

53.1 Derivatives of univariate polynomials

definition
pderiv :: *'a*::*real-normed-field poly* ⇒ *'a poly* **where**
pderiv = *poly-rec 0* (λ*a p p'*. *p* + *pCons 0 p'*)

lemma *pderiv-0* [*simp*]: *pderiv 0* = *0*
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*
apply (*induct p*)
apply *simp*
apply (*simp add*: *pderiv-pCons*)
apply (*rule lemma-DERIV-subst*)
apply (*rule DERIV-add DERIV-mult DERIV-const DERIV-ident* | *assumption*)+
apply *simp*
done

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*) *Ring-and-Field.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

54 Product-plus: Additive group operations on product types

theory *Product-plus*
imports *Main*
begin

54.1 Operations

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

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

instance ..
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*

54.2 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*)

end

55 Product-Vector: Cartesian Products as Vector Spaces

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

55.1 Product is a real vector space

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

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

lemma *fst-scaleR* [*simp*]: *fst* (*scaleR r A*) = *scaleR r* (*fst A*)
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* × *'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

55.2 Product is a normed vector space

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

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

definition *sgn-prod-def*:
sgn (*x*::*'a* × *'b*) = *scaleR* (*inverse* (*norm x*)) *x*

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

instance proof
fix *r* :: *real* **and** *x y* :: *'a* × *'b*
show *0* ≤ *norm x*
unfolding *norm-prod-def* **by** *simp*
show *norm x* = *0* ⟷ *x* = *0*
unfolding *norm-prod-def*
by (*simp add*: *expand-prod-eq*)
show *norm* (*x* + *y*) ≤ *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: norm-scaleR 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)
qed

end
```

55.3 Product is an inner product space

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

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

lemma inner-Pair [simp]: inner (a, b) (c, d) = inner a c + inner b d
  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-left-distrib)
  show inner (scaleR r x) y = r * inner x y
    unfolding inner-prod-def
    by (simp add: inner-scaleR-left 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

55.4 Pair operations are linear and continuous

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*: $0 \leq x$ **and** *y*: $0 \leq y$
shows $sqrt\ (x + y) \leq sqrt\ x + sqrt\ y$
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* $(\lambda x.\ (f\ x,\ g\ x))$
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*: $\bigwedge x.\ norm\ (f\ x) \leq norm\ x * Kf$
using *f.pos-bounded* **by** *fast*
obtain *Kg* **where** $0 < Kg$ **and** *norm-g*: $\bigwedge x.\ norm\ (g\ x) \leq norm\ x * Kg$
using *g.pos-bounded* **by** *fast*
have $\forall x.\ norm\ (f\ x,\ g\ x) \leq norm\ x * (Kf + Kg)$
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 ∃K. ∀x. norm (f x, g x) ≤ norm x * K ..
qed
```

TODO: The next three proofs are nearly identical to each other. Is there a good way to factor out the common parts?

```
lemma LIMSEQ-Pair:
  assumes X ----> a and Y ----> b
  shows (λn. (X n, Y n)) ----> (a, b)
proof (rule LIMSEQ-I)
  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: ∀n≥M. norm (X n - a) < ?s
    using LIMSEQ-D [OF ‹X ----> a› ‹0 < ?s›] ..
  obtain N where N: ∀n≥N. norm (Y n - b) < ?s
    using LIMSEQ-D [OF ‹Y ----> b› ‹0 < ?s›] ..
  have ∀n≥max M N. norm ((X n, Y n) - (a, b)) < r
    using M N by (simp add: real-sqrt-sum-squares-less norm-Pair)
  then show ∃n0. ∀n≥n0. norm ((X n, Y n) - (a, b)) < r ..
qed

lemma Cauchy-Pair:
  assumes Cauchy X and Cauchy Y
  shows Cauchy (λn. (X n, Y n))
proof (rule 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. norm (X m - X n) < ?s
    using CauchyD [OF ‹Cauchy X› ‹0 < ?s›] ..
  obtain N where N: ∀m≥N. ∀n≥N. norm (Y m - Y n) < ?s
    using CauchyD [OF ‹Cauchy Y› ‹0 < ?s›] ..
  have ∀m≥max M N. ∀n≥max M N. norm ((X m, Y m) - (X n, Y n)) < r
    using M N by (simp add: real-sqrt-sum-squares-less norm-Pair)
  then show ∃n0. ∀m≥n0. ∀n≥n0. norm ((X m, Y m) - (X n, Y n)) < r ..
qed

lemma LIM-Pair:
  assumes f -- x --> a and g -- x --> b
  shows (λx. (f x, g x)) -- x --> (a, b)
proof (rule LIM-I)
  fix r :: real assume 0 < r
  then have 0 < r / sqrt 2 (is 0 < ?e)
    by (simp add: divide-pos-pos)
  obtain s where s: 0 < s
    ∀z. z ≠ x ∧ norm (z - x) < s ⟶ norm (f z - a) < ?e
    using LIM-D [OF ‹f -- x --> a› ‹0 < ?e›] by fast
```

```
  obtain t where t: 0 < t
    ∀z. z ≠ x ∧ norm (z − x) < t ⟶ norm (g z − b) < ?e
    using LIM-D [OF ‹g −− x −−> b› ‹0 < ?e›] by fast
  have 0 < min s t ∧
    (∀z. z ≠ x ∧ norm (z − x) < min s t ⟶ norm ((f z, g z) − (a, b)) < r)
    using s t by (simp add: real-sqrt-sum-squares-less norm-Pair)
  then show
    ∃s>0. ∀z. z ≠ x ∧ norm (z − x) < s ⟶ norm ((f z, g z) − (a, b)) < r ..
qed

lemma isCont-Pair [simp]:
  ⟦isCont f x; isCont g x⟧ ⟹ isCont (λx. (f x, g x)) x
  unfolding isCont-def by (rule LIM-Pair)
```

55.5 Product is a complete vector space

```
instance ∗ :: (banach, banach) banach
proof
  fix X :: nat ⇒ 'a × 'b assume Cauchy X
  have 1: (λn. fst (X n)) −−−−> lim (λn. fst (X n))
    using fst.Cauchy [OF ‹Cauchy X›]
    by (simp add: Cauchy-convergent-iff convergent-LIMSEQ-iff)
  have 2: (λn. snd (X n)) −−−−> lim (λn. snd (X n))
    using snd.Cauchy [OF ‹Cauchy X›]
    by (simp add: Cauchy-convergent-iff convergent-LIMSEQ-iff)
  have X −−−−> (lim (λn. fst (X n)), lim (λn. snd (X n)))
    using LIMSEQ-Pair [OF 1 2] by simp
  then show convergent X
    by (rule convergentI)
qed
```

55.6 Frechet derivatives involving pairs

```
lemma FDERIV-Pair:
  assumes f: FDERIV f x :> f' and g: FDERIV g x :> g'
  shows FDERIV (λx. (f x, g x)) x :> (λh. (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

56 Random: A HOL random engine

theory *Random*
imports *Code-Index*
begin

notation *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o→* *60*)

56.1 Auxiliary functions

definition *inc-shift* :: *index* ⇒ *index* ⇒ *index* **where**
inc-shift v k = (*if v* = *k then 1 else k* + *1*)

definition *minus-shift* :: *index* ⇒ *index* ⇒ *index* ⇒ *index* **where**
minus-shift r k l = (*if k* < *l then r* + *k* − *l else k* − *l*)

fun *log* :: *index* ⇒ *index* ⇒ *index* **where**
log b i = (*if b* ≤ *1* ∨ *i* < *b then 1 else 1* + *log b* (*i div b*))

56.2 Random seeds

types *seed* = *index* × *index*

primrec *next* :: *seed* ⇒ *index* × *seed* **where**
next (*v*, *w*) = (*let*
k = *v div 53668*;
v' = *minus-shift 2147483563* (*40014* ∗ (*v mod 53668*)) (*k* ∗ *12211*);
l = *w div 52774*;
w' = *minus-shift 2147483399* (*40692* ∗ (*w mod 52774*)) (*l* ∗ *3791*);
z = *minus-shift 2147483562 v'* (*w'* + *1*) + *1*
in (*z*, (*v'*, *w'*)))

lemma *next-not-0*:
fst (*next s*) ≠ *0*
by (*cases s*) (*auto simp add*: *minus-shift-def Let-def*)

primrec *seed-invariant* :: *seed* ⇒ *bool* **where**
seed-invariant (*v*, *w*) ⟷ *0* < *v* ∧ *v* < *9438322952* ∧ *0* < *w* ∧ *True*

lemma *if-same*: (*if b then f x else f y*) = *f* (*if b then x else y*)
by (*cases b*) *simp-all*

definition *split-seed* :: *seed* ⇒ *seed* × *seed* **where**
split-seed s = (*let*
(*v*, *w*) = *s*;
(*v'*, *w'*) = *snd* (*next s*);
v'' = *inc-shift 2147483562 v*;
s'' = (*v''*, *w'*);
w'' = *inc-shift 2147483398 w*;
s''' = (*v'*, *w''*)
in (*s''*, *s'''*))

56.3 Base selectors

fun *iterate* :: *index* ⇒ ('*b* ⇒ '*a* ⇒ '*b* × '*a*) ⇒ '*b* ⇒ '*a* ⇒ '*b* × '*a* **where**
iterate k f x = (*if k* = *0 then Pair x else f x* ⟶ *iterate* (*k* − *1*) *f*)

definition *range* :: *index* ⇒ *seed* ⇒ *index* × *seed* **where**
range k = *iterate* (*log 2147483561 k*)
(λ*l*. *next* ⟶ (λ*v*. *Pair* (*v* + *l* ∗ *2147483561*))) *1*
⟶ (λ*v*. *Pair* (*v mod k*))

lemma *range*:
k > *0* ⟹ *fst* (*range k s*) < *k*
by (*simp add*: *range-def scomp-apply split-def del*: *log.simps iterate.simps*)

definition *select* :: '*a list* ⇒ *seed* ⇒ '*a* × *seed* **where**
select xs = *range* (*Code-Index.of-nat* (*length xs*))
⟶ (λ*k*. *Pair* (*nth xs* (*Code-Index.nat-of k*)))

lemma *select*:
assumes *xs* ≠ []
shows *fst* (*select xs s*) ∈ *set xs*
proof −
from *assms* **have** *Code-Index.of-nat* (*length xs*) > *0* **by** *simp*
with *range* **have**
fst (*range* (*Code-Index.of-nat* (*length xs*)) *s*) < *Code-Index.of-nat* (*length xs*)
by *best*
then have
Code-Index.nat-of (*fst* (*range* (*Code-Index.of-nat* (*length xs*)) *s*)) < *length xs*
by *simp*
then show *?thesis*
by (*simp add*: *scomp-apply split-beta select-def*)
qed

definition *select-default* :: *index* ⇒ '*a* ⇒ '*a* ⇒ *seed* ⇒ '*a* × *seed* **where**
[*code del*]: *select-default k x y* = *range k*
⟶ (λ*l*. *Pair* (*if l* + *1* < *k then x else y*))

lemma *select-default-zero*:
fst (*select-default 0 x y s*) = *y*
by (*simp add*: *scomp-apply split-beta select-default-def*)

lemma *select-default-code* [*code*]:
select-default k x y = (*if k* = *0*
then range 1 o→ (λ-. *Pair y*)
else range k o→ (λ*l*. *Pair* (*if l* + *1* < *k then x else y*)))
proof
fix *s*
have *snd* (*range* (*Code-Index.of-nat 0*) *s*) = *snd* (*range* (*Code-Index.of-nat 1*) *s*)
by (*simp add*: *range-def scomp-Pair scomp-apply split-beta*)
then show *select-default k x y s* = (*if k* = *0*
then range 1 o→ (λ-. *Pair y*)
else range k o→ (λ*l*. *Pair* (*if l* + *1* < *k then x else y*))) *s*
by (*cases k* = *0*) (*simp-all add*: *select-default-def scomp-apply split-beta*)
qed

56.4 *ML* interface

ML ⟪
structure Random-Engine =
struct

type seed = *int* ∗ *int*;

local

val seed = *ref*
(*let*
val now = *Time.toMilliseconds* (*Time.now* ());
val (*q*, *s1*) = *IntInf.divMod* (*now*, *2147483562*);
val s2 = *q mod 2147483398*;
in (*s1* + *1*, *s2* + *1*) *end*);

in

fun run f =
let
val (*x*, *seed*′) = *f* (! *seed*);
val - = *seed* := *seed*′
in x end;

end;

end;
⟫

no-notation *fcomp* (**infixl** *o>* *60*)
no-notation *scomp* (**infixl** *o→* *60*)

end

57 Quickcheck: A simple counterexample generator

theory *Quickcheck*
imports *Random Code-Eval Map*
begin

57.1 The *random* class

class *random* = *typerep* +
  fixes *random* :: *index* ⇒ *seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *seed*

    Type *'a itself*

instantiation *itself* :: ({*type*, *typerep*}) *random*
begin

definition
  random - = *Pair* (*TYPE*(*'a*), λ*u*. *Code-Eval.Const* (*STR ''TYPE''*) *TYPE-REP*(*'a*))

instance ..

end

57.2 Quickcheck generator

ML ⟪
structure StateMonad =
struct

fun liftT T sT = *sT* −−> *HOLogic.mk-prodT* (*T*, *sT*);
fun liftT' sT = *sT* −−> *sT*;

fun return T sT x = *Const* (@{*const-name Pair*}, *T* −−> *liftT T sT*) $ *x*;

fun scomp T1 T2 sT f g = *Const* (@{*const-name scomp*},
  liftT T1 sT −−> (*T1* −−> *liftT T2 sT*) −−> *liftT T2 sT*) $ *f* $ *g*;

end;

structure Quickcheck =

```
struct

open Quickcheck;

val eval-ref : (unit -> int -> int * int -> term list option * (int * int)) option
ref = ref NONE;

fun mk-generator-expr thy prop tys =
  let
    val bound-max = length tys - 1;
    val bounds = map-index (fn (i, ty) =>
      (2 * (bound-max - i) + 1, 2 * (bound-max - i), 2 * i, ty)) tys;
    val result = list-comb (prop, map (fn (i, -, -, -) => Bound i) bounds);
    val terms = HOLogic.mk-list @{typ term} (map (fn (-, i, -, -) => Bound i $
@{term ()}) bounds);
    val check = @{term If :: bool ⇒ term list option ⇒ term list option ⇒ term
list option}
      $ result $ @{term None :: term list option} $ (@{term Some :: term list ⇒
term list option } $ terms);
     val return = @{term Pair :: term list option ⇒ seed ⇒ term list option ×
seed};
    fun mk-termtyp ty = HOLogic.mk-prodT (ty, @{typ unit ⇒ term});
    fun mk-split ty = Sign.mk-const thy
     (@{const-name split}, [ty, @{typ unit ⇒ term}, StateMonad.liftT @{typ term
list option} @{typ seed}]);
    fun mk-scomp-split ty t t' =
       StateMonad.scomp (mk-termtyp ty) @{typ term list option} @{typ seed} t
(*FIXME*)
        (mk-split ty $ Abs (, ty, Abs (, @{typ unit ⇒ term}, t')));
    fun mk-bindclause (-, -, i, ty) = mk-scomp-split ty
      (Sign.mk-const thy (@{const-name random}, [ty]) $ Bound i)
    val t = fold-rev mk-bindclause bounds (return $ check);
  in Abs (n, @{typ index}, t) end;

fun compile-generator-expr thy t =
  let
    val tys = (map snd o fst o strip-abs) t;
    val t' = mk-generator-expr thy t tys;
    val f = Code-ML.eval-term (Quickcheck.eval-ref, eval-ref) thy t' [];
   in f #> Random-Engine.run #> (Option.map o map) (Code.postprocess-term
thy) end;

end
⟩⟩

setup ⟨⟨
  Quickcheck.add-generator (code, Quickcheck.compile-generator-expr o ProofCon-
text.theory-of)
⟩⟩
```

end

58 Quicksort: Quicksort

theory *Quicksort*
imports *Main Multiset*
begin

context *linorder*
begin

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

lemma *quicksort-permutes* [*simp*]:
multiset-of (*quicksort xs*) = *multiset-of xs*
by (*induct xs rule*: *quicksort.induct*) (*auto simp*: *union-ac*)

lemma *set-quicksort* [*simp*]: *set* (*quicksort xs*) = *set xs*
by(*simp add*: *set-count-greater-0*)

lemma *sorted-quicksort*: *sorted*(*quicksort xs*)
apply (*induct xs rule*: *quicksort.induct*)
apply *simp*
apply (*simp add*:*sorted-Cons sorted-append not-le less-imp-le*)
apply (*metis leD le-cases le-less-trans*)
done

end

end

59 Quotient: Quotient types

theory *Quotient*
imports *Main*
begin

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

59.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations ∼ :: *'a* => *'a* => *bool*.

class *eqv* =
  fixes *eqv* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** ∼ *50*)

class *equiv* = *eqv* +
  assumes *equiv-refl* [*intro*]: *x* ∼ *x*
  assumes *equiv-trans* [*trans*]: *x* ∼ *y* ⟹ *y* ∼ *z* ⟹ *x* ∼ *z*
  assumes *equiv-sym* [*sym*]: *x* ∼ *y* ⟹ *y* ∼ *x*

lemma *equiv-not-sym* [*sym*]: ¬ (*x* ∼ *y*) ==> ¬ (*y* ∼ (*x*::*'a*::*equiv*))
proof −
  assume ¬ (*x* ∼ *y*) **then show** ¬ (*y* ∼ *x*)
    by (*rule contrapos-nn*) (*rule equiv-sym*)
qed

lemma *not-equiv-trans1* [*trans*]: ¬ (*x* ∼ *y*) ==> *y* ∼ *z* ==> ¬ (*x* ∼ (*z*::*'a*::*equiv*))
proof −
  assume ¬ (*x* ∼ *y*) **and** *y* ∼ *z*
  show ¬ (*x* ∼ *z*)
  proof
    assume *x* ∼ *z*
    also from ‹*y* ∼ *z*› **have** *z* ∼ *y* **..**
    finally have *x* ∼ *y* **.**
    with ‹¬ (*x* ∼ *y*)› **show** *False* **by** *contradiction*
  qed
qed

lemma *not-equiv-trans2* [*trans*]: *x* ∼ *y* ==> ¬ (*y* ∼ *z*) ==> ¬ (*x* ∼ (*z*::*'a*::*equiv*))
proof −
  assume ¬ (*y* ∼ *z*) **then have** ¬ (*z* ∼ *y*) **..**
  also assume *x* ∼ *y* **then have** *y* ∼ *x* **..**
  finally have ¬ (*z* ∼ *x*) **. then show** (¬ *x* ∼ *z*) **..**
qed

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

typedef *'a quot* = {{*x*. *a* ∼ *x*} | *a*::*'a*::*eqv*. *True*}
  by *blast*

lemma *quotI* [*intro*]: {*x*. *a* ∼ *x*} ∈ *quot*
  unfolding *quot-def* **by** *blast*

lemma *quotE* [*elim*]: *R* ∈ *quot* ==> (!!*a*. *R* = {*x*. *a* ∼ *x*} ==> *C*) ==> *C*
  unfolding *quot-def* **by** *blast*

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

definition
  class :: *'a*::*equiv* => *'a quot* (⌊-⌋) **where**

$\lfloor a \rfloor = \mathit{Abs\text{-}quot}\ \{x.\ a \sim x\}$

theorem *quot-exhaust*: $\exists\, a.\ A = \lfloor a \rfloor$
proof (*cases A*)
fix *R* **assume** *R*: $A = \mathit{Abs\text{-}quot}\ R$
assume $R \in \mathit{quot}$ **then have** $\exists\, a.\ R = \{x.\ a \sim x\}$ **by** *blast*
with *R* **have** $\exists\, a.\ A = \mathit{Abs\text{-}quot}\ \{x.\ a \sim x\}$ **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*

59.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $(\lfloor a \rfloor = \lfloor b \rfloor) = (a \sim b)$
proof
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

59.3 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 $!!x\ x'\ y\ y'.\ \lfloor x \rfloor = \lfloor x' \rfloor \Longrightarrow \lfloor y \rfloor = \lfloor y' \rfloor \Longrightarrow g\ x\ y = g\ x'\ y'$
shows $f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
using *assms* **and** *TrueI*
by (*rule quot-cond-function*)

theorem *quot-function'*:
$(!!X\ Y.\ f\ X\ Y == g\ (pick\ X)\ (pick\ Y)) \Longrightarrow$
$(!!x\ x'\ y\ y'.\ x \sim x' \Longrightarrow y \sim y' \Longrightarrow g\ x\ y = g\ x'\ y') \Longrightarrow$
$f\ \lfloor a \rfloor\ \lfloor b \rfloor = g\ a\ b$
by (*rule quot-function*) (*simp-all only*: *quot-equality*)

end

60 Ramsey: Ramsey's Theorem

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

60.1 Preliminaries

60.1.1 "Axiom" of Dependent Choice

consts *choice* :: $('a \Rightarrow bool) \Rightarrow ('a * 'a)\ set \Rightarrow nat \Rightarrow 'a$
— An integer-indexed chain of choices
primrec
choice-0: $choice\ P\ r\ 0 = (SOME\ x.\ P\ x)$

choice-Suc: $choice\ P\ r\ (Suc\ n) = (SOME\ y.\ P\ y\ \&\ (choice\ P\ r\ n,\ y) \in r)$

lemma *choice-n*:
assumes *P0*: *P x0*
and *Pstep*: $!!x.\ P\ x \Longrightarrow \exists y.\ P\ y\ \&\ (x,y) \in r$
shows $P\ (choice\ P\ r\ n)$
proof (*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*: $!!x.\ P\ x \Longrightarrow \exists y.\ P\ y\ \&\ (x,y) \in r$
obtains $f :: nat \Rightarrow 'a$ **where**
$!!n.\ P\ (f\ n)$ **and** $!!n\ m.\ n < m \Longrightarrow (f\ n,\ f\ m) \in r$
proof

fix *n*
show *P* (*choice P r n*) **by** (*blast intro*: *choice-n* [*OF P0 Pstep*])
next
have *PSuc*: ∀ *n*. (*choice P r n*, *choice P r* (*Suc n*)) ∈ *r*
using *Pstep* [*OF choice-n* [*OF P0 Pstep*]]
by (*auto intro*: *someI2-ex*)
fix *n m* :: *nat*
assume *less*: *n* < *m*
show (*choice P r n*, *choice P r m*) ∈ *r* **using** *PSuc*
by (*auto intro*: *less-Suc-induct* [*OF less*] *transD* [*OF trans*])
qed

60.1.2 Partitions of a Set

definition
part :: *nat* => *nat* => *'a set* => (*'a set* => *nat*) => *bool*
— the function *f* partitions the *r*-subsets of the typically infinite set *Y* into *s* distinct categories.
where
part r s Y f = (∀ *X*. *X* ⊆ *Y* & *finite X* & *card X* = *r* --> *f X* < *s*)

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

lemma *part-Suc-imp-part*:
[| *infinite Y*; *part* (*Suc r*) *s Y f*; *y* ∈ *Y* |]
==> *part r s* (*Y* − {*y*}) (%*u*. *f* (*insert y u*))
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*

60.2 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes *s* **and** *r*::*nat*
shows
!!(*YY*::*'a set*) (*f*::*'a set* => *nat*).
[|*infinite YY*; *part r s YY f*|]
==> ∃ *Y' t'*. *Y'* ⊆ *YY* & *infinite Y'* & *t'* < *s* &
(∀ *X*. *X* ⊆ *Y'* & *finite X* & *card X* = *r* --> *f X* = *t'*)
proof (*induct r*)
case *0*
thus *?case* **by** (*auto simp add*: *part-def card-eq-0-iff cong*: *conj-cong*)
next
case (*Suc r*)
show *?case*
proof −

from *Suc.prems infinite-imp-nonempty* **obtain** *yy* **where** *yy*: $yy \in YY$ **by** *blast*
let *?ramr* = $\{((y,Y,t),(y',Y',t')).\ y' \in Y\ \&\ Y' \subseteq Y\}$
let *?propr* = $\%(y,Y,t).$
$y \in YY\ \&\ y \notin Y\ \&\ Y \subseteq YY\ \&\ infinite\ Y\ \&\ t < s$
$\&\ (\forall X.\ X \subseteq Y\ \&\ finite\ X\ \&\ card\ X = r \dashrightarrow (f\ o\ insert\ y)\ X = t)$
have *infYY'*: *infinite* $(YY - \{yy\})$ **using** *Suc.prems* **by** *auto*
have *partf'*: *part r s* $(YY - \{yy\})$ $(f \circ insert\ yy)$
by (*simp add*: *o-def part-Suc-imp-part yy Suc.prems*)
have *transr*: *trans ?ramr* **by** (*force simp add*: *trans-def*)
from *Suc.hyps* [*OF infYY' partf'*]
obtain *Y0* **and** *t0*
where $Y0 \subseteq YY - \{yy\}$ *infinite Y0* $t0 < s$
$\forall X.\ X \subseteq Y0 \wedge finite\ X \wedge card\ X = r \longrightarrow (f \circ insert\ yy)\ X = t0$
by *blast*
with *yy* **have** *propr0*: *?propr*(*yy*,*Y0*,*t0*) **by** *blast*
have *proprstep*: $\bigwedge x.$ *?propr x* $\Longrightarrow \exists y.$ *?propr y* $\wedge$ $(x, y) \in$ *?ramr*
proof −
fix *x*
assume *px*: *?propr x* **thus** *?thesis x*
proof (*cases x*)
case (*fields yx Yx tx*)
then obtain *yx'* **where** *yx'*: $yx' \in Yx$ **using** *px*
by (*blast dest*: *infinite-imp-nonempty*)
have *infYx'*: *infinite* $(Yx - \{yx'\})$ **using** *fields px* **by** *auto*
with *fields px yx' Suc.prems*
have *partfx'*: *part r s* $(Yx - \{yx'\})$ $(f \circ insert\ yx')$
by (*simp add*: *o-def part-Suc-imp-part part-subset* [**where** *?YY*=*YY*])
from *Suc.hyps* [*OF infYx' partfx'*]
obtain *Y'* **and** *t'*
where *Y'*: $Y' \subseteq Yx - \{yx'\}$ *infinite Y'* $t' < s$
$\forall X.\ X \subseteq Y' \wedge finite\ X \wedge card\ X = r \longrightarrow (f \circ insert\ yx')\ X = t'$
by *blast*
show *?thesis*
proof
show *?propr* (yx',Y',t') & $(x, (yx',Y',t')) \in$ *?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* ==> $(g\ n, g\ m) \in$ *?ramr* **by** *blast*
let *?gy* = *fst o g*
let *?gt* = *snd o snd o g*
have *rangeg*: $\exists k.$ *range ?gt* $\subseteq \{..<k\}$
proof (*intro exI subsetI*)
fix *x*
assume $x \in$ *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
```

60.3 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [4].

definition
disj-wf :: $('a * 'a)set \Rightarrow bool$
where
disj-wf r = $(\exists T.\ \exists n{::}nat.\ (\forall i{<}n.\ wf(T\ i))\ \&\ r = (\bigcup i{<}n.\ T\ i))$

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

lemma *transition-idx-less*:
$[|i{<}j;\ (s\ j,\ s\ i) \in T\ k;\ k{<}n|] \Longrightarrow transition\text{-}idx\ s\ T\ \{i,j\} < n$
apply (*subgoal-tac transition-idx s T* $\{i,\ j\} \leq k$, *simp*)
apply (*simp add*: *transition-idx-def*, *blast intro*: *Least-le*)
done

lemma *transition-idx-in*:
$[|i{<}j;\ (s\ j,\ s\ i) \in T\ k|] \Longrightarrow (s\ j,\ s\ i) \in T\ (transition\text{-}idx\ s\ T\ \{i,j\})$
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*:
$disj\text{-}wf(r) = (\exists T.\ \exists n{::}nat.\ (\forall i{<}n.\ wf(T\ i))\ \&\ r \subseteq (\bigcup i{<}n.\ T\ i))$
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 $\exists s.\ \forall i.\ (s\ (Suc\ i),\ s\ i) \in r$
then obtain *s* **where** *sSuc*: $\forall i.\ (s\ (Suc\ i),\ s\ i) \in r$ **..**
have *s*: $!!i\ j.\ i < j \Longrightarrow (s\ j,\ s\ i) \in r$
proof −
fix *i* **and** *j*::*nat*
assume *less*: *i*<*j*
thus $(s\ j,\ s\ i) \in r$
proof (*rule less-Suc-induct*)
show $\bigwedge i.\ (s\ (Suc\ i),\ s\ i) \in r$ **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
```

61 Reflection: Generic reflection and reification

theory *Reflection*
imports *Main*
uses *reify-data.ML* (*reflection.ML*)
begin

setup ⟪ *Reify-Data.setup* ⟫

lemma *ext2*: $(\forall x.\ f\ x = g\ x) \Longrightarrow f = g$
  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

62 RBT: Red-Black Trees

This theory defines purely functional red-black trees which can be used as an efficient representation of finite maps.

62.1 Data type and invariant

The type $('k, 'v)$ *rbt* denotes red-black trees with keys of type $'k$ and values of type $'v$. To function properly, the key type must belong to the *linorder* class.

A value t of this type is a valid red-black tree if it satisfies the invariant *isrbt t*. This theory provides lemmas to prove that the invariant is satisfied throughout the computation.

The interpretation function *RBT.map-of* returns the partial map represented by a red-black tree:

$RBT.map\text{-}of :: ('a, 'b)\ rbt \Rightarrow 'a \rightharpoonup 'b$

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in $O(\log n)$.

62.2 Operations

Currently, the following operations are supported:

$Empty :: ('a, 'b)\ rbt$

Returns the empty tree. $O(1)$

$insrt :: 'a \Rightarrow 'b \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Updates the map at a given position. $O(\log n)$

$RBT.delete :: 'a \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Deletes a map entry at a given position. $O(\log n)$

$union :: ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt \Rightarrow ('a, 'b)\ rbt$

Forms the union of two trees, preferring entries from the first one.

$RBT.map :: ('a \Rightarrow 'b) \Rightarrow ('c, 'a)\ rbt \Rightarrow ('c, 'b)\ rbt$

Maps a function over the values of a map. $O(n)$

62.3 Invariant preservation

isrbt Empty (*Empty-isrbt*)
isrbt ?t $\Longrightarrow$ *isrbt* (*insrt ?k ?v ?t*) (*insrt-isrbt*)
isrbt ?t $\Longrightarrow$ *isrbt* (*RBT.delete ?k ?t*) (*delete-isrbt*)
isrbt ?lt $\Longrightarrow$ *isrbt* (*union ?lt ?rt*) (*union-isrbt*)
isrbt (*RBT.map ?f ?t*) = *isrbt ?t* (*map-isrbt*)

62.4 Map Semantics

map-of-Empty

RBT.map-of Empty = *Map.empty*

map-of-insert

isrbt ?t $\Longrightarrow$ *RBT.map-of* (*insrt ?k ?v ?t*) = *RBT.map-of ?t*(*?k* $\mapsto$ *?v*)

map-of-delete

isrbt ?t $\Longrightarrow$ *RBT.map-of* (*RBT.delete ?k ?t*) = *RBT.map-of ?t*$\restriction_{(-\ \{?k\})}$

map-of-union

$[\![$*isrbt ?s*; *st ?t*$]\!]$
$\Longrightarrow$ *RBT.map-of* (*union ?s ?t*) = *RBT.map-of ?s* ++ *RBT.map-of ?t*

map-of-map

RBT.map-of (*RBT.map ?f ?t*) = *Option.map ?f* $\circ$ *RBT.map-of ?t*

end

63 State-Monad: Combinator syntax for generic, open state monads (single threaded monads)

theory *State-Monad*
imports *Main*
begin

63.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threaded).

To enter from the Haskell world, http://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

63.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

"yielding" transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, "yielding" a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and α, β, γ, ... for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

notation *fcomp* (**infixl** *o>* *60*)
notation (*xsymbols*) *fcomp* (**infixl** *o>* *60*)
notation *scomp* (**infixl** *o–>* *60*)
notation (*xsymbols*) *scomp* (**infixl** *o→* *60*)

abbreviation (*input*)
return ≡ *Pair*

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

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

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *Pair* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.

- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units ("open monad").
- The type of states may change due to a transformation.

63.3 Monad laws

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

lemmas *monad-simp = Pair-scomp scomp-Pair id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

lemmas *monad-collapse = monad-simp fcomp-apply scomp-apply split-beta*

63.4 Syntax

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

nonterminals *do-expr*

syntax
-do :: *do-expr* ⇒ *'a*
(*do - done* [*12*] *12*)
-scomp :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*- <− -;// -* [*1000*, *13*, *12*] *12*)
-fcomp :: *'a* ⇒ *do-expr* ⇒ *do-expr*
(*-;// -* [*13*, *12*] *12*)
-let :: *pttrn* ⇒ *'a* ⇒ *do-expr* ⇒ *do-expr*
(*let - = -;// -* [*1000*, *13*, *12*] *12*)
-done :: *'a* ⇒ *do-expr*
(*-* [*12*] *12*)

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

translations
-do f => f
-scomp x f g => f o→ (λ*x*. *g*)
-fcomp f g => f o> g
-let x t f => CONST Let t (λ*x*. *f*)
-done f => f

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 (-scomp, dummyT) $ v $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax fcomp}, -) $ f $ g) =
        Const (-fcomp, dummyT) $ f $ unfold-monad g
    | unfold-monad (Const (@{const-syntax Let}, -) $ f $ g) =
        let
          val (v, g') = dest-abs-eta g;
        in Const (-let, dummyT) $ v $ f $ unfold-monad g' end
    | unfold-monad (Const (@{const-syntax Pair}, -) $ f) =
        Const (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 (-do, dummyT) $ unfold-monad (Const (@{const-syntax scomp}, dum-
myT) $ f $ g), ts);
  fun fcomp-monad-tr' (f::g::ts) = if contains-scomp g then list-comb
        (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 (-do, dummyT) $ unfold-monad (Const (@{const-syntax Let}, dum-
myT) $ 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/ex/Random.thy.

end

64 Topology-Euclidean-Space: Elementary topology in Euclidean space.

theory *Topology-Euclidean-Space*
imports *SEQ Euclidean-Space*
begin

declare *fstcart-pastecart*[*simp*] *sndcart-pastecart*[*simp*]

64.1 General notion of a topology

definition *istopology* $L \longleftrightarrow \{\} \in L \wedge (\forall S \in L.\ \forall T \in L.\ S \cap T \in L) \wedge (\forall K.\ K \subseteq L \longrightarrow \bigcup K \in L)$
typedef (**open**) *'a topology* = $\{L::('a\ set)\ set.\ istopology\ L\}$
morphisms *openin topology*
unfolding *istopology-def* **by** *blast*

lemma *istopology-open-in*[*intro*]: *istopology*(*openin U*)
using *openin*[*of U*] **by** *blast*

lemma *topology-inverse'*: *istopology U* $\Longrightarrow$ *openin* (*topology U*) = *U*
using *topology-inverse*[*unfolded mem-def Collect-def*] **.**

lemma *topology-inverse-iff*: *istopology U* $\longleftrightarrow$ *openin* (*topology U*) = *U*
using *topology-inverse*[*of U*] *istopology-open-in*[*of topology U*] **by** *auto*

lemma *topology-eq*: *T1* = *T2* $\longleftrightarrow$ ($\forall S.$ *openin T1 S* $\longleftrightarrow$ *openin T2 S*)
proof−
{assume *T1*=*T2* **hence** $\forall S.$ *openin T1 S* $\longleftrightarrow$ *openin T2 S* **by** *simp***}**
moreover
{assume *H*: $\forall S.$ *openin T1 S* $\longleftrightarrow$ *openin T2 S*
hence *openin T1* = *openin T2* **by** (*metis mem-def set-ext*)
hence *topology* (*openin T1*) = *topology* (*openin T2*) **by** *simp*
hence *T1* = *T2* **unfolding** *openin-inverse* **.}**
ultimately show *?thesis* **by** *blast*
qed

Infer the "universe" from union of all sets in the topology.

definition *topspace T* = $\bigcup\{S.\ openin\ T\ S\}$

64.2 Main properties of open sets

lemma *openin-clauses*:
fixes *U* :: *'a topology*
shows *openin U* {}
$\bigwedge S\ T.$ *openin U S* $\Longrightarrow$ *openin U T* $\Longrightarrow$ *openin U* (*S*∩*T*)
$\bigwedge K.$ ($\forall S \in K.$ *openin U S*) $\Longrightarrow$ *openin U* ($\bigcup K$)
using *openin*[*of U*] **unfolding** *istopology-def Collect-def mem-def*
by (*metis mem-def subset-eq*)+

lemma *openin-subset*[*intro*]: *openin U S* $\Longrightarrow$ *S* $\subseteq$ *topspace U*
unfolding *topspace-def* **by** *blast*
lemma *openin-empty*[*simp*]: *openin U* {} **by** (*simp add*: *openin-clauses*)

lemma *openin-Int*[*intro*]: *openin U S* $\Longrightarrow$ *openin U T* $\Longrightarrow$ *openin U* (*S* $\cap$ *T*)
by (*simp add*: *openin-clauses*)

lemma *openin-Union*[*intro*]: ($\forall$ *S* $\in$*K*. *openin U S*) $\Longrightarrow$ *openin U* ($\bigcup$ *K*) **by** (*simp add*: *openin-clauses*)

lemma *openin-Un*[*intro*]: *openin U S* $\Longrightarrow$ *openin U T* $\Longrightarrow$ *openin U* (*S* $\cup$ *T*)
using *openin-Union*[*of* {*S*,*T*} *U*] **by** *auto*

lemma *openin-topspace*[*intro*, *simp*]: *openin U* (*topspace U*) **by** (*simp add*: *openin-Union topspace-def*)

lemma *openin-subopen*: *openin U S* $\longleftrightarrow$ ($\forall$ *x* $\in$ *S*. $\exists$ *T*. *openin U T* $\wedge$ *x* $\in$ *T* $\wedge$ *T* $\subseteq$ *S*) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof−
{assume *?lhs* **then have** *?rhs* **by** *auto* **}**
moreover
{assume *H*: *?rhs*
then obtain *t* **where** *t*: $\forall$ *x*$\in$*S*. *openin U* (*t x*) $\wedge$ *x* $\in$ *t x* $\wedge$ *t x* $\subseteq$ *S*
unfolding *Ball-def ex-simps*(*6*)[*symmetric*] *choice-iff* **by** *blast*
from *t* **have** *th0*: $\forall$ *x*$\in$ *t*‘*S*. *openin U x* **by** *auto*
have $\bigcup$ *t*‘*S* = *S* **using** *t* **by** *auto*
with *openin-Union*[*OF th0*] **have** *openin U S* **by** *simp* **}**
ultimately show *?thesis* **by** *blast*
qed

64.3 Closed sets

definition *closedin U S* $\longleftrightarrow$ *S* $\subseteq$ *topspace U* $\wedge$ *openin U* (*topspace U* − *S*)

lemma *closedin-subset*: *closedin U S* $\Longrightarrow$ *S* $\subseteq$ *topspace U* **by** (*metis closedin-def*)
lemma *closedin-empty*[*simp*]: *closedin U* {} **by** (*simp add*: *closedin-def*)
lemma *closedin-topspace*[*intro*,*simp*]:
closedin U (*topspace U*) **by** (*simp add*: *closedin-def*)
lemma *closedin-Un*[*intro*]: *closedin U S* $\Longrightarrow$ *closedin U T* $\Longrightarrow$ *closedin U* (*S* $\cup$ *T*)
by (*auto simp add*: *Diff-Un closedin-def*)

lemma *Diff-Inter*[*intro*]: *A* − $\bigcap$ *S* = $\bigcup$ {*A* − *s*|*s*. *s*$\in$*S*} **by** *auto*
lemma *closedin-Inter*[*intro*]: **assumes** *Ke*: *K* $\neq$ {} **and** *Kc*: $\forall$ *S* $\in$*K*. *closedin U S*
shows *closedin U* ($\bigcap$ *K*) **using** *Ke Kc* **unfolding** *closedin-def Diff-Inter* **by** *auto*

lemma *closedin-Int*[*intro*]: *closedin U S* ⟹ *closedin U T* ⟹ *closedin U* (*S* ∩ *T*)
using *closedin-Inter*[*of* {*S*,*T*} *U*] **by** *auto*

lemma *Diff-Diff-Int*: *A* − (*A* − *B*) = *A* ∩ *B* **by** *blast*
lemma *openin-closedin-eq*: *openin U S* ⟷ *S* ⊆ *topspace U* ∧ *closedin U* (*topspace U* − *S*)
apply (*auto simp add*: *closedin-def*)
apply (*metis openin-subset subset-eq*)
apply (*auto simp add*: *Diff-Diff-Int*)
apply (*subgoal-tac topspace U* ∩ *S* = *S*)
by *auto*

lemma *openin-closedin*: *S* ⊆ *topspace U* ⟹ (*openin U S* ⟷ *closedin U* (*topspace U* − *S*))
by (*simp add*: *openin-closedin-eq*)

lemma *openin-diff*[*intro*]: **assumes** *oS*: *openin U S* **and** *cT*: *closedin U T* **shows** *openin U* (*S* − *T*)
proof−
have *S* − *T* = *S* ∩ (*topspace U* − *T*) **using** *openin-subset*[*of U S*] *oS cT*
by (*auto simp add*: *topspace-def openin-subset*)
then show *?thesis* **using** *oS cT* **by** (*auto simp add*: *closedin-def*)
qed

lemma *closedin-diff*[*intro*]: **assumes** *oS*: *closedin U S* **and** *cT*: *openin U T* **shows** *closedin U* (*S* − *T*)
proof−
have *S* − *T* = *S* ∩ (*topspace U* − *T*) **using** *closedin-subset*[*of U S*] *oS cT*
by (*auto simp add*: *topspace-def*)
then show *?thesis* **using** *oS cT* **by** (*auto simp add*: *openin-closedin-eq*)
qed

64.4 Subspace topology.

definition *subtopology U V* = *topology* {*S* ∩ *V* |*S*. *openin U S*}

lemma *istopology-subtopology*: *istopology* {*S* ∩ *V* |*S*. *openin U S*} (**is** *istopology ?L*)
proof−
have {} ∈ *?L* **by** *blast*
{fix *A B* **assume** *A*: *A* ∈ *?L* **and** *B*: *B* ∈ *?L*
from *A B* **obtain** *Sa* **and** *Sb* **where** *Sa*: *openin U Sa A* = *Sa* ∩ *V* **and** *Sb*: *openin U Sb B* = *Sb* ∩ *V* **by** *blast*
have *A*∩*B* = (*Sa* ∩ *Sb*) ∩ *V openin U* (*Sa* ∩ *Sb*) **using** *Sa Sb* **by** *blast*+
then have *A* ∩ *B* ∈ *?L* **by** *blast***}**
moreover
{fix *K* **assume** *K*: *K* ⊆ *?L*
have *th0*: *?L* = (λ*S*. *S* ∩ *V*) ‘ *openin U*

```
      apply (rule set-ext)
      apply (simp add: Ball-def image-iff)
      by (metis mem-def)
    from K[unfolded th0 subset-image-iff]
    obtain Sk where Sk: Sk ⊆ openin U K = (λS. S ∩ V) ` Sk by blast
    have ⋃K = (⋃Sk) ∩ V using Sk by auto
     moreover have openin U (⋃ Sk) using Sk by (auto simp add: subset-eq
mem-def)
    ultimately have ⋃K ∈ ?L by blast}
  ultimately show ?thesis unfolding istopology-def by blast
qed

lemma openin-subtopology:
  openin (subtopology U V) S ⟷ (∃ T. (openin U T) ∧ (S = T ∩ V))
  unfolding subtopology-def topology-inverse'[OF istopology-subtopology]
  by (auto simp add: Collect-def)

lemma topspace-subtopology: topspace(subtopology U V) = topspace U ∩ V
  by (auto simp add: topspace-def openin-subtopology)

lemma closedin-subtopology:
  closedin (subtopology U V) S ⟷ (∃T. closedin U T ∧ S = T ∩ V)
  unfolding closedin-def topspace-subtopology
  apply (simp add: openin-subtopology)
  apply (rule iffI)
  apply clarify
  apply (rule-tac x=topspace U - T in exI)
  by auto

lemma openin-subtopology-refl: openin (subtopology U V) V ⟷ V ⊆ topspace
U
  unfolding openin-subtopology
  apply (rule iffI, clarify)
  apply (frule openin-subset[of U]) apply blast
  apply (rule exI[where x=topspace U])
  by auto

lemma subtopology-superset: assumes UV: topspace U ⊆ V
  shows subtopology U V = U
proof−
  {fix S
    {fix T assume T: openin U T S = T ∩ V
      from T openin-subset[OF T(1)] UV have eq: S = T by blast
      have openin U S unfolding eq using T by blast}
    moreover
    {assume S: openin U S
      hence ∃ T. openin U T ∧ S = T ∩ V
        using openin-subset[OF S] UV by auto}
    ultimately have (∃ T. openin U T ∧ S = T ∩ V) ⟷ openin U S by blast}
```

then show *?thesis* **unfolding** *topology-eq openin-subtopology* **by** *blast*
qed

lemma *subtopology-topspace*[*simp*]: *subtopology U* (*topspace U*) = *U*
by (*simp add*: *subtopology-superset*)

lemma *subtopology-UNIV*[*simp*]: *subtopology U UNIV* = *U*
by (*simp add*: *subtopology-superset*)

64.5 The universal Euclidean versions are what we use most of the time

definition *open S* ⟷ (∀ *x* ∈ *S*. ∃ *e* >*0*. ∀ *x'*. *dist x' x* < *e* ⟶ *x'* ∈ *S*)
definition *closed S* ⟷ *open*(*UNIV* − *S*)
definition *euclidean* = *topology open*

lemma *open-empty*[*intro*,*simp*]: *open* {} **by** (*simp add*: *open-def*)
lemma *open-UNIV*[*intro*,*simp*]: *open UNIV*
by (*simp add*: *open-def*, *rule exI*[**where** *x*=*1*], *auto*)

lemma *open-inter*[*intro*]: **assumes** *S*: *open S* **and** *T*: *open T*
shows *open* (*S* ∩ *T*)
proof−
note *thS* = *S*[*unfolded open-def*, *rule-format*]
note *thT* = *T*[*unfolded open-def*, *rule-format*]
{fix *x* **assume** *x*: *x* ∈ *S*∩*T*
hence *xS*: *x* ∈ *S* **and** *xT*: *x* ∈ *T* **by** *simp-all*
from *thS*[*OF xS*] **obtain** *eS* **where** *eS*: *eS* > *0* ∀ *x'*. *dist x' x* < *eS* ⟶ *x'* ∈ *S* **by** *blast*
from *thT*[*OF xT*] **obtain** *eT* **where** *eT*: *eT* > *0* ∀ *x'*. *dist x' x* < *eT* ⟶ *x'* ∈ *T* **by** *blast*
from *real-lbound-gt-zero*[*OF eS*(*1*) *eT*(*1*)] **obtain** *e* **where** *e*: *e* > *0 e* < *eS e* < *eT* **by** *blast*
{ fix *x'* **assume** *d*: *dist x' x* < *e*
hence *dS*: *dist x' x* < *eS* **and** *dT*: *dist x' x* < *eT* **using** *e* **by** *arith*+
from *eS*(*2*)[*rule-format*, *OF dS*] *eT*(*2*)[*rule-format*, *OF dT*] **have** *x'* ∈ *S*∩*T* **by** *blast*}
hence ∃ *e* >*0*. ∀ *x'*. *dist x' x* < *e* ⟶ *x'* ∈ (*S*∩*T*) **using** *e* **by** *blast*}
then show *?thesis* **unfolding** *open-def* **by** *blast*
qed

lemma *open-Union*[*intro*]: (∀ *S*∈*K*. *open S*) ⟹ *open* (⋃ *K*)
by (*simp add*: *open-def*) *metis*

lemma *open-openin*: *open S* ⟷ *openin euclidean S*
unfolding *euclidean-def*
apply (*rule cong*[**where** *x*=*S* **and** *y*=*S*])
apply (*rule topology-inverse*[*symmetric*])

apply (*auto simp add*: *istopology-def*)
by (*auto simp add*: *mem-def subset-eq*)

lemma *topspace-euclidean*: *topspace euclidean* = *UNIV*
apply (*simp add*: *topspace-def*)
apply (*rule set-ext*)
by (*auto simp add*: *open-openin*[*symmetric*])

lemma *topspace-euclidean-subtopology*[*simp*]: *topspace* (*subtopology euclidean S*) = *S*
by (*simp add*: *topspace-euclidean topspace-subtopology*)

lemma *closed-closedin*: *closed S* ⟷ *closedin euclidean S*
by (*simp add*: *closed-def closedin-def topspace-euclidean open-openin*)

lemma *open-Un*[*intro*]: *open S* ⟹ *open T* ⟹ *open* (*S*∪*T*)
by (*auto simp add*: *open-openin*)

lemma *open-subopen*: *open S* ⟷ (∀ *x*∈*S*. ∃ *T*. *open T* ∧ *x* ∈ *T* ∧ *T* ⊆ *S*)
by (*simp add*: *open-openin openin-subopen*[*symmetric*])

lemma *closed-empty*[*intro*, *simp*]: *closed* {} **by** (*simp add*: *closed-closedin*)

lemma *closed-UNIV*[*simp*,*intro*]: *closed UNIV*
by (*simp add*: *closed-closedin topspace-euclidean*[*symmetric*])

lemma *closed-Un*[*intro*]: *closed S* ⟹ *closed T* ⟹ *closed* (*S*∪*T*)
by (*auto simp add*: *closed-closedin*)

lemma *closed-Int*[*intro*]: *closed S* ⟹ *closed T* ⟹ *closed* (*S*∩*T*)
by (*auto simp add*: *closed-closedin*)

lemma *closed-Inter*[*intro*]: **assumes** *H*: ∀ *S* ∈*K*. *closed S* **shows** *closed* (⋂ *K*)
using *H*
unfolding *closed-closedin*
apply (*cases K* = {})
apply (*simp add*: *closed-closedin*[*symmetric*])
apply (*rule closedin-Inter*, *auto*)
done

lemma *open-closed*: *open S* ⟷ *closed* (*UNIV* − *S*)
by (*simp add*: *open-openin closed-closedin topspace-euclidean openin-closedin-eq*)

lemma *closed-open*: *closed S* ⟷ *open*(*UNIV* − *S*)
by (*simp add*: *open-openin closed-closedin topspace-euclidean closedin-def*)

lemma *open-diff*[*intro*]: *open S* ⟹ *closed T* ⟹ *open* (*S* − *T*)
by (*auto simp add*: *open-openin closed-closedin*)

lemma *closed-diff*[*intro*]: *closed S* $\Longrightarrow$ *open T* $\Longrightarrow$ *closed*(*S*−*T*)
by (*auto simp add*: *open-openin closed-closedin*)

lemma *open-Inter*[*intro*]: **assumes** *fS*: *finite S* **and** *h*: $\forall$ *T*$\in$*S*. *open T* **shows** *open* ($\bigcap$ *S*)
using *h* **by** (*induct rule*: *finite-induct*[*OF fS*], *auto*)

lemma *closed-Union*[*intro*]: **assumes** *fS*: *finite S* **and** *h*: $\forall$ *T*$\in$*S*. *closed T* **shows** *closed* ($\bigcup$ *S*)
using *h* **by** (*induct rule*: *finite-induct*[*OF fS*], *auto*)

64.6 Open and closed balls.

definition *ball x e* = {*y*. *dist x y* < *e*}
definition *cball x e* = {*y*. *dist x y* $\leq$ *e*}

lemma *mem-ball*[*simp*]: *y* $\in$ *ball x e* $\longleftrightarrow$ *dist x y* < *e* **by** (*simp add*: *ball-def*)
lemma *mem-cball*[*simp*]: *y* $\in$ *cball x e* $\longleftrightarrow$ *dist x y* $\leq$ *e* **by** (*simp add*: *cball-def*)
lemma *mem-ball-0*[*simp*]: *x* $\in$ *ball 0 e* $\longleftrightarrow$ *norm x* < *e* **by** (*simp add*: *dist-def*)
lemma *mem-cball-0*[*simp*]: *x* $\in$ *cball 0 e* $\longleftrightarrow$ *norm x* $\leq$ *e* **by** (*simp add*: *dist-def*)
lemma *centre-in-cball*[*simp*]: *x* $\in$ *cball x e* $\longleftrightarrow$ *0*$\leq$ *e* **by** *simp*
lemma *ball-subset-cball*[*simp*,*intro*]: *ball x e* $\subseteq$ *cball x e* **by** (*simp add*: *subset-eq*)
lemma *subset-ball*[*intro*]: *d* <= *e* ==> *ball x d* $\subseteq$ *ball x e* **by** (*simp add*: *subset-eq*)
lemma *subset-cball*[*intro*]: *d* <= *e* ==> *cball x d* $\subseteq$ *cball x e* **by** (*simp add*: *subset-eq*)
lemma *ball-max-Un*: *ball a* (*max r s*) = *ball a r* $\cup$ *ball a s*
by (*simp add*: *expand-set-eq*) *arith*

lemma *ball-min-Int*: *ball a* (*min r s*) = *ball a r* $\cap$ *ball a s*
by (*simp add*: *expand-set-eq*)

64.7 Topological properties of open balls

lemma *diff-less-iff*: (*a*::*real*) − *b* > *0* $\longleftrightarrow$ *a* > *b*
(*a*::*real*) − *b* < *0* $\longleftrightarrow$ *a* < *b*
a − *b* < *c* $\longleftrightarrow$ *a* < *c* +*b* *a* − *b* > *c* $\longleftrightarrow$ *a* > *c* +*b* **by** *arith*+
lemma *diff-le-iff*: (*a*::*real*) − *b* $\geq$ *0* $\longleftrightarrow$ *a* $\geq$ *b* (*a*::*real*) − *b* $\leq$ *0* $\longleftrightarrow$ *a* $\leq$ *b*
a − *b* $\leq$ *c* $\longleftrightarrow$ *a* $\leq$ *c* +*b* *a* − *b* $\geq$ *c* $\longleftrightarrow$ *a* $\geq$ *c* +*b* **by** *arith*+

lemma *open-ball*[*intro*, *simp*]: *open* (*ball x e*)
unfolding *open-def ball-def Collect-def Ball-def mem-def*
unfolding *dist-sym*
apply *clarify*
apply (*rule-tac x*=*e* − *dist xa x* **in** *exI*)
using *dist-triangle-alt*[**where** *z*=*x*]
apply (*clarsimp simp add*: *diff-less-iff*)
apply *atomize*
apply (*erule-tac x*=*x'* **in** *allE*)
apply (*erule-tac x*=*xa* **in** *allE*)
by *arith*

lemma *centre-in-ball*[*simp*]: $x \in \mathit{ball}\ x\ e \longleftrightarrow e > 0$ **by** (*metis mem-ball dist-refl*)
lemma *open-contains-ball*: *open* $S \longleftrightarrow (\forall x{\in}S.\ \exists e{>}0.\ \mathit{ball}\ x\ e \subseteq S)$
unfolding *open-def subset-eq mem-ball Ball-def dist-sym* **..**

lemma *open-contains-ball-eq*: *open* $S \Longrightarrow \forall x.\ x{\in}S \longleftrightarrow (\exists e{>}0.\ \mathit{ball}\ x\ e \subseteq S)$
by (*metis open-contains-ball subset-eq centre-in-ball*)

lemma *ball-eq-empty*[*simp*]: $\mathit{ball}\ x\ e = \{\} \longleftrightarrow e \leq 0$
unfolding *mem-ball expand-set-eq*
apply (*simp add*: *not-less*)
by (*metis dist-pos-le order-trans dist-refl*)

lemma *ball-empty*[*intro*]: $e \leq 0 \Longrightarrow \mathit{ball}\ x\ e = \{\}$ **by** *simp*

64.8 Basic "localization" results are handy for connectedness.

lemma *openin-open*: *openin* (*subtopology euclidean* U) $S \longleftrightarrow (\exists T.\ \mathit{open}\ T \wedge (S = U \cap T))$
by (*auto simp add*: *openin-subtopology open-openin*[*symmetric*])

lemma *openin-open-Int*[*intro*]: *open* $S \Longrightarrow$ *openin* (*subtopology euclidean* U) ($U \cap S$)
by (*auto simp add*: *openin-open*)

lemma *open-openin-trans*[*trans*]:
open $S \Longrightarrow$ *open* $T \Longrightarrow T \subseteq S \Longrightarrow$ *openin* (*subtopology euclidean* S) T
by (*metis Int-absorb1 openin-open-Int*)

lemma *open-subset*: $S \subseteq T \Longrightarrow$ *open* $S \Longrightarrow$ *openin* (*subtopology euclidean* T) S
by (*auto simp add*: *openin-open*)

lemma *closedin-closed*: *closedin* (*subtopology euclidean* U) $S \longleftrightarrow (\exists T.\ \mathit{closed}\ T \wedge S = U \cap T)$
by (*simp add*: *closedin-subtopology closed-closedin Int-ac*)

lemma *closedin-closed-Int*: *closed* S ==> *closedin* (*subtopology euclidean* U) ($U \cap S$)
by (*metis closedin-closed*)

lemma *closed-closedin-trans*: *closed* $S \Longrightarrow$ *closed* $T \Longrightarrow T \subseteq S \Longrightarrow$ *closedin* (*subtopology euclidean* S) T
apply (*subgoal-tac* $S \cap T = T$)
apply *auto*
apply (*frule closedin-closed-Int*[*of* T S])
by *simp*

lemma *closed-subset*: $S \subseteq T \Longrightarrow$ *closed* $S \Longrightarrow$ *closedin* (*subtopology euclidean* T)

S
by (*auto simp add*: *closedin-closed*)

lemma *openin-euclidean-subtopology-iff*: *openin* (*subtopology euclidean U*) *S*
⟷ *S* ⊆ *U* ∧ (∀ *x*∈*S*. ∃ *e*>*0*. ∀ *x'*∈*U*. *dist x' x* < *e* ⟶ *x'*∈ *S*) (**is** *?lhs* ⟷ *?rhs*)
proof−
{assume *?lhs* **hence** *?rhs* **unfolding** *openin-subtopology open-openin*[*symmetric*]
by (*simp add*: *open-def*) *blast***}**
moreover
{assume *SU*: *S* ⊆ *U* **and** *H*: ⋀*x*. *x* ∈ *S* ⟹ ∃ *e*>*0*. ∀ *x'*∈*U*. *dist x' x* < *e* ⟶ *x'* ∈ *S*
from *H* **obtain** *d* **where** *d*: ⋀*x* . *x*∈ *S* ⟹ *d x* > *0* ∧ (∀ *x'* ∈ *U*. *dist x' x* < *d x* ⟶ *x'* ∈ *S*)
by *metis*
let *?T* = ⋃{*B*. ∃ *x*∈*S*. *B* = *ball x* (*d x*)}
have *oT*: *open ?T* **by** *auto*
{ fix *x* **assume** *x*∈*S*
hence *x* ∈ ⋃{*B*. ∃ *x*∈*S*. *B* = *ball x* (*d x*)}
apply *simp* **apply**(*rule-tac x*=*ball x*(*d x*) **in** *exI*) **apply** *auto*
unfolding *dist-refl* **using** *d*[*of x*] **by** *auto*
hence *x*∈ *?T* ∩ *U* **using** *SU* **and** ‹*x*∈*S*› **by** *auto* **}**
moreover
{ fix *y* **assume** *y*∈*?T*
then obtain *B* **where** *y*∈*B B*∈{*B*. ∃ *x*∈*S*. *B* = *ball x* (*d x*)} **by** *auto*
then obtain *x* **where** *x*∈*S* **and** *x*:*y* ∈ *ball x* (*d x*) **by** *auto*
assume *y*∈*U*
hence *y*∈*S* **using** *d*[*OF* ‹*x*∈*S*›] **and** *x* **by**(*auto simp add*: *dist-sym*) **}**
ultimately have *S* = *?T* ∩ *U* **by** *blast*
with *oT* **have** *?lhs* **unfolding** *openin-subtopology open-openin*[*symmetric*] **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

These "transitivity" results are handy too.

lemma *openin-trans*[*trans*]: *openin* (*subtopology euclidean T*) *S* ⟹ *openin* (*subtopology euclidean U*) *T*
⟹ *openin* (*subtopology euclidean U*) *S*
unfolding *open-openin openin-open* **by** *blast*

lemma *openin-open-trans*: *openin* (*subtopology euclidean T*) *S* ⟹ *open T* ⟹ *open S*
by (*auto simp add*: *openin-open intro*: *openin-trans*)

lemma *closedin-trans*[*trans*]:
closedin (*subtopology euclidean T*) *S* ⟹
closedin (*subtopology euclidean U*) *T*
==> *closedin* (*subtopology euclidean U*) *S*
by (*auto simp add*: *closedin-closed closed-closedin closed-Inter Int-assoc*)

lemma *closedin-closed-trans*: *closedin* (*subtopology euclidean T*) *S* ⟹ *closed T* ⟹ *closed S*
by (*auto simp add*: *closedin-closed intro*: *closedin-trans*)

64.9 Connectedness

definition *connected S* ⟷
~(∃ *e1 e2*. *open e1* ∧ *open e2* ∧ *S* ⊆ (*e1* ∪ *e2*) ∧ (*e1* ∩ *e2* ∩ *S* = {})
∧ ~(*e1* ∩ *S* = {}) ∧ ~(*e2* ∩ *S* = {}))

lemma *connected-local*:
connected S ⟷ ~(∃ *e1 e2*.
openin (*subtopology euclidean S*) *e1* ∧
openin (*subtopology euclidean S*) *e2* ∧
S ⊆ *e1* ∪ *e2* ∧
e1 ∩ *e2* = {} ∧
~(*e1* = {}) ∧
~(*e2* = {}))
unfolding *connected-def openin-open* **by** *blast*

lemma *exists-diff*: (∃ *S*. *P*(*UNIV* − *S*)) ⟷ (∃ *S*. *P S*) (**is** *?lhs* ⟷ *?rhs*)
proof−

{assume *?lhs* **hence** *?rhs* **by** *blast* **}**
moreover
{fix *S* **assume** *H*: *P S*
have *S* = *UNIV* − (*UNIV* − *S*) **by** *auto*
with *H* **have** *P* (*UNIV* − (*UNIV* − *S*)) **by** *metis* **}**
ultimately show *?thesis* **by** *metis*
qed

lemma *connected-clopen*: *connected S* ⟷
(∀ *T*. *openin* (*subtopology euclidean S*) *T* ∧
closedin (*subtopology euclidean S*) *T* ⟶ *T* = {} ∨ *T* = *S*) (**is** *?lhs* ⟷ *?rhs*)
proof−
have ¬ *connected S* ⟷ (∃ *e1 e2*. *open e1* ∧ *open* (*UNIV* − *e2*) ∧ *S* ⊆ *e1* ∪ (*UNIV* − *e2*) ∧ *e1* ∩ (*UNIV* − *e2*) ∩ *S* = {} ∧ *e1* ∩ *S* ≠ {} ∧ (*UNIV* − *e2*) ∩ *S* ≠ {})
unfolding *connected-def openin-open closedin-closed*
apply (*subst exists-diff*) **by** *blast*
hence *th0*: *connected S* ⟷ ¬ (∃ *e2 e1*. *closed e2* ∧ *open e1* ∧ *S* ⊆ *e1* ∪ (*UNIV* − *e2*) ∧ *e1* ∩ (*UNIV* − *e2*) ∩ *S* = {} ∧ *e1* ∩ *S* ≠ {} ∧ (*UNIV* − *e2*) ∩ *S* ≠ {})
(**is** - ⟷ ¬ (∃ *e2 e1*. *?P e2 e1*)) **apply** (*simp add*: *closed-def*) **by** *metis*

have *th1*: *?rhs* ⟷ ¬ (∃ *t′ t*. *closed t′*∧*t* = *S*∩*t′* ∧ *t*≠{} ∧ *t*≠*S* ∧ (∃ *t′*. *open t′* ∧ *t* = *S* ∩ *t′*))
(**is** - ⟷ ¬ (∃ *t′ t*. *?Q t′ t*))

unfolding *connected-def openin-open closedin-closed* **by** *auto*
{fix *e2*
{fix *e1* **have** *?P e2 e1* ⟷ (∃ *t*. *closed e2* ∧ *t* = *S*∩*e2* ∧ *open e1* ∧ *t* = *S*∩*e1* ∧ *t*≠{} ∧ *t*≠*S*)
by *auto***}**
then have (∃ *e1*. *?P e2 e1*) ⟷ (∃ *t*. *?Q e2 t*) **by** *metis***}**
then have ∀ *e2*. (∃ *e1*. *?P e2 e1*) ⟷ (∃ *t*. *?Q e2 t*) **by** *blast*
then show *?thesis* **unfolding** *th0 th1* **by** *simp*
qed

lemma *connected-empty*[*simp*, *intro*]: *connected* {}
by (*simp add*: *connected-def*)

64.10 Hausdorff and other separation properties

lemma *hausdorff*:
assumes *xy*: *x* ≠ *y*
shows ∃ *U V*. *open U* ∧ *open V* ∧ *x*∈ *U* ∧ *y* ∈ *V* ∧ (*U* ∩ *V* = {}) (**is** ∃ *U V*. *?P U V*)
proof−
let *?U* = *ball x* (*dist x y* / *2*)
let *?V* = *ball y* (*dist x y* / *2*)
have *th0*: ⋀*d x y z*. (*d x z* :: *real*) <= *d x y* + *d y z* ⟹ *d y z* = *d z y*
==> ~(*d x y* ∗ *2* < *d x z* ∧ *d z y* ∗ *2* < *d x z*) **by** *arith*
have *?P ?U ?V* **using** *dist-pos-lt*[*OF xy*] *th0*[*of dist*,*OF dist-triangle dist-sym*]
by (*auto simp add*: *dist-refl expand-set-eq less-divide-eq-number-of1*)
then show *?thesis* **by** *blast*
qed

lemma *separation-t2*: *x* ≠ *y* ⟷ (∃ *U V*. *open U* ∧ *open V* ∧ *x* ∈ *U* ∧ *y* ∈ *V* ∧ *U* ∩ *V* = {})
using *hausdorff*[*of x y*] **by** *blast*

lemma *separation-t1*: *x* ≠ *y* ⟷ (∃ *U V*. *open U* ∧ *open V* ∧ *x* ∈*U* ∧ *y*∉ *U* ∧ *x*∉*V* ∧ *y*∈*V*)
using *separation-t2*[*of x y*] **by** *blast*

lemma *separation-t0*: *x* ≠ *y* ⟷ (∃ *U*. *open U* ∧ ~(*x*∈*U* ⟷ *y*∈*U*)) **by**(*metis separation-t1*)

64.11 Limit points

definition *islimpt*:: *real* ^'*n*::*finite* ⇒ (*real*^'*n*) *set* ⇒ *bool* (**infixr** *islimpt 60*) **where**
islimpt-def: *x islimpt S* ⟷ (∀ *T*. *x*∈*T* ⟶ *open T* ⟶ (∃ *y*∈*S*. *y*∈*T* ∧ *y*≠*x*))

lemma *islimptE*: **assumes** *x islimpt S* **and** *x* ∈ *T* **and** *open T*
obtains (∃ *y*∈*S*. *y*∈*T* ∧ *y*≠*x*)
using *assms* **unfolding** *islimpt-def* **by** *auto*

lemma *islimpt-subset*: *x islimpt S* $\Longrightarrow$ *S* $\subseteq$ *T* ==> *x islimpt T* **by** (*auto simp add*: *islimpt-def*)
lemma *islimpt-approachable*: *x islimpt S* $\longleftrightarrow$ ($\forall$ *e*>*0*. $\exists$ *x'*$\in$*S*. *x'* $\neq$ *x* $\wedge$ *dist x' x* < *e*)
unfolding *islimpt-def*
apply *auto*
apply(*erule-tac x*=*ball x e* **in** *allE*)
apply (*auto simp add*: *dist-refl*)
apply(*rule-tac x*=*y* **in** *bexI*) **apply** (*auto simp add*: *dist-sym*)
by (*metis open-def dist-sym open-ball centre-in-ball mem-ball*)

lemma *islimpt-approachable-le*: *x islimpt S* $\longleftrightarrow$ ($\forall$ *e*>*0*. $\exists$ *x'*$\in$ *S*. *x'* $\neq$ *x* $\wedge$ *dist x' x* <= *e*)
unfolding *islimpt-approachable*
using *approachable-lt-le*[**where** *f*=λ*x'*. *dist x' x* **and** *P*=λ*x'*. $\neg$ (*x'*$\in$*S* $\wedge$ *x'*$\neq$*x*)]
by *metis*

lemma *islimpt-UNIV*[*simp*, *intro*]: (*x*:: *real* ^'*n*::*finite*) *islimpt UNIV*
proof−
{
fix *e*::*real* **assume** *ep*: *e*>*0*
from *vector-choose-size*[*of e*/*2*] *ep* **have** $\exists$(*c*:: *real* ^'*n*). *norm c* = *e*/*2* **by** *auto*
then obtain *c* ::*real*^'*n* **where** *c*: *norm c* = *e*/*2* **by** *blast*
let *?x* = *x* + *c*
have *?x* $\neq$ *x* **using** *c ep* **by** (*auto simp add*: *norm-eq-0-imp*)
moreover have *dist ?x x* < *e* **using** *c ep* **apply** *simp* **by** *norm*
ultimately have $\exists$ *x'*. *x'* $\neq$ *x*$\wedge$ *dist x' x* < *e* **by** *blast***}**
then show *?thesis* **unfolding** *islimpt-approachable* **by** *blast*
qed

lemma *closed-limpt*: *closed S* $\longleftrightarrow$ ($\forall$ *x*. *x islimpt S* $\longrightarrow$ *x* $\in$ *S*)
unfolding *closed-def*
apply (*subst open-subopen*)
apply (*simp add*: *islimpt-def subset-eq*)
by (*metis DiffE DiffI UNIV-I insertCI insert-absorb mem-def*)

lemma *islimpt-EMPTY*[*simp*]: $\neg$ *x islimpt* {}
unfolding *islimpt-approachable* **apply** *auto* **by** *ferrack*

lemma *closed-positive-orthant*: *closed* {*x*::*real*^'*n*::*finite*. $\forall$ *i*. *0* <=*x*$*i*}
proof−
let *?U* = *UNIV* :: '*n set*
let *?O* = {*x*::*real*^'*n*. $\forall$ *i*. *x*$*i*>=*0*}
{fix *x*:: *real*^'*n* **and** *i*::'*n* **assume** *H*: $\forall$ *e*>*0*. $\exists$ *x'*$\in$*?O*. *x'* $\neq$ *x* $\wedge$ *dist x' x* < *e*
and *xi*: *x*$*i* < *0*
from *xi* **have** *th0*: −*x*$*i* > *0* **by** *arith*
from *H*[*rule-format*, *OF th0*] **obtain** *x'* **where** *x'*: *x'* $\in$*?O x'* $\neq$ *x dist x' x* < −*x* $ *i* **by** *blast*

have *th*: $\bigwedge b\ a\ (x::real).\ abs\ x <= b \Longrightarrow b <= a ==> \sim(a + x < 0)$ **by** *arith*
have *th'*: $\bigwedge x\ (y::real).\ x < 0 \Longrightarrow 0 <= y ==> abs\ x <= abs\ (y - x)$ **by** *arith*
have *th1*: $|x\$i| \leq |(x' - x)\$i|$ **using** $x'(1)$ *xi*
apply (*simp only*: *vector-component*)
by (*rule th'*) *auto*
have *th2*: $|dist\ x\ x'| \geq |(x' - x)\$i|$ **using** *component-le-norm*[*of* $x'-x$ *i*]
apply (*simp add*: *dist-def*) **by** *norm*
from *th*[*OF th1 th2*] $x'(3)$ **have** *False* **by** (*simp add*: *dist-sym dist-pos-le*) **}**
then show *?thesis* **unfolding** *closed-limpt islimpt-approachable*
unfolding *not-le*[*symmetric*] **by** *blast*
qed

lemma *finite-set-avoid*: **assumes** *fS*: *finite S* **shows** $\exists d>0.\ \forall x\in S.\ x \neq a \longrightarrow d <= dist\ a\ x$
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **apply** *auto* **by** *ferrack*
next
case (*2 x F*)
from *2* **obtain** *d* **where** *d*: $d>0\ \forall x\in F.\ x\neq a \longrightarrow d \leq dist\ a\ x$ **by** *blast*
{assume $x = a$ **hence** *?case* **using** *d* **by** *auto* **}**
moreover
{assume *xa*: $x\neq a$
let *?d* = *min d* (*dist a x*)
have *dp*: *?d* > *0* **using** *xa d*(*1*) **using** *dist-nz* **by** *auto*
from *d* **have** *d'*: $\forall x\in F.\ x\neq a \longrightarrow ?d \leq dist\ a\ x$ **by** *auto*
with *dp xa* **have** *?case* **by**(*auto intro*!: *exI*[**where** $x=?d$]) **}**
ultimately show *?case* **by** *blast*
qed

lemma *islimpt-finite*: **assumes** *fS*: *finite S* **shows** $\neg$ *a islimpt S*
unfolding *islimpt-approachable*
using *finite-set-avoid*[*OF fS*, *of a*] **by** (*metis dist-sym not-le*)

lemma *islimpt-Un*: *x islimpt* $(S \cup T) \longleftrightarrow$ *x islimpt S* $\vee$ *x islimpt T*
apply (*rule iffI*)
defer
apply (*metis Un-upper1 Un-upper2 islimpt-subset*)
unfolding *islimpt-approachable*
apply *auto*
apply (*erule-tac x=min e ea* **in** *allE*)
apply *auto*
done

lemma *discrete-imp-closed*:
assumes *e*: $0 < e$ **and** *d*: $\forall x \in S.\ \forall y \in S.\ norm(y - x) < e \longrightarrow y = x$
shows *closed S*
proof−

{fix x **assume** C: $\forall\, e{>}0.\ \exists\, x'{\in}S.\ x' \neq x \wedge dist\ x'\ x < e$
from e **have** $e2$: $e/2 > 0$ **by** *arith*
from C[*rule-format*, *OF e2*] **obtain** y **where** y: $y \in S\ y{\neq}x\ dist\ y\ x < e/2$ **by** *blast*
let *?m* = *min* $(e/2)$ $(dist\ x\ y)$
from $e2\ y(2)$ **have** *mp*: *?m* > *0* **by** (*simp add*: *dist-nz*[*THEN sym*])
from C[*rule-format*, *OF mp*] **obtain** z **where** z: $z \in S\ z{\neq}x\ dist\ z\ x < ?m$ **by** *blast*
have *th*: *norm* $(z - y) < e$ **using** $z\ y$ **by** *norm*
from d[*rule-format*, *OF* $y(1)\ z(1)$ *th*] $y\ z$
have *False* **by** (*auto simp add*: *dist-sym*)**}**
then show *?thesis* **by** (*metis islimpt-approachable closed-limpt*)
qed

64.12 Interior of a Set

definition *interior* $S = \{x.\ \exists\, T.\ open\ T \wedge x \in T \wedge T \subseteq S\}$

lemma *interior-eq*: *interior* $S = S \longleftrightarrow$ *open* S
apply (*simp add*: *expand-set-eq interior-def*)
apply (*subst* (*2*) *open-subopen*) **by** *blast*

lemma *interior-open*: *open* S ==> (*interior* $S = S$) **by** (*metis interior-eq*)

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

lemma *open-interior*[*simp*, *intro*]: *open*(*interior* S)
apply (*simp add*: *interior-def*)
apply (*subst open-subopen*) **by** *blast*

lemma *interior-interior*[*simp*]: *interior*(*interior* S) = *interior* S **by** (*metis interior-eq open-interior*)
lemma *interior-subset*: *interior* $S \subseteq S$ **by** (*auto simp add*: *interior-def*)
lemma *subset-interior*: $S \subseteq T$ ==> (*interior* S) $\subseteq$ (*interior* T) **by** (*auto simp add*: *interior-def*)
lemma *interior-maximal*: $T \subseteq S \Longrightarrow$ *open* T ==> $T \subseteq$ (*interior* S) **by** (*auto simp add*: *interior-def*)
lemma *interior-unique*: $T \subseteq S \Longrightarrow$ *open* $T \Longrightarrow (\forall\, T'.\ T' \subseteq S \wedge open\ T' \longrightarrow T' \subseteq T) \Longrightarrow$ *interior* $S = T$
by (*metis equalityI interior-maximal interior-subset open-interior*)
lemma *mem-interior*: $x \in$ *interior* $S \longleftrightarrow (\exists\, e.\ 0 < e \wedge ball\ x\ e \subseteq S)$
apply (*simp add*: *interior-def*)
by (*metis open-contains-ball centre-in-ball open-ball subset-trans*)

lemma *open-subset-interior*: *open* S ==> $S \subseteq$ *interior* $T \longleftrightarrow S \subseteq T$
by (*metis interior-maximal interior-subset subset-trans*)

lemma *interior-inter*[*simp*]: *interior*$(S \cap T)$ = *interior* $S \cap$ *interior* T
apply (*rule equalityI*, *simp*)

apply (*metis Int-lower1 Int-lower2 subset-interior*)
by (*metis Int-mono interior-subset open-inter open-interior open-subset-interior*)

lemma *interior-limit-point*[*intro*]: **assumes** *x*: $x \in$ *interior S* **shows** *x islimpt S*
proof−
from *x* **obtain** *e* **where** *e*: $e{>}0\ \forall x'.\ dist\ x\ x' < e \longrightarrow x' \in S$
unfolding *mem-interior subset-eq Ball-def mem-ball* **by** *blast*
{**fix** *d*::*real* **assume** *d*: *d>0*
let *?m* = *min d e* / *2*
have *mde2*: *?m* ≥ *0* **using** *e*(*1*) *d*(*1*) **by** *arith*
from *vector-choose-dist*[*OF mde2*, *of x*]
obtain *y* **where** *y*: *dist x y* = *?m* **by** *blast*
have *th*: *dist x y* < *e dist x y* < *d* **unfolding** *y* **using** *e*(*1*) *d*(*1*) **by** *arith*+
have ∃ *x'*∈*S*. *x'*≠ *x* ∧ *dist x' x* < *d*
apply (*rule bexI*[**where** *x*=*y*])
using *e th y* **by** (*auto simp add*: *dist-sym*)}
then show *?thesis* **unfolding** *islimpt-approachable* **by** *blast*
qed

lemma *interior-closed-Un-empty-interior*:
assumes *cS*: *closed S* **and** *iT*: *interior T* = {}
shows *interior*(*S* ∪ *T*) = *interior S*
proof−
have *interior S* ⊆ *interior* (*S*∪*T*)
by (*rule subset-interior*, *blast*)
moreover
{**fix** *x e* **assume** *e*: *e* > *0* ∀ *x'* ∈ *ball x e*. *x'*∈(*S*∪*T*)
{**fix** *y* **assume** *y*: *y* ∈ *ball x e*
{**fix** *d*::*real* **assume** *d*: *d* > *0*
let *?k* = *min d* (*e* − *dist x y*)
have *kp*: *?k* > *0* **using** *d e*(*1*) *y*[*unfolded mem-ball*] **by** *norm*
have *?k*/*2* ≥ *0* **using** *kp* **by** *simp*
then obtain *w* **where** *w*: *dist y w* = *?k*/ *2* **by** (*metis vector-choose-dist*)
from *iT*[*unfolded expand-set-eq mem-interior*]
have ¬ *ball w* (*?k*/*4*) ⊆ *T* **using** *kp* **by** (*auto simp add*: *less-divide-eq-number-of1*)
then obtain *z* **where** *z*: *dist w z* < *?k*/*4 z* ∉ *T* **by** (*auto simp add*: *subset-eq*)
have *z* ∉ *T* ∧ *z*≠ *y* ∧ *dist z y* < *d* ∧ *dist x z* < *e* **using** *z* **apply** *simp*
using *w e*(*1*) *d* **apply** (*auto simp only*: *dist-sym*)
apply (*auto simp add*: *min-def cong del*: *if-weak-cong*)
apply (*cases d* ≤ *e* − *dist x y*, *auto simp add*: *ring-simps cong del*: *if-weak-cong*)
apply *norm*
apply (*cases d* ≤ *e* − *dist x y*, *auto simp add*: *ring-simps not-le not-less cong del*: *if-weak-cong*)
apply *norm*
apply *norm*
apply (*cases d* ≤ *e* − *dist x y*, *auto simp add*: *ring-simps not-le not-less cong del*: *if-weak-cong*)

apply *norm*
apply *norm*
done
then have $\exists z.\ z \notin T \wedge z \neq y \wedge dist\ z\ y < d \wedge dist\ x\ z < e$ **by** *blast*
then have $\exists x' \in S.\ x' \neq y \wedge dist\ x'\ y < d$ **using** *e* **by** *auto*}
then have $y \in S$ **by** (*metis islimpt-approachable cS closed-limpt*) }
then have $x \in interior\ S$ **unfolding** *mem-interior* **using** *e*(*1*) **by** *blast*}
hence $interior\ (S \cup T) \subseteq interior\ S$ **unfolding** *mem-interior Ball-def subset-eq*
by *blast*
ultimately show *?thesis* **by** *blast*
qed

64.13 Closure of a Set

definition $closure\ S = S \cup \{x \mid x.\ x\ islimpt\ S\}$

lemma *closure-interior*: $closure\ S = UNIV - interior\ (UNIV - S)$
proof−
{ **fix** *x*
have $x \in UNIV - interior\ (UNIV - S) \longleftrightarrow x \in closure\ S$ (**is** *?lhs* = *?rhs*)
proof
let *?exT* = $\lambda\ y.\ (\exists\ T.\ open\ T \wedge y \in T \wedge T \subseteq UNIV - S)$
assume *?lhs*
hence ∗:¬ *?exT x*
unfolding *interior-def*
by *simp*
{ **assume** ¬ *?rhs*
hence *False* **using** ∗
unfolding *closure-def islimpt-def*
by *blast*
}
thus *?rhs*
by *blast*
next
assume *?rhs* **thus** *?lhs*
unfolding *closure-def interior-def islimpt-def*
by *blast*
qed
}
thus *?thesis*
by *blast*
qed

lemma *interior-closure*: $interior\ S = UNIV - (closure\ (UNIV - S))$
proof−
{ **fix** *x*
have $x \in interior\ S \longleftrightarrow x \in UNIV - (closure\ (UNIV - S))$
unfolding *interior-def closure-def islimpt-def*
by *blast*

```
  }
  thus ?thesis
    by blast
qed

lemma closed-closure[simp, intro]: closed (closure S)
proof−
  have closed (UNIV − interior (UNIV −S)) by blast
  thus ?thesis using closure-interior[of S] by simp
qed

lemma closure-hull: closure S = closed hull S
proof−
  have S ⊆ closure S
    unfolding closure-def
    by blast
  moreover
  have closed (closure S)
    using closed-closure[of S]
    by assumption
  moreover
  { fix t
    assume ∗:S ⊆ t closed t
    { fix x
      assume x islimpt S
      hence x islimpt t using ∗(1)
        using islimpt-subset[of x, of S, of t]
        by blast
    }
    with ∗ have closure S ⊆ t
      unfolding closure-def
      using closed-limpt[of t]
      by blast
  }
  ultimately show ?thesis
    using hull-unique[of S, of closure S, of closed]
    unfolding mem-def
    by simp
qed

lemma closure-eq: closure S = S ⟷ closed S
  unfolding closure-hull
  using hull-eq[of closed, unfolded mem-def, OF  closed-Inter, of S]
  by (metis mem-def subset-eq)

lemma closure-closed[simp]: closed S ⟹ closure S = S
  using closure-eq[of S]
  by simp
```

lemma *closure-closure*[*simp*]: *closure* (*closure S*) = *closure S*
unfolding *closure-hull*
using *hull-hull*[*of closed S*]
by *assumption*

lemma *closure-subset*: $S \subseteq$ *closure S*
unfolding *closure-hull*
using *hull-subset*[*of S closed*]
by *assumption*

lemma *subset-closure*: $S \subseteq T \Longrightarrow$ *closure S* $\subseteq$ *closure T*
unfolding *closure-hull*
using *hull-mono*[*of S T closed*]
by *assumption*

lemma *closure-minimal*: $S \subseteq T \Longrightarrow$ *closed T* $\Longrightarrow$ *closure S* $\subseteq T$
using *hull-minimal*[*of S T closed*]
unfolding *closure-hull mem-def*
by *simp*

lemma *closure-unique*: $S \subseteq T \wedge$ *closed T* $\wedge$ $(\forall\ T'.\ S \subseteq T' \wedge$ *closed* $T' \longrightarrow T \subseteq T') \Longrightarrow$ *closure S* $= T$
using *hull-unique*[*of S T closed*]
unfolding *closure-hull mem-def*
by *simp*

lemma *closure-empty*[*simp*]: *closure* $\{\} = \{\}$
using *closed-empty closure-closed*[*of* $\{\}$]
by *simp*

lemma *closure-univ*[*simp*]: *closure UNIV* = *UNIV*
using *closure-closed*[*of UNIV*]
by *simp*

lemma *closure-eq-empty*: *closure S* $= \{\} \longleftrightarrow S = \{\}$
using *closure-empty closure-subset*[*of S*]
by *blast*

lemma *closure-subset-eq*: *closure S* $\subseteq S \longleftrightarrow$ *closed S*
using *closure-eq*[*of S*] *closure-subset*[*of S*]
by *simp*

lemma *open-inter-closure-eq-empty*:
open S $\Longrightarrow$ $(S \cap$ *closure T*$) = \{\} \longleftrightarrow S \cap T = \{\}$
using *open-subset-interior*[*of S UNIV* $-$ *T*]
using *interior-subset*[*of UNIV* $-$ *T*]
unfolding *closure-interior*
by *auto*

lemma *open-inter-closure-subset*: *open S* $\Longrightarrow$ $(S \cap (closure\ T)) \subseteq closure(S \cap T)$
proof
fix *x*
assume *as*: *open S x* $\in S \cap closure\ T$
{ assume $*$:*x islimpt T*
{ fix *e*::*real*
assume $e > 0$
from *as* ‹*open S*› **obtain** e' **where** $e' > 0$ **and** e':$\forall x'.\ dist\ x'\ x < e' \longrightarrow x' \in S$
unfolding *open-def*
by *auto*
let *?e* = *min e e'*
from ‹*e*>*0*› ‹*e'*>*0*› **have** *?e* > *0*
by *simp*
then obtain *y* **where** *y*:$y \in T\ y \neq x \wedge dist\ y\ x < ?e$
using *islimpt-approachable*[*of x T*] **using** $*$
by *blast*
hence $\exists x' \in S \cap T.\ x' \neq x \wedge dist\ x'\ x < e$ **using** e'
using *y*
by(*rule-tac x*=*y* **in** *bexI*, *simp*+)
}
hence *x islimpt S* $\cap$ *T*
using *islimpt-approachable*[*of x S* $\cap$ *T*]
by *blast*
}
then show *x* $\in$ *closure* (*S* $\cap$ *T*) **using** *as*
unfolding *closure-def*
by *blast*
qed

lemma *closure-complement*: *closure*(*UNIV* $-$ *S*) = *UNIV* $-$ *interior*(*S*)
proof$-$
have *S* = *UNIV* $-$ (*UNIV* $-$ *S*)
by *auto*
thus *?thesis*
unfolding *closure-interior*
by *auto*
qed

lemma *interior-complement*: *interior*(*UNIV* $-$ *S*) = *UNIV* $-$ *closure*(*S*)
unfolding *closure-interior*
by *blast*

64.14 Frontier (aka boundary)

definition *frontier S* = *closure S* $-$ *interior S*

lemma *frontier-closed*: *closed*(*frontier S*)
by (*simp add*: *frontier-def closed-diff closed-closure*)

lemma *frontier-closures*: *frontier S* = (*closure S*) ∩ (*closure*(*UNIV* − *S*))
by (*auto simp add*: *frontier-def interior-closure*)

lemma *frontier-straddle*: *a* ∈ *frontier S* ⟷ (∀ *e*>*0*. (∃ *x*∈*S*. *dist a x* < *e*) ∧ (∃ *x*. *x* ∉ *S* ∧ *dist a x* < *e*)) (**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs*
{ fix *e*::*real*
assume *e* > *0*
let *?rhse* = (∃ *x*∈*S*. *dist a x* < *e*) ∧ (∃ *x*. *x* ∉ *S* ∧ *dist a x* < *e*)
{ assume *a*∈*S*
have ∃ *x*∈*S*. *dist a x* < *e* **using** *dist-refl*[*of a*] ‹*e*>*0*› ‹*a*∈*S*› **by**(*rule-tac x*=*a* **in** *bexI*) *auto*
moreover have ∃ *x*. *x* ∉ *S* ∧ *dist a x* < *e* **using** ‹*?lhs*› ‹*a*∈*S*›
unfolding *frontier-closures closure-def islimpt-def* **using** *dist-refl*[*of a*] ‹*e*>*0*›
by (*auto*, *erule-tac x*=*ball a e* **in** *allE*, *auto*)
ultimately have *?rhse* **by** *auto*
}
moreover
{ assume *a*∉*S*
hence *?rhse* **using** ‹*?lhs*›
unfolding *frontier-closures closure-def islimpt-def*
using *open-ball*[*of a e*] *dist-refl*[*of a*] ‹*e* > *0*›
by (*auto*, *erule-tac x* = *ball a e* **in** *allE*, *auto*)
}
ultimately have *?rhse* **by** *auto*
}
thus *?rhs* **by** *auto*
next
assume *?rhs*
moreover
{ fix *T* **assume** *a*∉*S* **and**
as:∀ *e*>*0*. (∃ *x*∈*S*. *dist a x* < *e*) ∧ (∃ *x*. *x* ∉ *S* ∧ *dist a x* < *e*) *a* ∉ *S a* ∈ *T open T*
from ‹*open T*› ‹*a* ∈ *T*› **have** ∃ *e*>*0*. *ball a e* ⊆ *T* **unfolding** *open-contains-ball*[*of T*] **by** *auto*
then obtain *e* **where** *e*>*0 ball a e* ⊆ *T* **by** *auto*
then obtain *y* **where** *y*:*y*∈*S dist a y* < *e* **using** *as*(*1*) **by** *auto*
have ∃ *y*∈*S*. *y* ∈ *T* ∧ *y* ≠ *a*
using ‹*dist a y* < *e*› ‹*ball a e* ⊆ *T*› **unfolding** *ball-def* **using** ‹*y*∈*S*› ‹*a*∉*S*› **by** *auto*
}
hence *a* ∈ *closure S* **unfolding** *closure-def islimpt-def* **using** ‹*?rhs*› **by** *auto*
moreover
{ fix *T* **assume** *a* ∈ *T open T a*∈*S*
then obtain *e* **where** *e*>*0* **and** *balle*: *ball a e* ⊆ *T* **unfolding** *open-contains-ball* **using** ‹*?rhs*› **by** *auto*
obtain *x* **where** *x* ∉ *S dist a x* < *e* **using** ‹*?rhs*› **using** ‹*e*>*0*› **by** *auto*

hence $\exists\, y{\in}UNIV - S.\ y \in T \wedge y \neq a$ **using** *balle* ‹*a*∈*S*› **unfolding** *ball-def*
by (*rule-tac x*=*x* **in** *bexI*)*auto*
}
hence *a islimpt* (*UNIV* − *S*) ∨ *a*∉*S* **unfolding** *islimpt-def* **by** *auto*
ultimately show *?lhs* **unfolding** *frontier-closures* **using** *closure-def*[*of UNIV* − *S*] **by** *auto*
qed

lemma *frontier-subset-closed*: *closed S* ⟹ *frontier S* ⊆ *S*
by (*metis frontier-def closure-closed Diff-subset*)

lemma *frontier-empty*: *frontier* {} = {}
by (*simp add*: *frontier-def closure-empty*)

lemma *frontier-subset-eq*: *frontier S* ⊆ *S* ⟷ *closed S*
proof−
{ assume *frontier S* ⊆ *S*
hence *closure S* ⊆ *S* **using** *interior-subset* **unfolding** *frontier-def* **by** *auto*
hence *closed S* **using** *closure-subset-eq* **by** *auto*
}
thus *?thesis* **using** *frontier-subset-closed*[*of S*] **by** *auto*
qed

lemma *frontier-complement*: *frontier*(*UNIV* − *S*) = *frontier S*
by (*auto simp add*: *frontier-def closure-complement interior-complement*)

lemma *frontier-disjoint-eq*: *frontier S* ∩ *S* = {} ⟷ *open S*
using *frontier-complement frontier-subset-eq*[*of UNIV* − *S*]
unfolding *open-closed* **by** *auto*

64.15 A variant of nets (Slightly non-standard but good for our purposes).

typedef (**open**) *'a net* =
{*g* :: *'a* ⇒ *'a* ⇒ *bool*. ∀ *x y*. (∀ *z*. *g z x* ⟶ *g z y*) ∨ (∀ *z*. *g z y* ⟶ *g z x*)}
morphisms *netord mknet* **by** *blast*
lemma *net*: (∀ *z*. *netord n z x* ⟶ *netord n z y*) ∨ (∀ *z*. *netord n z y* ⟶ *netord n z x*)
using *netord*[*of n*] **by** *auto*

lemma *oldnet*: *netord n x x* ⟹ *netord n y y* ⟹
∃ *z*. *netord n z z* ∧ (∀ *w*. *netord n w z* ⟶ *netord n w x* ∧ *netord n w y*)
by (*metis net*)

lemma *net-dilemma*:
∃ *a*. (∃ *x*. *netord net x a*) ∧ (∀ *x*. *netord net x a* ⟶ *P x*) ⟹
∃ *b*. (∃ *x*. *netord net x b*) ∧ (∀ *x*. *netord net x b* ⟶ *Q x*)
⟹ ∃ *c*. (∃ *x*. *netord net x c*) ∧ (∀ *x*. *netord net x c* ⟶ *P x* ∧ *Q x*)
by (*metis net*)

64.16 Common nets and The "within" modifier for nets.

definition *at a = mknet(λx y. 0 < dist x a ∧ dist x a <= dist y a)*
definition *at-infinity = mknet(λx y. norm x ≥ norm y)*
definition *sequentially = mknet(λ(m::nat) n. m >= n)*

definition *within* :: *'a net ⇒ 'a set ⇒ 'a net* (**infixr** *within 70*) **where**
within-def: *net within S = mknet (λx y. netord net x y ∧ x ∈ S)*

definition *indirection* :: *real ^'n::finite ⇒ real ^'n ⇒ (real ^'n) net* (**infixr** *indirection 70*) **where**
indirection-def: *a indirection v = (at a) within {b. ∃ c≥0. b − a = c*s v}*

Prove That They are all nets.

lemma *mknet-inverse'*: *netord (mknet r) = r ⟷ (∀ x y. (∀ z. r z x ⟶ r z y) ∨ (∀ z. r z y ⟶ r z x))*
using *mknet-inverse[of r]* **apply** (*auto simp add*: *netord-inverse*) **by** (*metis net*)

method-setup *net* = ⟪
let
val ss1 = HOL-basic-ss addsimps [@{thm expand-fun-eq} RS sym]
val ss2 = HOL-basic-ss addsimps [@{thm mknet-inverse'}]
fun tac ths = ObjectLogic.full-atomize-tac THEN' Simplifier.simp-tac (ss1 addsimps ths) THEN' Simplifier.asm-full-simp-tac ss2
in Attrib.thms >> (fn ths => K (SIMPLE-METHOD' (tac ths))) end
⟫ *reduces goals about net*

lemma *at*: ⋀*x y. netord (at a) x y ⟷ 0 < dist x a ∧ dist x a <= dist y a*
apply (*net at-def*)
by (*metis dist-sym real-le-linear real-le-trans*)

lemma *at-infinity*:
⋀*x y. netord at-infinity x y ⟷ norm x >= norm y*
apply (*net at-infinity-def*)
apply (*metis real-le-linear real-le-trans*)
done

lemma *sequentially*: ⋀*m n. netord sequentially m n ⟷ m >= n*
apply (*net sequentially-def*)
apply (*metis linorder-linear min-max.le-supI2 min-max.sup-absorb1*)
done

lemma *within*: *netord (n within S) x y ⟷ netord n x y ∧ x ∈ S*
proof−
have *∀ x y. (∀ z. netord n z x ∧ z ∈ S ⟶ netord n z y) ∨ (∀ z. netord n z y ∧ z ∈ S ⟶ netord n z x)*
by (*metis net*)
thus *?thesis*
unfolding *within-def*
using *mknet-inverse[of λx y. netord n x y ∧ x ∈ S]*

by *simp*
qed

lemma *in-direction*: *netord* (*a indirection v*) *x y* ⟷ *0* < *dist x a* ∧ *dist x a* ≤ *dist y a* ∧ (∃ *c* ≥ *0*. *x* − *a* = *c* ∗*s v*)
by (*simp add*: *within at indirection-def*)

lemma *within-UNIV*: *at x within UNIV* = *at x*
by (*simp add*: *within-def at-def netord-inverse*)

64.17 Identify Trivial limits, where we can't approach arbitrarily closely.

definition *trivial-limit* (*net*:: *'a net*) ⟷
(∀ (*a*::*'a*) *b*. *a* = *b*) ∨ (∃ (*a*::*'a*) *b*. *a* ≠ *b* ∧ (∀ *x*. ~(*netord* (*net*) *x a*) ∧ ~(*netord*(*net*) *x b*)))

lemma *trivial-limit-within*: *trivial-limit* (*at* (*a*::*real*^*'n*::*finite*) *within S*) ⟷ ~(*a islimpt S*)
proof−
{assume ∀ (*a*::*real*^*'n*) *b*. *a* = *b* **hence** ¬ *a islimpt S*
apply (*simp add*: *islimpt-approachable-le*)
by (*rule exI*[**where** *x*=*1*], *auto*)**}**
moreover
{fix *b c* **assume** *bc*: *b* ≠ *c* ∀ *x*. ¬ *netord* (*at a within S*) *x b* ∧ ¬ *netord* (*at a within S*) *x c*
have *dist a b* > *0* ∨ *dist a c* > *0* **using** *bc* **by** (*auto simp add*: *within at dist-nz*[*THEN sym*])
then have ¬ *a islimpt S*
using *bc*
unfolding *within at dist-nz islimpt-approachable-le*
by(*auto simp add*: *dist-triangle dist-sym dist-eq-0*[*THEN sym*]) **}**
moreover
{assume ¬ *a islimpt S*
then obtain *e* **where** *e*: *e* > *0* ∀ *x'* ∈ *S*. *x'* ≠ *a* ⟶ *dist x' a* > *e*
unfolding *islimpt-approachable-le* **by** (*auto simp add*: *not-le*)
from *e vector-choose-dist*[*of e a*] **obtain** *b* **where** *b*: *dist a b* = *e* **by** *auto*
from *b e*(*1*) **have** *a* ≠ *b* **by** (*simp add*: *dist-nz*)
moreover have ∀ *x*. ¬ ((*0* < *dist x a* ∧ *dist x a* ≤ *dist a a*) ∧ *x* ∈ *S*) ∧
¬ ((*0* < *dist x a* ∧ *dist x a* ≤ *dist b a*) ∧ *x* ∈ *S*)
using *e*(*2*) *b* **by** (*auto simp add*: *dist-refl dist-sym*)
ultimately have *trivial-limit* (*at a within S*) **unfolding** *trivial-limit-def within at*
by *blast***}**
ultimately show *?thesis* **unfolding** *trivial-limit-def* **by** *blast*
qed

lemma *trivial-limit-at*: ~(*trivial-limit* (*at a*))

apply (*subst within-UNIV*[*symmetric*])
by (*simp add*: *trivial-limit-within islimpt-UNIV*)

lemma *trivial-limit-at-infinity*: ~(*trivial-limit* (*at-infinity* :: ('*a*::{*norm*,*zero-neq-one*}) *net*))
apply (*simp add*: *trivial-limit-def at-infinity*)
by (*metis order-refl zero-neq-one*)

lemma *trivial-limit-sequentially*: ~(*trivial-limit sequentially*)
by (*auto simp add*: *trivial-limit-def sequentially*)

64.18 Some property holds "sufficiently close" to the limit point.

definition *eventually P net* ⟷ *trivial-limit net* ∨ (∃ *y*. (∃ *x*. *netord net x y*) ∧ (∀ *x*. *netord net x y* ⟶ *P x*))

lemma *eventually-happens*: *eventually P net* ==> *trivial-limit net* ∨ (∃ *x*. *P x*)
by (*metis eventually-def*)

lemma *eventually-within-le*: *eventually P* (*at a within S*) ⟷
(∃ *d*>*0*. ∀ *x*∈*S*. *0* < *dist x a* ∧ *dist x a* <= *d* ⟶ *P x*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
moreover
{ assume ¬ *a islimpt S*
then obtain *e* **where** *e*>*0* **and** *e*:∀ *x'*∈*S*. ¬ (*x'* ≠ *a* ∧ *dist x' a* ≤ *e*) **unfolding** *islimpt-approachable-le* **by** *auto*
hence *?rhs* **apply** *auto* **apply** (*rule-tac x*=*e* **in** *exI*) **by** *auto* **}**
moreover
{ assume ∃ *y*. (∃ *x*. *netord* (*at a within S*) *x y*) ∧ (∀ *x*. *netord* (*at a within S*) *x y* ⟶ *P x*)
then obtain *x y* **where** *xy*:*netord* (*at a within S*) *x y* ∧ (∀ *x*. *netord* (*at a within S*) *x y* ⟶ *P x*) **by** *auto*
hence *?rhs* **unfolding** *within at* **by** *auto*
}
ultimately show *?rhs* **unfolding** *eventually-def trivial-limit-within* **by** *auto*
next
assume *?rhs*
then obtain *d* **where** *d*>*0* **and** *d*:∀ *x*∈*S*. *0* < *dist x a* ∧ *dist x a* ≤ *d* ⟶ *P x* **by** *auto*
thus *?lhs*
unfolding *eventually-def trivial-limit-within islimpt-approachable-le within at* **unfolding** *dist-nz*[*THEN sym*] **by** (*clarsimp*, *rule-tac x*=*d* **in** *exI*, *auto*)
qed

lemma *eventually-within*: *eventually P* (*at a within S*) ⟷
(∃ *d*>*0*. ∀ *x*∈*S*. *0* < *dist x a* ∧ *dist x a* < *d* ⟶ *P x*)
proof−

{ fix *d*
assume *d>0* ∀ *x∈S. 0 < dist x a ∧ dist x a < d ⟶ P x*
hence ∀ *x∈S. 0 < dist x a ∧ dist x a ≤ (d/2) ⟶ P x* **using** *order-less-imp-le*
by *auto*
}
thus *?thesis* **unfolding** *eventually-within-le* **using** *approachable-lt-le*
by (*auto*, *rule-tac x=d/2* **in** *exI*, *auto*)
qed

lemma *eventually-at*: *eventually P* (*at a*) ⟷ (∃ *d>0*. ∀ *x*. *0 < dist x a ∧ dist x a < d ⟶ P x*)
apply (*subst within-UNIV*[*symmetric*])
by (*simp add*: *eventually-within*)

lemma *eventually-sequentially*: *eventually P sequentially* ⟷ (∃ *N*. ∀ *n≥N*. *P n*)
apply (*simp add*: *eventually-def sequentially trivial-limit-sequentially*)
apply (*metis dlo-simps*(*7*) *dlo-simps*(*9*) *le-maxI2 min-max.le-iff-sup min-max.sup-absorb1 order-antisym-conv*) **done**

lemma *eventually-at-infinity*: *eventually* (*P*::(*real^'n*::*finite* ⇒ *bool*)) *at-infinity* ⟷ (∃ *b*. ∀ *x*. *norm x >= b ⟶ P x*) (**is** *?lhs = ?rhs*)
proof
assume *?lhs* **thus** *?rhs*
unfolding *eventually-def at-infinity*
by (*auto simp add*: *trivial-limit-at-infinity*)
next
assume *?rhs*
then obtain *b* **where** *b*:∀ *x*. *b ≤ norm x ⟶ P x* **and** *b≥0*
by (*metis norm-ge-zero real-le-linear real-le-trans*)
obtain *y*::*real^'n* **where** *y*:*norm y = b* **using** ‹*b≥0*›
using *vector-choose-size*[*of b*] **by** *auto*
thus *?lhs* **unfolding** *eventually-def at-infinity* **using** *b y* **by** *auto*
qed

lemma *always-eventually*: (∀ (*x*::*'a*::*zero-neq-one*). *P x*) ==> *eventually P net*
apply (*auto simp add*: *eventually-def trivial-limit-def*)
by (*rule exI*[**where** *x=0*], *rule exI*[**where** *x=1*], *rule zero-neq-one*)

Combining theorems for "eventually"

lemma *eventually-and*: *eventually* (λ*x*. *P x ∧ Q x*) *net* ⟷ *eventually P net ∧ eventually Q net*
apply (*simp add*: *eventually-def*)
apply (*cases trivial-limit net*)
using *net-dilemma*[*of net P Q*] **by** *auto*

lemma *eventually-mono*: (∀ *x*. *P x ⟶ Q x*) ⟹ *eventually P net* ⟹ *eventually Q net*
by (*metis eventually-def*)

lemma *eventually-mp*: *eventually* (λx. *P x* ⟶ *Q x*) *net* ⟹ *eventually P net* ⟹ *eventually Q net*
apply (*atomize*(*full*))
unfolding *imp-conjL*[*symmetric*] *eventually-and*[*symmetric*]
by (*auto simp add*: *eventually-def*)

lemma *eventually-false*: *eventually* (λx. *False*) *net* ⟷ *trivial-limit net*
by (*auto simp add*: *eventually-def*)

lemma *not-eventually*: (∀ x. ¬ *P x*) ⟹ ~(*trivial-limit net*) ==> ~(*eventually P net*)
by (*auto simp add*: *eventually-def*)

64.19 Limits, defined as vacuously true when the limit is trivial.

definition *tendsto*:: ('*a* ⇒ *real* ^'*n*::*finite*) ⇒ *real* ^'*n* ⇒ '*a net* ⇒ *bool* (**infixr** −−−> *55*) **where**
tendsto-def: (*f* −−−> *l*) *net* ⟷ (∀ *e*>*0*. *eventually* (λx. *dist* (*f x*) *l* < *e*) *net*)

lemma *tendstoD*: (*f* −−−> *l*) *net* ⟹ *e*>*0* ⟹ *eventually* (λx. *dist* (*f x*) *l* < *e*) *net*
unfolding *tendsto-def* **by** *auto*

Notation Lim to avoid collition with lim defined in analysis

definition *Lim net f* = (*THE l*. (*f* −−−> *l*) *net*)

lemma *Lim*:
(*f* −−−> *l*) *net* ⟷
trivial-limit net ∨
(∀ *e*>*0*. ∃ *y*. (∃ *x*. *netord net x y*) ∧
(∀ *x*. *netord*(*net*) *x y* ⟶ *dist* (*f x*) *l* < *e*))
by (*auto simp add*: *tendsto-def eventually-def*)

Show that they yield usual definitions in the various cases.

lemma *Lim-within-le*: (*f* −−−> *l*)(*at a within S*) ⟷
(∀ *e*>*0*. ∃ *d*>*0*. ∀ *x*∈*S*. *0* < *dist x a* ∧ *dist x a* <= *d* ⟶ *dist* (*f x*) *l* < *e*)
by (*auto simp add*: *tendsto-def eventually-within-le*)

lemma *Lim-within*: (*f* −−−> *l*) (*at a within S*) ⟷
(∀ *e* >*0*. ∃ *d*>*0*. ∀ *x* ∈ *S*. *0* < *dist x a* ∧ *dist x a* < *d* ⟶ *dist* (*f x*) *l* < *e*)
by (*auto simp add*: *tendsto-def eventually-within*)

lemma *Lim-at*: (*f* −−−> *l*) (*at a*) ⟷
(∀ *e* >*0*. ∃ *d*>*0*. ∀ *x*. *0* < *dist x a* ∧ *dist x a* < *d* ⟶ *dist* (*f x*) *l* < *e*)
by (*auto simp add*: *tendsto-def eventually-at*)

lemma *Lim-at-infinity*:
(*f* −−−> *l*) *at-infinity* ⟷ (∀ *e*>*0*. ∃ *b*. ∀ *x*::*real*^'*n*::*finite*. *norm x* >= *b* ⟶ *dist* (*f x*) *l* < *e*)
by (*auto simp add*: *tendsto-def eventually-at-infinity*)

lemma *Lim-sequentially*:
(*S* −−−> *l*) *sequentially* ⟷
(∀ *e*>*0*. ∃ *N*. ∀ *n*≥*N*. *dist* (*S n*) *l* < *e*)
by (*auto simp add*: *tendsto-def eventually-sequentially*)

lemma *Lim-eventually*: *eventually* (λ*x*. *f x* = *l*) *net* ⟹ (*f* −−−> *l*) *net*
by (*auto simp add*: *eventually-def Lim dist-refl*)

The expected monotonicity property.

lemma *Lim-within-empty*: (*f* −−−> *l*) (*at x within* {})
by (*simp add*: *Lim-within-le*)

lemma *Lim-within-subset*: (*f* −−−> *l*) (*at a within S*) ⟹ *T* ⊆ *S* ⟹ (*f* −−−> *l*) (*at a within T*)
apply (*auto simp add*: *Lim-within-le*)
by (*metis subset-eq*)

lemma *Lim-Un*: **assumes** (*f* −−−> *l*) (*at x within S*) (*f* −−−> *l*) (*at x within T*)
shows (*f* −−−> *l*) (*at x within* (*S* ∪ *T*))
proof−
{ fix *e*::*real* **assume** *e*>*0*
obtain *d1* **where** *d1*:*d1*>*0* ∀ *xa*∈*T*. *0* < *dist xa x* ∧ *dist xa x* < *d1* ⟶ *dist* (*f xa*) *l* < *e* **using** *assms* **unfolding** *Lim-within* **using** ‹*e*>*0*› **by** *auto*
obtain *d2* **where** *d2*:*d2*>*0* ∀ *xa*∈*S*. *0* < *dist xa x* ∧ *dist xa x* < *d2* ⟶ *dist* (*f xa*) *l* < *e* **using** *assms* **unfolding** *Lim-within* **using** ‹*e*>*0*› **by** *auto*
have ∃ *d*>*0*. ∀ *xa*∈*S* ∪ *T*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *dist* (*f xa*) *l* < *e* **using** *d1 d2*
by (*rule-tac x*=*min d1 d2* **in** *exI*)*auto*
}
thus *?thesis* **unfolding** *Lim-within* **by** *auto*
qed

lemma *Lim-Un-univ*:
(*f* −−−> *l*) (*at x within S*) ⟹ (*f* −−−> *l*) (*at x within T*) ⟹ *S* ∪ *T* = (*UNIV*::(*real*^'*n*::*finite*) *set*)
==> (*f* −−−> *l*) (*at x*)
by (*metis Lim-Un within-UNIV*)

Interrelations between restricted and unrestricted limits.

lemma *Lim-at-within*: (*f* −−−> *l*)(*at a*) ==> (*f* −−−> *l*)(*at a within S*)
apply (*simp add*: *Lim-at Lim-within*)
by *metis*

lemma *Lim-within-open*:
assumes $a \in S$ *open* S
shows $(f \text{ ---> } l)(at\ a\ within\ S) \longleftrightarrow (f \text{ ---> } l)(at\ a)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
{ fix *e*::*real* **assume** *e*>*0*
obtain *d* **where** *d*: $d > 0\ \forall x{\in}S.\ 0 < dist\ x\ a \wedge dist\ x\ a < d \longrightarrow dist\ (f\ x)\ l < e$ **using** ‹*?lhs*› ‹*e*>*0*› **unfolding** *Lim-within* **by** *auto*
obtain *d'* **where** *d'*: $d'{>}0\ \forall x.\ dist\ x\ a < d' \longrightarrow x \in S$ **using** *assms* **unfolding** *open-def* **by** *auto*
from *d d'* **have** $\exists d{>}0.\ \forall x.\ 0 < dist\ x\ a \wedge dist\ x\ a < d \longrightarrow dist\ (f\ x)\ l < e$ **by** (*rule-tac x*= *min d d'* **in** *exI*)*auto*
}
thus *?rhs* **unfolding** *Lim-at* **by** *auto*
next
assume *?rhs*
{ fix *e*::*real* **assume** *e*>*0*
then obtain *d* **where** *d*>*0* **and** *d*:$\forall x.\ 0 < dist\ x\ a \wedge dist\ x\ a < d \longrightarrow dist\ (f\ x)\ l < e$ **using** ‹*?rhs*› **unfolding** *Lim-at* **by** *auto*
hence $\exists d{>}0.\ \forall x.\ 0 < dist\ x\ a \wedge dist\ x\ a < d \longrightarrow dist\ (f\ x)\ l < e$ **using** ‹*d*>*0*› **by** *auto*
}
thus *?lhs* **using** *Lim-at-within*[*of f l a S*] **by** (*auto simp add*: *Lim-at*)
qed

Another limit point characterization.

lemma *islimpt-sequential*:
x islimpt S $\longleftrightarrow$ $(\exists f.\ (\forall n{::}nat.\ f\ n \in S - \{x\}) \wedge (f \text{ ---> } x)\ sequentially)$ (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain *f* **where** *f*:$\forall y.\ y{>}0 \longrightarrow f\ y \in S \wedge f\ y \neq x \wedge dist\ (f\ y)\ x < y$
unfolding *islimpt-approachable* **using** *choice*[*of* $\lambda e\ y.\ e{>}0 \longrightarrow y{\in}S \wedge y{\neq}x \wedge dist\ y\ x < e$] **by** *auto*
{ fix *n*::*nat*
have *f* (*inverse* (*real n* + *1*)) ∈ *S* − {*x*} **using** *f* **by** *auto*
}
moreover
{ fix *e*::*real* **assume** *e*>*0*
hence $\exists N{::}nat.\ inverse\ (real\ (N + 1)) < e$ **using** *real-arch-inv*[*of e*] **apply** (*auto simp add*: *Suc-pred'*) **apply**(*rule-tac x*=*n* − *1* **in** *exI*) **by** *auto*
then obtain *N*::*nat* **where** *inverse* (*real* (*N* + *1*)) < *e* **by** *auto*
hence $\forall n{\geq}N.\ inverse\ (real\ n + 1) < e$ **by** (*auto*, *metis Suc-le-mono le-SucE less-imp-inverse-less nat-le-real-less order-less-trans real-of-nat-Suc real-of-nat-Suc-gt-zero*)
moreover have $\forall n{\geq}N.\ dist\ (f\ (inverse\ (real\ n + 1)))\ x < (inverse\ (real\ n + 1))$ **using** *f* ‹*e*>*0*› **by** *auto*
ultimately have $\exists N{::}nat.\ \forall n{\geq}N.\ dist\ (f\ (inverse\ (real\ n + 1)))\ x < e$ **apply**(*rule-tac x*=*N* **in** *exI*) **apply** *auto* **apply**(*erule-tac x*=*n* **in** *allE*)+ **by** *auto*
}

hence $((\lambda n.\ f\ (inverse\ (real\ n + 1))) \longrightarrow x)$ *sequentially*
unfolding *Lim-sequentially* **using** *f* **by** *auto*
ultimately show *?rhs* **apply** (*rule-tac* $x=(\lambda n{::}nat.\ f\ (inverse\ (real\ n + 1)))$ **in** *exI*) **by** *auto*
next
assume *?rhs*
then obtain $f{::}nat \Rightarrow real\hat{}'a$ **where** $f{:}(\forall n.\ f\ n \in S - \{x\})\ (\forall e{>}0.\ \exists N.\ \forall n{\geq}N.\ dist\ (f\ n)\ x < e)$ **unfolding** *Lim-sequentially* **by** *auto*
{ fix $e{::}real$ **assume** $e{>}0$
then obtain *N* **where** $dist\ (f\ N)\ x < e$ **using** *f*(*2*) **by** *auto*
moreover have $f\ N{\in}S\ f\ N \neq x$ **using** *f*(*1*) **by** *auto*
ultimately have $\exists x'{\in}S.\ x' \neq x \wedge dist\ x'\ x < e$ **by** *auto*
}
thus *?lhs* **unfolding** *islimpt-approachable* **by** *auto*
qed

Basic arithmetical combining theorems for limits.

lemma *Lim-linear*: **fixes** $f :: ('a \Rightarrow real\hat{}'n{::}finite)$ **and** $h :: (real\hat{}'n \Rightarrow real\hat{}'m{::}finite)$
assumes $(f \longrightarrow l)$ *net linear h*
shows $((\lambda x.\ h\ (f\ x)) \longrightarrow h\ l)$ *net*
proof (*cases trivial-limit net*)
case *True*
thus *?thesis* **unfolding** *tendsto-def* **unfolding** *eventually-def* **by** *auto*
next
case *False* **note** *cas* = *this*
obtain *b* **where** *b*: $b{>}0\ \forall x.\ norm\ (h\ x) \leq b * norm\ x$ **using** *assms*(*2*) **using** *linear-bounded-pos*[*of h*] **by** *auto*
{ fix $e{::}real$ **assume** $e > 0$
hence $e/b > 0$ **using** ‹$b{>}0$› **by** (*metis divide-pos-pos*)
then have $(\exists y.\ (\exists x.\ netord\ net\ x\ y) \wedge (\forall x.\ netord\ net\ x\ y \longrightarrow dist\ (f\ x)\ l < e/b))$ **using** *assms* ‹$e{>}0$› *cas*
unfolding *tendsto-def* **unfolding** *eventually-def* **by** *auto*
then obtain *y* **where** *y*: $\exists x.\ netord\ net\ x\ y\ \forall x.\ netord\ net\ x\ y \longrightarrow dist\ (f\ x)\ l < e/b$ **by** *auto*
{ fix *x*
have $netord\ net\ x\ y \longrightarrow dist\ (h\ (f\ x))\ (h\ l) < e$
using *y*(*2*) *b* **unfolding** *dist-def* **using** *linear-sub*[*of h f x l*] ‹*linear h*›
apply *auto* **by** (*metis b*(*1*) *b*(*2*) *dist-def dist-sym less-le-not-le linorder-not-le mult-imp-div-pos-le real-mult-commute xt1*(*7*))
}
hence $(\exists y.\ (\exists x.\ netord\ net\ x\ y) \wedge (\forall x.\ netord\ net\ x\ y \longrightarrow dist\ (h\ (f\ x))\ (h\ l) < e))$ **using** *y*
by(*rule-tac x*=*y* **in** *exI*) *auto*
}
thus *?thesis* **unfolding** *tendsto-def eventually-def* **using** ‹$b{>}0$› **by** *auto*
qed

lemma *Lim-const*: $((\lambda x.\ a) \longrightarrow a)$ *net*
by (*auto simp add*: *Lim dist-refl trivial-limit-def*)

lemma *Lim-cmul*: $(f \longrightarrow l)\ net \Longrightarrow ((\lambda x.\ c *s f\ x) \longrightarrow c *s l)\ net$
apply (*rule Lim-linear*[**where** $f = f$])
apply *simp*
apply (*rule linear-compose-cmul*)
apply (*rule linear-id*[*unfolded id-def*])
done

lemma *Lim-neg*: $(f \longrightarrow l)\ net \Longrightarrow ((\lambda x.\ -(f\ x)) \longrightarrow -l)\ net$
apply (*simp add*: *Lim dist-def group-simps*)
apply (*subst minus-diff-eq*[*symmetric*])
unfolding *norm-minus-cancel* **by** *simp*

lemma *Lim-add*: **fixes** $f :: 'a \Rightarrow real\hat{}'n$::*finite* **shows**
$(f \longrightarrow l)\ net \Longrightarrow (g \longrightarrow m)\ net \Longrightarrow ((\lambda x.\ f(x) + g(x)) \longrightarrow l + m)\ net$
proof−
assume *as*:$(f \longrightarrow l)\ net\ (g \longrightarrow m)\ net$
{ fix *e*::*real*
assume *e*>*0*
hence ∗:*eventually* $(\lambda x.\ dist\ (f\ x)\ l < e/2)\ net$
eventually $(\lambda x.\ dist\ (g\ x)\ m < e/2)\ net$ **using** *as*
by (*auto intro*: *tendstoD simp del*: *less-divide-eq-number-of1*)
hence *eventually* $(\lambda x.\ dist\ (f\ x + g\ x)\ (l + m) < e)\ net$
proof(*cases trivial-limit net*)
case *True*
thus *?thesis* **unfolding** *eventually-def* **by** *auto*
next
case *False*
hence *fl*:$(\exists y.\ (\exists x.\ netord\ net\ x\ y) \wedge (\forall x.\ netord\ net\ x\ y \longrightarrow dist\ (f\ x)\ l < e\ /\ 2))$ **and**
gl:$(\exists y.\ (\exists x.\ netord\ net\ x\ y) \wedge (\forall x.\ netord\ net\ x\ y \longrightarrow dist\ (g\ x)\ m < e\ /\ 2))$
using ∗ **unfolding** *eventually-def* **by** *auto*
obtain *c* **where** *c*:$(\exists x.\ netord\ net\ x\ c)\ (\forall x.\ netord\ net\ x\ c \longrightarrow dist\ (f\ x)\ l < e\ /\ 2 \wedge dist\ (g\ x)\ m < e\ /\ 2)$
using *net-dilemma*[*of net*, *OF fl gl*] **by** *auto*
{ fix *x* **assume** *netord net x c*
with *c*(*2*) **have** *dist* $(f\ x + g\ x)\ (l + m) < e$ **using** *dist-triangle-add*[*of f x g x l m*] **by** *auto*
}
with *c* **show** *?thesis* **unfolding** *eventually-def* **by** *auto*
qed
}
thus *?thesis* **unfolding** *tendsto-def* **by** *auto*
qed

lemma *Lim-sub*: $(f \longrightarrow l)\ net \Longrightarrow (g \longrightarrow m)\ net \Longrightarrow ((\lambda x.\ f(x) - g(x)) \longrightarrow l - m)\ net$

unfolding *diff-minus*
by (*simp add*: *Lim-add Lim-neg*)

lemma *Lim-null*: (*f* ---> *l*) *net* ⟷ ((λ*x*. *f*(*x*) − *l*) ---> 0) *net* **by** (*simp add*: *Lim dist-def*)
lemma *Lim-null-norm*: (*f* ---> 0) *net* ⟷ ((λ*x*. *vec1*(*norm*(*f x*))) ---> 0) *net*
by (*simp add*: *Lim dist-def norm-vec1*)

lemma *Lim-null-comparison*:
assumes *eventually* (λ*x*. *norm*(*f x*) <= *g x*) *net* ((λ*x*. *vec1*(*g x*)) ---> 0) *net*
shows (*f* ---> 0) *net*
proof(*simp add*: *tendsto-def*, *rule*+)
fix *e*::*real* **assume** *0*<*e*
{ fix *x*
assume *norm* (*f x*) ≤ *g x dist* (*vec1* (*g x*)) *0* < *e*
hence *dist* (*f x*) *0* < *e* **unfolding** *vec-def* **using** *dist-vec1*[*of g x 0*]
by (*vector dist-def norm-vec1 dist-refl real-vector-norm-def dot-def vec1-def*)
}
thus *eventually* (λ*x*. *dist* (*f x*) *0* < *e*) *net*
using *eventually-and*[*of* λ*x*. *norm*(*f x*) <= *g x* λ*x*. *dist* (*vec1* (*g x*)) *0* < *e net*]
using *eventually-mono*[*of* (λ*x*. *norm* (*f x*) ≤ *g x* ∧ *dist* (*vec1* (*g x*)) *0* < *e*) (λ*x*. *dist* (*f x*) *0* < *e*) *net*]
using *assms* ‹*e*>*0*› **unfolding** *tendsto-def* **by** *auto*
qed

lemma *Lim-component*: (*f* ---> *l*) *net*
==> ((λ*a*. *vec1*((*f a* :: *real* ^'*n*::*finite*)$*i*)) ---> *vec1*(*l*$*i*)) *net*
apply (*simp add*: *Lim dist-def vec1-sub*[*symmetric*] *norm-vec1 vector-minus-component*[*symmetric*] *del*: *vector-minus-component*)
apply (*auto simp del*: *vector-minus-component*)
apply (*erule-tac x*=*e* **in** *allE*)
apply *clarify*
apply (*rule-tac x*=*y* **in** *exI*)
apply (*auto simp del*: *vector-minus-component*)
apply (*rule order-le-less-trans*)
apply (*rule component-le-norm*)
by *auto*

lemma *Lim-transform-bound*:
assumes *eventually* (λ*n*. *norm*(*f n*) <= *norm*(*g n*)) *net* (*g* ---> 0) *net*
shows (*f* ---> 0) *net*
proof(*simp add*: *tendsto-def*, *rule*+)
fix *e*::*real* **assume** *e*>*0*
{ fix *x*
assume *norm* (*f x*) ≤ *norm* (*g x*) *dist* (*g x*) *0* < *e*
hence *dist* (*f x*) *0* < *e* **by** *norm***}**
thus *eventually* (λ*x*. *dist* (*f x*) *0* < *e*) *net*
using *eventually-and*[*of* λ*x*. *norm* (*f x*) ≤ *norm* (*g x*) λ*x*. *dist* (*g x*) *0* < *e net*]

using *eventually-mono*[*of* $\lambda x.\ norm\ (f\ x) \leq norm\ (g\ x) \wedge dist\ (g\ x)\ 0 < e\ \lambda x.\ dist\ (f\ x)\ 0 < e$ *net*]
using *assms* ‹*e*>*0*› **unfolding** *tendsto-def* **by** *blast*
qed

Deducing things about the limit from the elements.

lemma *Lim-in-closed-set*:
assumes *closed S eventually* $(\lambda x.\ f(x) \in S)$ *net* $\neg$(*trivial-limit net*) (f ---> l) *net*
shows $l \in S$
proof−
{ assume $l \notin S$
obtain *e* **where** *e*:*e*>*0 ball l e* $\subseteq$ *UNIV* − *S* **using** *assms*(*1*) ‹$l \notin S$› **unfolding** *closed-def open-contains-ball* **by** *auto*
hence ∗:$\forall x.\ dist\ l\ x < e \longrightarrow x \notin S$ **by** *auto*
obtain *y* **where** $(\exists x.\ netord\ net\ x\ y) \wedge (\forall x.\ netord\ net\ x\ y \longrightarrow dist\ (f\ x)\ l < e)$
using *assms*(*3*,*4*) ‹*e*>*0*› **unfolding** *tendsto-def eventually-def* **by** *blast*
hence $(\exists x.\ netord\ net\ x\ y) \wedge (\forall x.\ netord\ net\ x\ y \longrightarrow f\ x \notin S)$ **using** ∗ **by** (*auto simp add*: *dist-sym*)
hence *False* **using** *assms*(*2*,*3*)
using *eventually-and*[*of* $(\lambda x.\ f\ x \in S)\ (\lambda x.\ f\ x \notin S)$] *not-eventually*[*of* $(\lambda x.\ f\ x \in S \wedge f\ x \notin S)$ *net*]
unfolding *eventually-def* **by** *blast*
}
thus *?thesis* **by** *blast*
qed

Need to prove closed(cball(x,e)) before deducing this as a corollary.

lemma *Lim-norm-ubound*:
assumes $\neg$(*trivial-limit net*) (f ---> l) *net eventually* $(\lambda x.\ norm(f\ x) <= e)$ *net*
shows $norm(l) <= e$
proof−
obtain *y* **where** *y*: $\exists x.\ netord\ net\ x\ y\ \ \forall x.\ netord\ net\ x\ y \longrightarrow norm\ (f\ x) \leq e$ **using** *assms*(*1*,*3*) **unfolding** *eventually-def* **by** *auto*
show *?thesis*
proof(*rule ccontr*)
assume $\neg\ norm\ l \leq e$
then obtain *z* **where** *z*: $\exists x.\ netord\ net\ x\ z\ \ \forall x.\ netord\ net\ x\ z \longrightarrow dist\ (f\ x)\ l < norm\ l - e$
using *assms*(*2*)[*unfolded Lim*] **using** *assms*(*1*) **apply** *simp* **apply**(*erule-tac x*=*norm l* − *e* **in** *allE*) **by** *auto*
obtain *w* **where** *w*:*netord net w z netord net w y* **using** *net*[*of net*] **using** *z*(*1*) *y*(*1*) **by** *blast*
hence $dist\ (f\ w)\ l < norm\ l - e \wedge norm\ (f\ w) <= e$ **using** *z*(*2*) *y*(*2*) **by** *auto*
thus *False* **using** ‹$\neg\ norm\ l \leq e$› **by** *norm*
qed
qed

lemma *Lim-norm-lbound*:
assumes ¬ (*trivial-limit net*) (*f* ---> *l*) *net* *eventually* (λ*x*. *e* <= *norm*(*f x*)) *net*
shows *e* ≤ *norm l*
proof−
obtain *y* **where** *y*: ∃ *x*. *netord net x y* ∀ *x*. *netord net x y* ⟶ *e* ≤ *norm* (*f x*) **using** *assms*(*1*,*3*) **unfolding** *eventually-def* **by** *auto*
show *?thesis*
proof(*rule ccontr*)
assume ¬ *e* ≤ *norm l*
then obtain *z* **where** *z*: ∃ *x*. *netord net x z* ∀ *x*. *netord net x z* ⟶ *dist* (*f x*) *l* < *e* − *norm l*
using *assms*(*2*)[*unfolded Lim*] **using** *assms*(*1*) **apply** *simp* **apply**(*erule-tac x*=*e* − *norm l* **in** *allE*) **by** *auto*
obtain *w* **where** *w*:*netord net w z* *netord net w y* **using** *net*[*of net*] **using** *z*(*1*) *y*(*1*) **by** *blast*
hence *dist* (*f w*) *l* < *e* − *norm l* ∧ *e* ≤ *norm* (*f w*) **using** *z*(*2*) *y*(*2*) **by** *auto*
thus *False* **using** ‹¬ *e* ≤ *norm l*› **by** *norm*
qed
qed

Uniqueness of the limit, when nontrivial.

lemma *Lim-unique*:
fixes *l*::*real*^'*a*::*finite* **and** *net*::'*b*::*zero-neq-one net*
assumes ¬(*trivial-limit net*) (*f* ---> *l*) *net* (*f* ---> *l'*) *net*
shows *l* = *l'*
proof−
{ **fix** *e*::*real* **assume** *e*>*0*
hence *eventually* (λ*x*. *norm* (*0*::*real*^'*a*) ≤ *e*) *net* **unfolding** *norm-0* **using** *always-eventually*[*of - net*] **by** *auto*
hence *norm* (*l* − *l'*) ≤ *e* **using** *Lim-norm-ubound*[*of net* λ*x*. *0 l*−*l'*] **using** *assms* **using** *Lim-sub*[*of f l net f l'*] **by** *auto*
} **note** ∗ = *this*
{ **assume** *norm* (*l* − *l'*) > *0*
hence *norm* (*l* − *l'*) = *0* **using** ∗[*of* (*norm* (*l* − *l'*)) */2*] **using** *norm-ge-zero*[*of l* − *l'*] **by** *simp*
}
hence *l* = *l'* **using** *norm-ge-zero*[*of l* − *l'*] **unfolding** *le-less* **and** *dist-nz*[*of l l'*, *unfolded dist-def*, *THEN sym*] **by** *auto*
thus *?thesis* **using** *assms* **using** *Lim-sub*[*of f l net f l'*] **by** *simp*
qed

lemma *tendsto-Lim*:
~(*trivial-limit* (*net*::('*b*::*zero-neq-one net*))) ⟹ (*f* ---> *l*) *net* ==> *Lim net f* = *l*
unfolding *Lim-def* **using** *Lim-unique*[*of net f*] **by** *auto*

Limit under bilinear function (surprisingly tedious, but important)

lemma *norm-bound-lemma*:

*0 < e ⟹ ∃ d>0. ∀ (x'::real^'b::finite) y'::real^'a::finite. norm(x' − (x::real^'b)) < d ∧ norm(y' − y) < d ⟶ norm(x') * norm(y' − y) + norm(x' − x) * norm(y) < e*
proof −
assume *e*: *0 < e*
have *th1*: *(2 * norm x + 2 * norm y + 2) > 0* **using** *norm-ge-zero*[*of x*] *norm-ge-zero*[*of y*] **by** *norm*
hence *th0*: *0 < e / (2 * norm x + 2 * norm y + 2)* **using** ‹*e>0*› **using** *divide-pos-pos* **by** *auto*
moreover
{ fix *x' y'*
assume *h*: *norm (x' − x) < 1 norm (x' − x) < e / (2 * norm x + 2 * norm y + 2)*
*norm (y' − y) < 1 norm (y' − y) < e / (2 * norm x + 2 * norm y + 2)*
have *th*: *⋀a b (c::real). a ≥ 0 ⟹ c ≥ 0 ⟹ a + (b + c) < e ==> b < e* **by** *arith*
from *h* **have** *thx*: *norm (x' − x) * norm y < e / 2*
using *th0 th1* **apply** (*simp add*: *field-simps*)
apply (*rule th*) **defer defer apply** *assumption*
by (*simp-all add*: *norm-ge-zero zero-le-mult-iff*)

have *norm x' − norm x < 1* **apply**(*rule le-less-trans*)
using *h*(*1*) **using** *norm-triangle-ineq2*[*of x' x*] **by** *auto*
hence *:norm x' < 1 + norm x* **by** *auto*

have *thy*: *norm (y' − y) * norm x' < e / (2 * norm x + 2 * norm y + 2) * (1 + norm x)*
using *mult-strict-mono'*[*OF h*(*4*) * *norm-ge-zero norm-ge-zero*] **by** *auto*
also have *... ≤ e/2* **apply** *simp* **unfolding** *divide-le-eq*
using *th1 th0* ‹*e>0*› **by** *auto*

finally have *norm x' * norm (y' − y) + norm (x' − x) * norm y < e*
using *thx* **and** *e* **by** (*simp add*: *field-simps*) **}**
ultimately show *?thesis* **apply**(*rule-tac x=min 1 (e / 2 / (norm x + norm y + 1))* **in** *exI*) **by** *auto*
qed

lemma *Lim-bilinear*:
fixes *net* :: *'a net* **and** *h*:: *real ^'m::finite ⇒ real ^'n::finite ⇒ real ^'p::finite*
assumes *(f ---> l) net* **and** *(g ---> m) net* **and** *bilinear h*
shows *((λx. h (f x) (g x)) ---> (h l m)) net*
proof(*cases trivial-limit net*)
case *True* **thus** *((λx. h (f x) (g x)) ---> h l m) net* **unfolding** *Lim* **..**
next
case *False* **note** *ntriv = this*
obtain *B* **where** *B>0* **and** *B:∀ x y. norm (h x y) ≤ B * norm x * norm y* **using** *bilinear-bounded-pos*[*OF assms*(*3*)] **by** *auto*
{ fix *e::real* **assume** *e>0*
obtain *d* **where** *d>0* **and** *d:∀ x' y'. norm (x' − l) < d ∧ norm (y' − m) < d*

$\longrightarrow$ *norm* x' $*$ *norm* $(y' - m)$ $+$ *norm* $(x' - l)$ $*$ *norm* $m < e$ / B **using** ‹$B>0$› ‹$e>0$›
using *norm-bound-lemma*[*of* e / B l m] **using** *divide-pos-pos* **by** *auto*

have $*$:$\bigwedge x\ y.\ h\ (f\ x)\ (g\ x) - h\ l\ m = h\ (f\ x)\ (g\ x - m) + h\ (f\ x - l)\ m$
unfolding *bilinear-rsub*[*OF assms*(*3*)]
unfolding *bilinear-lsub*[*OF assms*(*3*)] **by** *auto*

{ fix x **assume** *dist* $(f\ x)\ l < d \wedge$ *dist* $(g\ x)\ m < d$
hence $**$:*norm* $(f\ x) *$ *norm* $(g\ x - m) +$ *norm* $(f\ x - l) *$ *norm* $m < e$ / B
using d[*THEN spec*[**where** x=$f\ x$], *THEN spec*[**where** x=$g\ x$]] **unfolding** *dist-def* **by** *auto*
have *norm* $(h\ (f\ x)\ (g\ x - m)) +$ *norm* $(h\ (f\ x - l)\ m) \leq B *$ *norm* $(f\ x) *$ *norm* $(g\ x - m) + B *$ *norm* $(f\ x - l) *$ *norm* m
using B[*THEN spec*[**where** x=$f\ x$], *THEN spec*[**where** x=$g\ x - m$]]
using B[*THEN spec*[**where** x=$f\ x - l$], *THEN spec*[**where** x=m]] **by** *auto*
also have $\ldots < e$ **using** $**$ **and** ‹$B>0$› **by**(*auto simp add*: *field-simps*)
finally have *dist* $(h\ (f\ x)\ (g\ x))\ (h\ l\ m) < e$ **unfolding** *dist-def* **and** $*$ **using** *norm-triangle-lt* **by** *auto*
}
moreover
obtain c **where** $(\exists\, x.$ *netord net* $x\ c) \wedge (\forall\, x.$ *netord net* $x\ c \longrightarrow$ *dist* $(f\ x)\ l < d \wedge$ *dist* $(g\ x)\ m < d)$
using *net-dilemma*[*of net* $\lambda x.$ *dist* $(f\ x)\ l < d$ $\lambda x.$ *dist* $(g\ x)\ m < d$] **using** *assms*(*1*,*2*) **unfolding** *Lim* **using** *False* **and** ‹$d>0$› **by** (*auto elim*!: *allE*[**where** x=d])
ultimately have $\exists\, y.\ (\exists\, x.$ *netord net* $x\ y) \wedge (\forall\, x.$ *netord net* $x\ y \longrightarrow$ *dist* $(h\ (f\ x)\ (g\ x))\ (h\ l\ m) < e)$ **by** *auto* **}**
thus $((\lambda x.\ h\ (f\ x)\ (g\ x)) \dashrightarrow h\ l\ m)$ *net* **unfolding** *Lim* **by** *auto*
qed

These are special for limits out of the same vector space.

lemma *Lim-within-id*: $(id \dashrightarrow a)$ $(at\ a\ within\ s)$ **by** (*auto simp add*: *Lim-within id-def*)
lemma *Lim-at-id*: $(id \dashrightarrow a)$ $(at\ a)$
apply (*subst within-UNIV*[*symmetric*]) **by** (*simp add*: *Lim-within-id*)

lemma *Lim-at-zero*: $(f \dashrightarrow l)$ $(at\ (a$::*real*^'*a*::*finite*$)) \longleftrightarrow ((\lambda x.\ f(a + x)) \dashrightarrow l)$ $(at\ 0)$ (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ fix e::*real* **assume** $e>0$
with ‹*?lhs*› **obtain** d **where** d:$d>0\ \forall\, x.\ 0 <$ *dist* $x\ a \wedge$ *dist* $x\ a < d \longrightarrow$ *dist* $(f\ x)\ l < e$ **unfolding** *Lim-at* **by** *auto*
{ fix x::*real*^'*a* **assume** $0 <$ *dist* $x\ 0 \wedge$ *dist* $x\ 0 < d$
hence *dist* $(f\ (a + x))\ l < e$ **using** d
apply(*erule-tac* x=x+a **in** *allE*) **by**(*auto simp add*: *comm-monoid-add.mult-commute dist-def dist-sym*)
}

hence $\exists\, d{>}0.\ \forall x.\ 0 < dist\ x\ 0 \wedge dist\ x\ 0 < d \longrightarrow dist\ (f\ (a + x))\ l < e$ **using** *d(1)* **by** *auto*
}
thus *?rhs* **unfolding** *Lim-at* **by** *auto*
next
assume *?rhs*
{ **fix** *e*::*real* **assume** *e>0*
with ‹*?rhs*› **obtain** *d* **where** *d*:*d>0* $\forall x.\ 0 < dist\ x\ 0 \wedge dist\ x\ 0 < d \longrightarrow dist\ (f\ (a + x))\ l < e$
unfolding *Lim-at* **by** *auto*
{ **fix** *x*::*real*^'*a* **assume** $0 < dist\ x\ a \wedge dist\ x\ a < d$
hence $dist\ (f\ x)\ l < e$ **using** *d* **apply**(*erule-tac x=x−a* **in** *allE*)
by(*auto simp add*: *comm-monoid-add.mult-commute dist-def dist-sym*)
}
hence $\exists\, d{>}0.\ \forall x.\ 0 < dist\ x\ a \wedge dist\ x\ a < d \longrightarrow dist\ (f\ x)\ l < e$ **using** *d(1)* **by** *auto*
}
thus *?lhs* **unfolding** *Lim-at* **by** *auto*
qed

It's also sometimes useful to extract the limit point from the net.

definition *netlimit net* = (*SOME a*. $\forall x.\ \sim$(*netord net x a*))

lemma *netlimit-within*: **assumes**$\sim$(*trivial-limit* (*at a within S*))
shows (*netlimit* (*at a within S*) = *a*)
proof−
{ **fix** *x* **assume** $x \neq a$
then obtain *y* **where** *y*:$dist\ y\ a \leq dist\ a\ a \wedge 0 < dist\ y\ a \wedge y \in S \vee dist\ y\ a \leq dist\ x\ a \wedge 0 < dist\ y\ a \wedge y \in S$ **using** *assms* **unfolding** *trivial-limit-def within at* **by** *blast*
assume $\forall y.\ \neg\ netord\ (at\ a\ within\ S)\ y\ x$
hence $x = a$ **using** *y* **unfolding** *within at* **by** (*auto simp add*: *dist-refl dist-nz*)
}
moreover
have $\forall y.\ \neg\ netord\ (at\ a\ within\ S)\ y\ a$ **using** *assms* **unfolding** *trivial-limit-def within at* **by** (*auto simp add*: *dist-refl*)
ultimately show *?thesis* **unfolding** *netlimit-def* **using** *some-equality*[*of* $\lambda x.\ \forall y.\ \neg\ netord\ (at\ a\ within\ S)\ y\ x$] **by** *blast*
qed

lemma *netlimit-at*: *netlimit*(*at a*) = *a*
apply (*subst within-UNIV*[*symmetric*])
using *netlimit-within*[*of a UNIV*]
by (*simp add*: *trivial-limit-at within-UNIV*)

Transformation of limit.

lemma *Lim-transform*: **assumes** $((\lambda x.\ f\ x - g\ x) \dashrightarrow 0)\ net\ (f \dashrightarrow l)\ net$
shows $(g \dashrightarrow l)\ net$
proof−

from *assms* **have** $((\lambda x.\ f\ x - g\ x - f\ x) \longrightarrow 0 - l)\ net$ **using** *Lim-sub*[*of* $\lambda x.\ f\ x - g\ x\ 0\ net\ f\ l$] **by** *auto*
thus *?thesis* **using** *Lim-neg* [*of* $\lambda\ x.\ -\ g\ x\ -l\ net$] **by** *auto*
qed

lemma *Lim-transform-eventually*: $eventually\ (\lambda x.\ f\ x = g\ x)\ net \Longrightarrow (f \longrightarrow l)\ net \Longrightarrow (g \longrightarrow l)\ net$
using *Lim-eventually*[*of* $\lambda x.\ f\ x - g\ x\ 0\ net$] *Lim-transform*[*of f g net l*] **by** *auto*

lemma *Lim-transform-within*:
assumes $0 < d\ (\forall x' \in S.\ 0 < dist\ x'\ x \wedge dist\ x'\ x < d \longrightarrow f\ x' = g\ x')$
$(f \longrightarrow l)\ (at\ x\ within\ S)$
shows $(g \longrightarrow l)\ (at\ x\ within\ S)$
proof−
have $((\lambda x.\ f\ x - g\ x) \longrightarrow 0)\ (at\ x\ within\ S)$ **unfolding** *Lim-within*[*of* $\lambda x.\ f\ x - g\ x\ 0\ x\ S$] **using** *assms*(*1*,*2*) **by** *auto*
thus *?thesis* **using** *Lim-transform*[*of f g at x within S l*] **using** *assms*(*3*) **by** *auto*
qed

lemma *Lim-transform-at*: $0 < d \Longrightarrow (\forall x'.\ 0 < dist\ x'\ x \wedge dist\ x'\ x < d \longrightarrow f\ x' = g\ x') \Longrightarrow$
$(f \longrightarrow l)\ (at\ x) \Longrightarrow (g \longrightarrow l)\ (at\ x)$
apply (*subst within-UNIV*[*symmetric*])
using *Lim-transform-within*[*of d UNIV x f g l*]
by (*auto simp add*: *within-UNIV*)

Common case assuming being away from some crucial point like 0.

lemma *Lim-transform-away-within*:
fixes *f*:: *real* ^'*m*::*finite* ⇒ *real* ^'*n*::*finite*
assumes $a \neq b\ \forall x \in S.\ x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$
and $(f \longrightarrow l)\ (at\ a\ within\ S)$
shows $(g \longrightarrow l)\ (at\ a\ within\ S)$
proof−
have $\forall x' \in S.\ 0 < dist\ x'\ a \wedge dist\ x'\ a < dist\ a\ b \longrightarrow f\ x' = g\ x'$ **using** *assms*(*2*)
apply *auto* **apply**(*erule-tac* $x{=}x'$ **in** *ballE*) **by** (*auto simp add*: *dist-sym dist-refl*)
thus *?thesis* **using** *Lim-transform-within*[*of dist a b S a f g l*] **using** *assms*(*1*,*3*)
unfolding *dist-nz* **by** *auto*
qed

lemma *Lim-transform-away-at*:
fixes *f*:: *real* ^'*m*::*finite* ⇒ *real* ^'*n*::*finite*
assumes *ab*: $a \neq b$ **and** *fg*: $\forall x.\ x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$
and *fl*: $(f \longrightarrow l)\ (at\ a)$
shows $(g \longrightarrow l)\ (at\ a)$
using *Lim-transform-away-within*[*OF ab*, *of UNIV f g l*] *fg fl*
by (*auto simp add*: *within-UNIV*)

Alternatively, within an open set.

lemma *Lim-transform-within-open*:

fixes f:: *real ^'m::finite* $\Rightarrow$ *real ^'n::finite*
assumes *open S* $a \in S$ $\forall x{\in}S.\ x \neq a \longrightarrow f\ x = g\ x$ $(f \longrightarrow l)\ (at\ a)$
shows $(g \longrightarrow l)\ (at\ a)$
proof−
from *assms*(*1*,*2*) **obtain** *e*::*real* **where** *e*>*0* **and** *e*:*ball a e* $\subseteq S$ **unfolding** *open-contains-ball* **by** *auto*
hence $\forall x'.\ 0 < dist\ x'\ a \wedge dist\ x'\ a < e \longrightarrow f\ x' = g\ x'$ **using** *assms*(*3*)
unfolding *ball-def subset-eq* **apply** *auto* **apply**(*erule-tac x*=*x'* **in** *allE*) **apply**(*erule-tac x*=*x'* **in** *ballE*) **by**(*auto simp add*: *dist-refl dist-sym*)
thus *?thesis* **using** *Lim-transform-at*[*of e a f g l*] ‹*e*>*0*› *assms*(*4*) **by** *auto*
qed

A congruence rule allowing us to transform limits assuming not at point.

lemma *Lim-cong-within*[*cong add*]:
$(\bigwedge x.\ x \neq a \Longrightarrow f\ x = g\ x) ==> ((\lambda x.\ f\ x) \longrightarrow l)\ (at\ a\ within\ S) \longleftrightarrow ((g \longrightarrow l)\ (at\ a\ within\ S))$
by (*simp add*: *Lim-within dist-nz*[*symmetric*])

lemma *Lim-cong-at*[*cong add*]:
$(\bigwedge x.\ x \neq a ==> f\ x = g\ x) ==> (((\lambda x.\ f\ x) \longrightarrow l)\ (at\ a) \longleftrightarrow ((g \longrightarrow l)\ (at\ a)))$
by (*simp add*: *Lim-at dist-nz*[*symmetric*])

Useful lemmas on closure and set of possible sequential limits.

lemma *closure-sequential*:
$l \in closure\ S \longleftrightarrow (\exists x.\ (\forall n.\ x\ n \in S) \wedge (x \longrightarrow l)\ sequentially)$ (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **moreover**
{ **assume** $l \in S$
hence *?rhs* **using** *Lim-const*[*of l sequentially*] **by** *auto*
} **moreover**
{ **assume** *l islimpt S*
hence *?rhs* **unfolding** *islimpt-sequential* **by** *auto*
} **ultimately**
show *?rhs* **unfolding** *closure-def* **by** *auto*
next
assume *?rhs*
thus *?lhs* **unfolding** *closure-def* **unfolding** *islimpt-sequential* **by** *auto*
qed

lemma *closed-sequential-limits*:
closed S $\longleftrightarrow$ $(\forall x\ l.\ (\forall n.\ x\ n \in S) \wedge (x \longrightarrow l)\ sequentially \longrightarrow l \in S)$
unfolding *closed-limpt*
by (*metis closure-sequential closure-closed closed-limpt islimpt-sequential mem-delete*)

lemma *closure-approachable*: $x \in closure\ S \longleftrightarrow (\forall e{>}0.\ \exists y{\in}S.\ dist\ y\ x < e)$
apply (*auto simp add*: *closure-def islimpt-approachable*)
by (*metis dist-refl*)

lemma *closed-approachable*: *closed S ==> (∀ e>0. ∃ y∈S. dist y x < e) ⟷ x ∈ S*
by (*metis closure-closed closure-approachable*)

Some other lemmas about sequences.

lemma *seq-offset*: *(f ---> l) sequentially ==> ((λi. f(i + k)) ---> l) sequentially*
apply (*auto simp add*: *Lim-sequentially*)
by (*metis trans-le-add1*)

lemma *seq-offset-neg*: *(f ---> l) sequentially ==> ((λi. f(i − k)) ---> l) sequentially*
apply (*simp add*: *Lim-sequentially*)
apply (*subgoal-tac ⋀N k (n::nat). N + k <= n ==> N <= n − k*)
apply *metis*
by *arith*

lemma *seq-offset-rev*: *((λi. f(i + k)) ---> l) sequentially ==> (f ---> l) sequentially*
apply (*simp add*: *Lim-sequentially*)
apply (*subgoal-tac ⋀N k (n::nat). N + k <= n ==> N <= n − k ∧ (n − k) + k = n*)
by *metis arith*

lemma *seq-harmonic*: *((λn. vec1(inverse (real n))) ---> 0) sequentially*
proof−
{ fix *e::real* **assume** *e>0*
hence *∃ N::nat. ∀ n::nat≥N. inverse (real n) < e*
using *real-arch-inv*[*of e*] **apply** *auto* **apply**(*rule-tac x=n* **in** *exI*)
by (*metis dlo-simps*(*4*) *le-imp-inverse-le linorder-not-less real-of-nat-gt-zero-cancel-iff real-of-nat-less-iff xt1*(*7*))
}
thus *?thesis* **unfolding** *Lim-sequentially dist-def* **apply** *simp* **unfolding** *norm-vec1*
by *auto*
qed

More properties of closed balls.

lemma *closed-cball*: *closed*(*cball x e*)
proof−
{ fix *xa::nat⇒real^'a* **and** *l* **assume** *as*: *∀ n. dist x (xa n) ≤ e (xa ---> l) sequentially*
from *as*(*2*) **have** *((λn. x − xa n) ---> x − l) sequentially* **using** *Lim-sub*[*of λn. x x sequentially xa l*] *Lim-const*[*of x sequentially*] **by** *auto*
moreover from *as*(*1*) **have** *eventually (λn. norm (x − xa n) ≤ e) sequentially* **unfolding** *eventually-sequentially dist-def* **by** *auto*
ultimately have *dist x l ≤ e*
unfolding *dist-def*
using *Lim-norm-ubound*[*of sequentially - x − l e*] **using** *trivial-limit-sequentially*
by *auto*

}
thus *?thesis* **unfolding** *closed-sequential-limits* **by** *auto*
qed

lemma *open-contains-cball*: *open S* $\longleftrightarrow$ $(\forall x{\in}S.\ \exists e{>}0.\ \ cball\ x\ e \subseteq S)$
proof−
{ fix *x* **and** *e*::*real* **assume** $x{\in}S$ $e{>}0$ *ball x e* $\subseteq S$
hence $\exists d{>}0.$ *cball x d* $\subseteq S$ **unfolding** *subset-eq* **by** (*rule-tac x*=*e*/2 **in** *exI*, *auto*)
} moreover
{ fix *x* **and** *e*::*real* **assume** $x{\in}S$ $e{>}0$ *cball x e* $\subseteq S$
hence $\exists d{>}0.$ *ball x d* $\subseteq S$ **unfolding** *subset-eq* **apply**(*rule-tac x*=*e*/2 **in** *exI*) **by** *auto*
} ultimately
show *?thesis* **unfolding** *open-contains-ball* **by** *auto*
qed

lemma *open-contains-cball-eq*: *open S* ==> $(\forall x.\ x \in S \longleftrightarrow (\exists e{>}0.\ cball\ x\ e \subseteq S))$
by (*metis open-contains-cball subset-eq order-less-imp-le centre-in-cball mem-def*)

lemma *mem-interior-cball*: $x \in$ *interior S* $\longleftrightarrow$ $(\exists e{>}0.\ cball\ x\ e \subseteq S)$
apply (*simp add*: *interior-def*)
by (*metis open-contains-cball subset-trans ball-subset-cball centre-in-ball open-ball*)

lemma *islimpt-ball*: *y islimpt ball x e* $\longleftrightarrow$ $0 < e \wedge y \in cball\ x\ e$ (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ assume $e \leq 0$
hence $*$:*ball x e* = {} **using** *ball-eq-empty*[*of x e*] **by** *auto*
have *False* **using** ‹*?lhs*› **unfolding** $*$ **using** *islimpt-EMPTY*[*of y*] **by** *auto*
}
hence $e > 0$ **by** (*metis dlo-simps*(3))
moreover
have $y \in cball\ x\ e$ **using** *closed-cball*[*of x e*] *islimpt-subset*[*of y ball x e cball x e*] *ball-subset-cball*[*of x e*] ‹*?lhs*› **unfolding** *closed-limpt* **by** *auto*
ultimately show *?rhs* **by** *auto*
next
assume *?rhs* **hence** *e*>*0* **by** *auto*
{ fix *d*::*real* **assume** *d*>*0*
have $\exists x'{\in}ball\ x\ e.\ x' \neq y \wedge dist\ x'\ y < d$
proof(*cases* $d \leq dist\ x\ y$)
case *True* **thus** $\exists x'{\in}ball\ x\ e.\ x' \neq y \wedge dist\ x'\ y < d$
proof(*cases x*=*y*)
case *True* **hence** *False* **using** ‹$d \leq dist\ x\ y$› ‹*d*>*0*› *dist-refl*[*of x*] **by** *auto*
thus $\exists x'{\in}ball\ x\ e.\ x' \neq y \wedge dist\ x'\ y < d$ **by** *auto*
next
case *False*

```
      have dist x (y − (d / (2 ∗ dist y x)) ∗s (y − x))
           = norm (x − y + (d / (2 ∗ norm (y − x))) ∗s (y − x))
       unfolding mem-cball mem-ball dist-def diff-diff-eq2 diff-add-eq[THEN sym]
by auto
      also have ... = |− 1 + d / (2 ∗ norm (x − y))| ∗ norm (x − y)
       using vector-sadd-rdistrib[of − 1 d / (2 ∗ norm (y − x)), THEN sym, of
y − x]
        unfolding vector-smult-lneg vector-smult-lid
        by (auto simp add: dist-sym[unfolded dist-def] norm-mul)
      also have ... = |− norm (x − y) + d / 2|
        unfolding abs-mult-pos[of norm (x − y), OF norm-ge-zero[of x − y]]
         unfolding real-add-mult-distrib using ‹x≠y›[unfolded dist-nz, unfolded
dist-def] by auto
        also have ... ≤ e − d/2 using ‹d ≤ dist x y› and ‹d>0› and ‹?rhs›
by(auto simp add: dist-def)
      finally have y − (d / (2 ∗ dist y x)) ∗s (y − x) ∈ ball x e using ‹d>0›
by auto

      moreover

      have (d / (2∗dist y x)) ∗s (y − x) ≠ 0
        using ‹x≠y›[unfolded dist-nz] ‹d>0› unfolding vector-mul-eq-0 by (auto
simp add: dist-sym dist-refl)
      moreover
      have dist (y − (d / (2 ∗ dist y x)) ∗s (y − x)) y < d unfolding dist-def
apply simp unfolding norm-minus-cancel norm-mul
        using ‹d>0› ‹x≠y›[unfolded dist-nz] dist-sym[of x y]
        unfolding dist-def by auto
      ultimately show ∃x′∈ball x e. x′ ≠ y ∧ dist x′ y < d by (rule-tac x=y
− (d / (2∗dist y x)) ∗s (y − x) in bexI) auto
    qed
  next
    case False hence d > dist x y by auto
    show ∃x′∈ball x e. x′ ≠ y ∧ dist x′ y < d
    proof(cases x=y)
      case True
       obtain z where ∗∗:dist y z = (min e d) / 2 using vector-choose-dist[of
(min e d) / 2 y]
        using ‹d > 0› ‹e>0› by (auto simp add: dist-refl)
       show ∃x′∈ball x e. x′ ≠ y ∧ dist x′ y < d
         apply(rule-tac x=z in bexI) unfolding ‹x=y› dist-sym dist-refl dist-nz
using ∗∗ ‹d > 0› ‹e>0› by auto
    next
      case False thus ∃x′∈ball x e. x′ ≠ y ∧ dist x′ y < d
       using ‹d>0› ‹d > dist x y› ‹?rhs› by(rule-tac x=x in bexI, auto simp add:
dist-refl)
    qed
  qed }
 thus ?lhs unfolding mem-cball islimpt-approachable mem-ball by auto
```

qed

lemma *closure-ball*: *0 < e ==> (closure(ball x e) = cball x e)*
apply (*simp add*: *closure-def islimpt-ball expand-set-eq*)
by *arith*

lemma *interior-cball*: *interior(cball x e) = ball x e*
proof(*cases e≥0*)
case *False* **note** *cs = this*
from *cs* **have** *ball x e = {}* **using** *ball-empty*[*of e x*] **by** *auto* **moreover**
{ fix *y* **assume** *y ∈ cball x e*
hence *False* **unfolding** *mem-cball* **using** *dist-nz*[*of x y*] *cs* **by** (*auto simp add*: *dist-refl*) **}**
hence *cball x e = {}* **by** *auto*
hence *interior (cball x e) = {}* **using** *interior-empty* **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case *True* **note** *cs = this*
have *ball x e ⊆ cball x e* **using** *ball-subset-cball* **by** *auto* **moreover**
{ fix *S y* **assume** *as*: *S ⊆ cball x e open S y∈S*
then obtain *d* **where** *d>0* **and** *d*:∀ *x'*. *dist x' y < d ⟶ x' ∈ S* **unfolding** *open-def* **by** *blast*

then obtain *xa* **where** *xa*:*dist y xa = d / 2* **using** *vector-choose-dist*[*of d/2 y*] **by** *auto*
hence *xa-y*:*xa ≠ y* **using** *dist-nz*[*of y xa*] **using** ‹*d>0*› **by** *auto*
have *xa∈S* **using** *d*[*THEN spec*[**where** *x=xa*]] **using** *xa* **apply**(*auto simp add*: *dist-sym*) **unfolding** *dist-nz*[*THEN sym*] **using** *xa-y* **by** *auto*
hence *xa-cball*:*xa ∈ cball x e* **using** *as(1)* **by** *auto*

hence *y ∈ ball x e* **proof**(*cases x = y*)
case *True*
hence *e>0* **using** *xa-y*[*unfolded dist-nz*] *xa-cball*[*unfolded mem-cball*] **by** (*auto simp add*: *dist-sym*)
thus *y ∈ ball x e* **using** ‹*x = y* › **by** *simp*
next
case *False*
have *dist (y + (d / 2 / dist y x) *s (y − x)) y < d* **unfolding** *dist-def*
using ‹*d>0*› *norm-ge-zero*[*of y − x*] ‹*x ≠ y*› **by** *auto*
hence *∗*:*y + (d / 2 / dist y x) *s (y − x) ∈ cball x e* **using** *d as(1)*[*unfolded subset-eq*] **by** *blast*
have *y − x ≠ 0* **using** ‹*x ≠ y*› **by** *auto*
hence *∗∗*:*d / (2 ∗ norm (y − x)) > 0* **unfolding** *zero-less-norm-iff*[*THEN sym*]
using ‹*d>0*› *divide-pos-pos*[*of d 2∗norm (y − x)*] **by** *auto*

have *dist (y + (d / 2 / dist y x) *s (y − x)) x = norm (y + (d / (2 ∗ norm (y − x))) *s y − (d / (2 ∗ norm (y − x))) *s x − x)*
by (*auto simp add*: *dist-def vector-ssub-ldistrib add-diff-eq*)

also have ... = *norm* ((*1* + *d* / (*2* ∗ *norm* (*y* − *x*))) ∗*s* (*y* − *x*))
by (*auto simp add*: *vector-sadd-rdistrib vector-smult-lid ring-simps vector-sadd-rdistrib vector-ssub-ldistrib*)
also have ... = |*1* + *d* / (*2* ∗ *norm* (*y* − *x*))| ∗ *norm* (*y* − *x*) **using** ∗∗ **by** *auto*
also have ... = (*dist y x*) + *d*/*2***using** ∗∗ **by** (*auto simp add*: *left-distrib dist-def*)
finally have *e* ≥ *dist x y* +*d*/*2* **using** ∗[*unfolded mem-cball*] **by** (*auto simp add*: *dist-sym*)
thus *y* ∈ *ball x e* **unfolding** *mem-ball* **using** ‹*d*>*0*› **by** *auto*
qed }
hence ∀ *S* ⊆ *cball x e*. *open S* ⟶ *S* ⊆ *ball x e* **by** *auto*
ultimately show *?thesis* **using** *interior-unique*[*of ball x e cball x e*] **using** *open-ball*[*of x e*] **by** *auto*
qed

lemma *frontier-ball*: *0* < *e* ==> *frontier*(*ball a e*) = {*x*. *dist a x* = *e*}
apply (*simp add*: *frontier-def closure-ball interior-open open-ball order-less-imp-le*)
apply (*simp add*: *expand-set-eq*)
by *arith*

lemma *frontier-cball*: *frontier*(*cball a e*) = {*x*. *dist a x* = *e*}
apply (*simp add*: *frontier-def interior-cball closed-cball closure-closed order-less-imp-le*)
apply (*simp add*: *expand-set-eq*)
by *arith*

lemma *cball-eq-empty*: (*cball x e* = {}) ⟷ *e* < *0*
apply (*simp add*: *expand-set-eq not-le*)
by (*metis dist-pos-le dist-refl order-less-le-trans*)
lemma *cball-empty*: *e* < *0* ==> *cball x e* = {} **by** (*simp add*: *cball-eq-empty*)

lemma *cball-eq-sing*: (*cball x e* = {*x*}) ⟷ *e* = *0*
proof−
{ **assume** *as*:∀ *xa*. (*dist x xa* ≤ *e*) = (*xa* = *x*)
hence *e* ≥ *0* **apply** (*erule-tac x*=*x* **in** *allE*) **by** (*auto simp add*: *dist-pos-le dist-refl*)
then obtain *y* **where** *y*:*dist x y* = *e* **using** *vector-choose-dist*[*of e*] **by** *auto*
hence *e* = *0* **using** *as* **apply**(*erule-tac x*=*y* **in** *allE*) **by** (*auto simp add*: *dist-pos-le dist-refl*)
}
thus *?thesis* **unfolding** *expand-set-eq mem-cball* **by** (*auto simp add*: *dist-refl dist-nz dist-le-0*)
qed

lemma *cball-sing*: *e* = *0* ==> *cball x e* = {*x*} **by** (*simp add*: *cball-eq-sing*)

For points in the interior, localization of limits makes no difference.

lemma *eventually-within-interior*: **assumes** *x* ∈ *interior S*
shows *eventually P* (*at x within S*) ⟷ *eventually P* (*at x*) (**is** *?lhs* = *?rhs*)

proof−
from *assms* **obtain** *e* **where** *e*:*e*>*0* ∀ *y*. *dist x y* < *e* ⟶ *y* ∈ *S* **unfolding** *mem-interior ball-def subset-eq* **by** *auto*
{ assume *?lhs* **then obtain** *d* **where** *d*>*0* ∀ *xa*∈*S*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *P xa* **unfolding** *eventually-within* **by** *auto*
hence *?rhs* **unfolding** *eventually-at* **using** *e* **by** (*auto simp add*: *dist-sym intro*!: *add exI*[*of* - *min e d*])
} moreover
{ assume *?rhs* **hence** *?lhs* **unfolding** *eventually-within eventually-at* **by** *auto*
} ultimately
show *?thesis* **by** *auto*
qed

lemma *lim-within-interior*: *x* ∈ *interior S* ==> (*f* −−−> *l*) (*at x within S*) ⟷ (*f* −−−> *l*) (*at x*))
by (*simp add*: *tendsto-def eventually-within-interior*)

lemma *netlimit-within-interior*: **assumes** *x* ∈ *interior S*
shows *netlimit*(*at x within S*) = *x* (**is** *?lhs* = *?rhs*)
proof−
from *assms* **obtain** *e*::*real* **where** *e*:*e*>*0 ball x e* ⊆ *S* **using** *open-interior*[*of S*] **unfolding** *open-contains-ball* **using** *interior-subset*[*of S*] **by** *auto*
hence ¬ *trivial-limit* (*at x within S*) **using** *islimpt-subset*[*of x ball x e S*] **unfolding** *trivial-limit-within islimpt-ball centre-in-cball* **by** *auto*
thus *?thesis* **using** *netlimit-within* **by** *auto*
qed

64.20 Boundedness.

definition *bounded S* ⟷ (∃ *a*. ∀ (*x*::*real*^'*n*::*finite*) ∈ *S*. *norm x* <= *a*)

lemma *bounded-empty*[*simp*]: *bounded* {} **by** (*simp add*: *bounded-def*)
lemma *bounded-subset*: *bounded T* ⟹ *S* ⊆ *T* ==> *bounded S*
by (*metis bounded-def subset-eq*)

lemma *bounded-interior*[*intro*]: *bounded S* ==> *bounded*(*interior S*)
by (*metis bounded-subset interior-subset*)

lemma *bounded-closure*[*intro*]: **assumes** *bounded S* **shows** *bounded*(*closure S*)
proof−
from *assms* **obtain** *a* **where** *a*:∀ *x*∈*S*. *norm x* ≤ *a* **unfolding** *bounded-def* **by** *auto*
{ fix *x* **assume** *x*∈*closure S*
then obtain *xa* **where** *xa*:∀ *n*. *xa n* ∈ *S* (*xa* −−−> *x*) *sequentially* **unfolding** *closure-sequential* **by** *auto*
moreover have ∃ *y*. ∃ *x*. *netord sequentially x y* **using** *trivial-limit-sequentially* **unfolding** *trivial-limit-def* **by** *blast*
hence ∃ *y*. (∃ *x*. *netord sequentially x y*) ∧ (∀ *x*. *netord sequentially x y* ⟶ *norm* (*xa x*) ≤ *a*) **unfolding** *sequentially-def* **using** *a xa*(*1*) **by** *auto*

ultimately have *norm x ≤ a* **using** *Lim-norm-ubound*[*of sequentially xa x a*] *trivial-limit-sequentially* **unfolding** *eventually-def* **by** *auto*
}
thus *?thesis* **unfolding** *bounded-def* **by** *auto*
qed

lemma *bounded-cball*[*simp*,*intro*]: *bounded* (*cball x e*)
apply (*simp add*: *bounded-def*)
apply (*rule exI*[**where** *x*=*norm x* + *e*])
apply (*simp add*: *Ball-def*)
by *norm*

lemma *bounded-ball*[*simp*,*intro*]: *bounded*(*ball x e*)
by (*metis ball-subset-cball bounded-cball bounded-subset*)

lemma *finite-imp-bounded*[*intro*]: **assumes** *finite S* **shows** *bounded S*
proof−
{ fix *x F* **assume** *as*:*bounded F*
then obtain *a* **where** ∀ *x*∈*F*. *norm x* ≤ *a* **unfolding** *bounded-def* **by** *auto*
hence *bounded* (*insert x F*) **unfolding** *bounded-def* **by**(*auto intro*!: *add exI*[*of - max a* (*norm x*)])
}
thus *?thesis* **using** *finite-induct*[*of S bounded*] **using** *bounded-empty assms* **by** *auto*
qed

lemma *bounded-Un*[*simp*]: *bounded* (*S* ∪ *T*) ⟷ *bounded S* ∧ *bounded T*
apply (*auto simp add*: *bounded-def*)
by (*rule-tac x*=*max a aa* **in** *exI*, *auto*)

lemma *bounded-Union*[*intro*]: *finite F* ⟹ (∀ *S*∈*F*. *bounded S*) ⟹ *bounded*(⋃ *F*)
by (*induct rule*: *finite-induct*[*of F*], *auto*)

lemma *bounded-pos*: *bounded S* ⟷ (∃ *b*>*0*. ∀ *x*∈ *S*. *norm x* <= *b*)
apply (*simp add*: *bounded-def*)
apply (*subgoal-tac* ⋀*x* (*y*::*real*). *0* < *1* + *abs y* ∧ (*x* <= *y* ⟶ *x* <= *1* + *abs y*))
by *metis arith*

lemma *bounded-Int*[*intro*]: *bounded S* ∨ *bounded T* ⟹ *bounded* (*S* ∩ *T*)
by (*metis Int-lower1 Int-lower2 bounded-subset*)

lemma *bounded-diff*[*intro*]: *bounded S* ==> *bounded* (*S* − *T*)
apply (*metis Diff-subset bounded-subset*)
done

lemma *bounded-insert*[*intro*]:*bounded*(*insert x S*) ⟷ *bounded S*
by (*metis Diff-cancel Un-empty-right Un-insert-right bounded-Un bounded-subset finite.emptyI finite-imp-bounded infinite-remove subset-insertI*)

lemma *bot-bounded-UNIV*[*simp*, *intro*]: ~(*bounded* (*UNIV*:: (*real^'n*::*finite*) *set*))
proof(*auto simp add*: *bounded-pos not-le*)
fix *b*::*real* **assume** *b*: *b* >*0*
have *b1*: *b* +*1* ≥ *0* **using** *b* **by** *simp*
then obtain *x*:: *real^'n* **where** *norm x* = *b* + *1* **using** *vector-choose-size*[*of b+1*] **by** *blast*
hence *norm x* > *b* **using** *b* **by** *simp*
then show ∃(*x*::*real^'n*). *b* < *norm x* **by** *blast*
qed

lemma *bounded-linear-image*:
fixes *f* :: *real^'m*::*finite* ⇒ *real^'n*::*finite*
assumes *bounded S linear f*
shows *bounded*(*f* ' *S*)
proof−
from *assms*(*1*) **obtain** *b* **where** *b*:*b*>*0* ∀ *x*∈*S*. *norm x* ≤ *b* **unfolding** *bounded-pos* **by** *auto*
from *assms*(*2*) **obtain** *B* **where** *B*:*B*>*0* ∀ *x*. *norm* (*f x*) ≤ *B* ∗ *norm x* **using** *linear-bounded-pos* **by** *auto*
{ fix *x* **assume** *x*∈*S*
hence *norm x* ≤ *b* **using** *b* **by** *auto*
hence *norm* (*f x*) ≤ *B* ∗ *b* **using** *B*(*2*) **apply**(*erule-tac x*=*x* **in** *allE*)
by (*metis B*(*1*) *B*(*2*) *real-le-trans real-mult-le-cancel-iff2*)
}
thus *?thesis* **unfolding** *bounded-pos* **apply**(*rule-tac x*=*b*∗*B* **in** *exI*)
using *b B real-mult-order*[*of b B*] **by** (*auto simp add*: *real-mult-commute*)
qed

lemma *bounded-scaling*: *bounded S* ⟹ *bounded* ((λ*x*. *c* ∗*s x*) ' *S*)
apply (*rule bounded-linear-image*, *assumption*)
by (*rule linear-compose-cmul*, *rule linear-id*[*unfolded id-def*])

lemma *bounded-translation*: **assumes** *bounded S* **shows** *bounded* ((λ*x*. *a* + *x*) ' *S*)
proof−
from *assms* **obtain** *b* **where** *b*:*b*>*0* ∀ *x*∈*S*. *norm x* ≤ *b* **unfolding** *bounded-pos* **by** *auto*
{ fix *x* **assume** *x*∈*S*
hence *norm* (*a* + *x*) ≤ *b* + *norm a* **using** *norm-triangle-ineq*[*of a x*] *b* **by** *auto*
}
thus *?thesis* **unfolding** *bounded-pos* **using** *norm-ge-zero*[*of a*] *b*(*1*) **using** *add-strict-increasing*[*of b 0 norm a*]
by (*auto intro*!: *add exI*[*of* - *b* + *norm a*])
qed

Some theorems on sups and infs using the notion "bounded".

lemma *bounded-vec1*: *bounded*(*vec1* ' *S*) ⟷ (∃ *a*. ∀ *x*∈*S*. *abs x* <= *a*)

by (*simp add*: *bounded-def forall-vec1 norm-vec1 vec1-in-image-vec1*)

lemma *bounded-has-rsup*: **assumes** *bounded*(*vec1* ' *S*) *S* ≠ {}
shows ∀ *x*∈*S*. *x* <= *rsup S* **and** ∀ *b*. (∀ *x*∈*S*. *x* <= *b*) ⟶ *rsup S* <= *b*
proof
fix *x* **assume** *x*∈*S*
from *assms*(*1*) **obtain** *a* **where** *a*:∀ *x*∈*S*. |*x*| ≤ *a* **unfolding** *bounded-vec1* **by** *auto*
hence ∗:*S* ∗<= *a* **using** *setleI*[*of S a*] **by** (*metis abs-le-interval-iff mem-def*)
thus *x* ≤ *rsup S* **using** *rsup*[*OF* ‹*S*≠{}›] **using** *assms*(*1*)[*unfolded bounded-vec1*] **using** *isLubD2*[*of UNIV S rsup S x*] **using** ‹*x*∈*S*› **by** *auto*
next
show ∀ *b*. (∀ *x*∈*S*. *x* ≤ *b*) ⟶ *rsup S* ≤ *b* **using** *assms*
using *rsup*[*of S*, *unfolded isLub-def isUb-def leastP-def setle-def setge-def*]
apply (*auto simp add*: *bounded-vec1*)
by (*auto simp add*: *isLub-def isUb-def leastP-def setle-def setge-def*)
qed

lemma *rsup-insert*: **assumes** *bounded* (*vec1* ' *S*)
shows *rsup*(*insert x S*) = (*if S* = {} *then x else max x* (*rsup S*))
proof(*cases S*={})
case *True* **thus** *?thesis* **using** *rsup-finite-in*[*of* {*x*}] **by** *auto*
next
let *?S* = *insert x S*
case *False*
hence ∗:∀ *x*∈*S*. *x* ≤ *rsup S* **using** *bounded-has-rsup*(*1*)[*of S*] **using** *assms* **by** *auto*
hence *insert x S* ∗<= *max x* (*rsup S*) **unfolding** *setle-def* **by** *auto*
hence *isLub UNIV ?S* (*rsup ?S*) **using** *rsup*[*of ?S*] **by** *auto*
moreover
have ∗∗:*isUb UNIV ?S* (*max x* (*rsup S*)) **unfolding** *isUb-def setle-def* **using** ∗ **by** *auto*
{ fix *y* **assume** *as*:*isUb UNIV* (*insert x S*) *y*
hence *max x* (*rsup S*) ≤ *y* **unfolding** *isUb-def* **using** *rsup-le*[*OF* ‹*S*≠{}›]
unfolding *setle-def* **by** *auto* **}**
hence *max x* (*rsup S*) <=∗ *isUb UNIV* (*insert x S*) **unfolding** *setge-def Ball-def mem-def* **by** *auto*
hence *isLub UNIV ?S* (*max x* (*rsup S*)) **using** ∗∗ *isLubI2*[*of UNIV ?S max x* (*rsup S*)] **unfolding** *Collect-def* **by** *auto*
ultimately show *?thesis* **using** *real-isLub-unique*[*of UNIV ?S*] **using** ‹*S*≠{}› **by** *auto*
qed

lemma *sup-insert-finite*: *finite S* ⟹ *rsup*(*insert x S*) = (*if S* = {} *then x else max x* (*rsup S*))
apply (*rule rsup-insert*)
apply (*rule finite-imp-bounded*)
by *simp*

lemma *bounded-has-rinf*:
assumes *bounded*(*vec1* ' *S*) $S \neq \{\}$
shows $\forall x{\in}S.\ x >= rinf\ S$ **and** $\forall b.\ (\forall x{\in}S.\ x >= b) \longrightarrow rinf\ S >= b$
proof
fix *x* **assume** $x{\in}S$
from *assms*(*1*) **obtain** *a* **where** $a{:}\forall x{\in}S.\ |x| \leq a$ **unfolding** *bounded-vec1* **by** *auto*
hence $*{:}-\ a <=* S$ **using** *setgeI*[*of S* $-a$] **unfolding** *abs-le-interval-iff* **by** *auto*
thus $x \geq rinf\ S$ **using** *rinf*[*OF* ‹$S{\neq}\{\}$›] **using** *isGlbD2*[*of UNIV S rinf S x*] **using** ‹$x{\in}S$› **by** *auto*
next
show $\forall b.\ (\forall x{\in}S.\ x >= b) \longrightarrow rinf\ S \geq b$ **using** *assms*
using *rinf*[*of S, unfolded isGlb-def isLb-def greatestP-def setle-def setge-def*]
apply (*auto simp add*: *bounded-vec1*)
by (*auto simp add*: *isGlb-def isLb-def greatestP-def setle-def setge-def*)
qed

lemma *real-isGlb-unique*: [| *isGlb R S x*; *isGlb R S y* |] ==> $x = (y{::}real)$
apply (*frule isGlb-isLb*)
apply (*frule-tac x* = *y* **in** *isGlb-isLb*)
apply (*blast intro*!: *order-antisym dest*!: *isGlb-le-isLb*)
done

lemma *rinf-insert*: **assumes** *bounded* (*vec1* ' *S*)
shows *rinf*(*insert x S*) = (*if S* = {} *then x else min x* (*rinf S*)) (**is** *?lhs* = *?rhs*)
proof(*cases S*={})
case *True* **thus** *?thesis* **using** *rinf-finite-in*[*of* {*x*}] **by** *auto*
next
let *?S* = *insert x S*
case *False*
hence $*{:}\forall x{\in}S.\ x \geq rinf\ S$ **using** *bounded-has-rinf*(*1*)[*of S*] **using** *assms* **by** *auto*
hence *min x* (*rinf S*) $<=*$ *insert x S* **unfolding** *setge-def* **by** *auto*
hence *isGlb UNIV ?S* (*rinf ?S*) **using** *rinf*[*of ?S*] **by** *auto*
moreover
have $**{:}$*isLb UNIV ?S* (*min x* (*rinf S*)) **unfolding** *isLb-def setge-def* **using** $*$ **by** *auto*
{ fix *y* **assume** *as*:*isLb UNIV* (*insert x S*) *y*
hence *min x* (*rinf S*) $\geq y$ **unfolding** *isLb-def* **using** *rinf-ge*[*OF* ‹$S{\neq}\{\}$›]
unfolding *setge-def* **by** *auto* **}**
hence *isLb UNIV* (*insert x S*) $*<=$ *min x* (*rinf S*) **unfolding** *setle-def Ball-def mem-def* **by** *auto*
hence *isGlb UNIV ?S* (*min x* (*rinf S*)) **using** $**$ *isGlbI2*[*of UNIV ?S min x* (*rinf S*)] **unfolding** *Collect-def* **by** *auto*
ultimately show *?thesis* **using** *real-isGlb-unique*[*of UNIV ?S*] **using** ‹$S{\neq}\{\}$› **by** *auto*
qed

lemma *inf-insert-finite*: *finite S ==> rinf*(*insert x S*) = (*if S* = {} *then x else min x* (*rinf S*))
by (*rule rinf-insert*, *rule finite-imp-bounded*, *simp*)

64.21 Compactness (the definition is the one based on convegent subsequences).

definition *compact S* ⟷
(∀ (*f*::*nat* ⇒ *real*^'*n*::*finite*). (∀ *n*. *f n* ∈ *S*) ⟶
(∃ *l*∈*S*. ∃ *r*. (∀ *m n*. *m* < *n* ⟶ *r m* < *r n*) ∧ ((*f o r*) −−−> *l*) *sequentially*))

lemma *monotone-bigger*: **fixes** *r*::*nat*⇒*nat*
assumes ∀ *m n*::*nat*. *m* < *n* −−> *r m* < *r n*
shows *n* ≤ *r n*
proof(*induct n*)
show *0* ≤ *r 0* **by** *auto*
next
fix *n* **assume** *n* ≤ *r n*
moreover have *r n* < *r* (*Suc n*) **using** *assms* **by** *auto*
ultimately show *Suc n* ≤ *r* (*Suc n*) **by** *auto*
qed

lemma *lim-subsequence*: ∀ *m n*. *m* < *n* ⟶ *r m* < *r n* ⟹ (*s* −−−> *l*) *sequentially*
⟹ ((*s o r*) −−−> *l*) *sequentially*
unfolding *Lim-sequentially* **by** (*simp*, *metis monotone-bigger le-trans*)

lemma *num-Axiom*: *EX*! *g*. *g 0* = *e* ∧ (∀ *n*. *g* (*Suc n*) = *f n* (*g n*))
unfolding *Ex1-def*
apply (*rule-tac x*=*nat-rec e f* **in** *exI*)
apply (*rule conjI*)+
apply (*rule def-nat-rec-0*, *simp*)
apply (*rule allI*, *rule def-nat-rec-Suc*, *simp*)
apply (*rule allI*, *rule impI*, *rule ext*)
apply (*erule conjE*)
apply (*induct-tac x*)
apply (*simp add*: *nat-rec-0*)
apply (*erule-tac x*=*n* **in** *allE*)
apply (*simp*)
done

lemma *convergent-bounded-increasing*: **fixes** *s* ::*nat*⇒*real*
assumes ∀ *m n*. *m* ≤ *n* −−> *s m* ≤ *s n* **and** ∀ *n*. *abs*(*s n*) ≤ *b*
shows ∃ *l*. ∀ *e*::*real*>*0*. ∃ *N*. ∀ *n* ≥ *N*. *abs*(*s n* − *l*) < *e*
proof−
have *isUb UNIV* (*range s*) *b* **using** *assms*(*2*) **and** *abs-le-D1* **unfolding** *isUb-def*
and *setle-def* **by** *auto*
then obtain *t* **where** *t*:*isLub UNIV* (*range s*) *t* **using** *reals-complete*[*of range s*] **by** *auto*

{ fix *e*::*real* **assume** *e*>*0* **and** *as*:∀ *N*. ∃ *n*≥*N*. ¬ |*s n* − *t*| < *e*
{ fix *n*::*nat*
obtain *N* **where** *N*≥*n* **and** *n*:|*s N* − *t*| ≥ *e* **using** *as*[*THEN spec*[**where** *x*=*n*]] **by** *auto*
have *t* ≥ *s N* **using** *isLub-isUb*[*OF t*, *unfolded isUb-def setle-def*] **by** *auto*
with *n* **have** *s N* ≤ *t* − *e* **using** ‹*e*>*0*› **by** *auto*
hence *s n* ≤ *t* − *e* **using** *assms*(*1*)[*THEN spec*[**where** *x*=*n*], *THEN spec*[**where** *x*=*N*]] **using** ‹*n*≤*N*› **by** *auto* **}**
hence *isUb UNIV* (*range s*) (*t* − *e*) **unfolding** *isUb-def* **and** *setle-def* **by** *auto*
hence *False* **using** *isLub-le-isUb*[*OF t*, *of t* − *e*] **and** ‹*e*>*0*› **by** *auto* **}**
thus *?thesis* **by** *blast*
qed

lemma *convergent-bounded-monotone*: **fixes** *s*::*nat* ⇒ *real*
assumes ∀ *n*. *abs*(*s n*) ≤ *b* **and** (∀ *m n*. *m* ≤ *n* −−> *s m* ≤ *s n*) ∨ (∀ *m n*. *m* ≤ *n* −−> *s n* ≤ *s m*)
shows ∃ *l*. ∀ *e*::*real*>*0*. ∃ *N*. ∀ *n*≥*N*. *abs*(*s n* − *l*) < *e*
using *convergent-bounded-increasing*[*of s b*] *assms* **using** *convergent-bounded-increasing*[*of* λ*n*. − *s n b*]
apply *auto* **unfolding** *minus-add-distrib*[*THEN sym*, *unfolded diff-minus*[*THEN sym*]]
unfolding *abs-minus-cancel* **by**(*rule-tac x*=−*l* **in** *exI*)*auto*

lemma *compact-real-lemma*:
assumes ∀ *n*::*nat*. *abs*(*s n*) ≤ *b*
shows ∃ *l r*. (∀ *m n*::*nat*. *m* < *n* −−> *r m* < *r n*) ∧
(∀ *e*>*0*::*real*. ∃ *N*. ∀ *n*≥*N*. (*abs*(*s* (*r n*) − *l*) < *e*))
proof−
obtain *r* **where** *r*:∀ *m n*::*nat*. *m* < *n* ⟶ *r m* < *r n*
(∀ *m n*. *m* ≤ *n* ⟶ *s* (*r m*) ≤ *s* (*r n*)) ∨ (∀ *m n*. *m* ≤ *n* ⟶ *s* (*r n*) ≤ *s* (*r m*))
using *seq-monosub*[*of s*] **by** (*auto simp add*: *subseq-def monoseq-def*)
thus *?thesis* **using** *convergent-bounded-monotone*[*of s o r b*] **and** *assms* **by** *auto*
qed

lemma *compact-lemma*:
assumes *bounded s* **and** ∀ *n*. (*x*::*nat* ⇒*real*^'*a*::*finite*) *n* ∈ *s*
shows ∀ *d*.
∃ *l*::(*real*^'*a*::*finite*). ∃ *r*. (∀ *n m*::*nat*. *m* < *n* −−> *r m* < *r n*) ∧
(∀ *e*>*0*. ∃ *N*. ∀ *n*≥*N*. ∀ *i*∈*d*. |*x* (*r n*) $ *i* − *l* $ *i*| < *e*)
proof−
obtain *b* **where** *b*:∀ *x*∈*s*. *norm x* ≤ *b* **using** *assms*(*1*)[*unfolded bounded-def*] **by** *auto*
{ { fix *i*::'*a*
{ fix *n*::*nat*
have |*x n* $ *i*| ≤ *b* **using** *b*[*THEN bspec*[**where** *x*=*x n*]] **and** *component-le-norm*[*of x n i*] **and** *assms*(*2*)[*THEN spec*[**where** *x*=*n*]] **by** *auto* **}**
hence ∀ *n*. |*x n* $ *i*| ≤ *b* **by** *auto*

} note $b' = this$

fix $d::'a\ set$ **have** *finite d* **by** *simp*
hence $\exists\, l::(real\hat{}'a).\ \exists\ r.\ (\forall\, n\ m::nat.\ m < n \longrightarrow r\ m < r\ n) \wedge$
$(\forall\, e{>}0.\ \exists\, N.\ \forall\, n{\geq}N.\ \forall\, i{\in}d.\ |x\ (r\ n)\ \$\ i - l\ \$\ i| < e)$
proof(*induct d*) **case** *empty* **thus** *?case* **by** *auto*
next case (*insert k d*)
obtain $l1::real\hat{}'a$ **and** *r1* **where** $r1{:}\forall\, n\ m::nat.\ m < n \longrightarrow r1\ m < r1\ n$ **and** $lr1{:}\forall\, e{>}0.\ \exists\, N.\ \forall\, n{\geq}N.\ \forall\, i{\in}d.\ |x\ (r1\ n)\ \$\ i - l1\ \$\ i| < e$
using *insert*(*3*) **by** *auto*
obtain *l2 r2* **where** $r2{:}\forall\, m\ n::nat.\ m < n \longrightarrow r2\ m < r2\ n$ **and** $lr2{:}\forall\, e{>}0.\ \exists\, N.\ \forall\, n{\geq}N.\ |(x \circ r1)\ (r2\ n)\ \$\ k - l2| < e$
using $b'[of\ k]$ **and** *compact-real-lemma*[*of* $\lambda i.\ ((x \circ r1)\ i)\$k\ b$] **by** *auto*
def $r \equiv r1 \circ r2$ **have** $r{:}\forall\, m\ n.\ m < n \longrightarrow r\ m < r\ n$ **unfolding** *r-def o-def* **using** *r1* **and** *r2* **by** *auto*
moreover
def $l \equiv (\chi\ i.\ if\ i = k\ then\ l2\ else\ l1\$i)::real\hat{}'a$
{ fix $e::real$ **assume** $e{>}0$
from *lr1* **obtain** *N1* **where** $N1{:}\forall\, n{\geq}N1.\ \forall\, i{\in}d.\ |x\ (r1\ n)\ \$\ i - l1\ \$\ i| < e$ **using** ‹$e{>}0$› **by** *blast*
from *lr2* **obtain** *N2* **where** $N2{:}\forall\, n{\geq}N2.\ |(x \circ r1)\ (r2\ n)\ \$\ k - l2| < e$ **using** ‹$e{>}0$› **by** *blast*
{ fix *n* **assume** $n{:}n{\geq}\ N1 + N2$
fix *i* **assume** $i{:}i{\in}(insert\ k\ d)$
hence $|x\ (r\ n)\ \$\ i - l\ \$\ i| < e$
using *N2*[*THEN spec*[**where** $x{=}n$]] **and** *n*
using *N1*[*THEN spec*[**where** $x{=}r2\ n$]] **and** *n*
using *monotone-bigger*[*OF r*] **and** *i*
unfolding *l-def* **and** *r-def*
using *monotone-bigger*[*OF r2*, *of n*] **by** *auto* **}**
hence $\exists\, N.\ \forall\, n{\geq}N.\ \forall\, i{\in}(insert\ k\ d).\ |x\ (r\ n)\ \$\ i - l\ \$\ i| < e$ **by** *blast*
}
ultimately show *?case* **by** *auto*
qed } thus *?thesis* **by** *auto*
qed

lemma *bounded-closed-imp-compact*: **fixes** $s::(real\hat{}'a::finite)\ set$
assumes *bounded s* **and** *closed s*
shows *compact s*
proof−
let *?d* = $UNIV::'a\ set$
{ fix *f* **assume** $as{:}\forall\, n::nat.\ f\ n \in s$
obtain $l::real\hat{}'a$ **and** *r* **where** $r{:}\forall\, n\ m::nat.\ m < n \longrightarrow r\ m < r\ n$
and $lr{:}\forall\, e{>}0.\ \exists\, N.\ \forall\, n{\geq}N.\ \forall\, i{\in}?d.\ |f\ (r\ n)\ \$\ i - l\ \$\ i| < e$
using *compact-lemma*[*OF assms*(*1*) *as*, *THEN spec*[**where** $x{=}?d$]] **by** *auto*
{ fix $e::real$ **assume** $e{>}0$
hence $0 < e\ /\ (real\text{-}of\text{-}nat\ (card\ ?d))$ **using** *zero-less-card-finite* **using** *divide-pos-pos*[*of e*, *of real-of-nat* (*card ?d*)] **by** *auto*

then obtain N::*nat* **where** N:$\forall\, n{\geq}N.\ \forall\, i{\in}?d.\ |f\ (r\ n)\ \$\ i - l\ \$\ i| < e\ /$ (*real-of-nat* (*card* *?d*)) **using** *lr*[*THEN spec*[**where** $x{=}e\ /$ (*real-of-nat* (*card* *?d*))]] **by** *blast*
{ fix n **assume** n:$n{\geq}N$
hence *finite* *?d* *?d* $\neq$ {} **by** *auto*
moreover
{ fix i **assume** i:$i \in$ *?d*
hence $|((f \circ r)\ n - l)\ \$\ i| < e\ /$ *real-of-nat* (*card* *?d*) **using** ‹$n{\geq}N$› **using** N[*THEN spec*[**where** $x{=}n$]]
by *auto* **}**
ultimately have $(\sum i \in ?d.\ |((f \circ r)\ n - l)\ \$\ i|)$
$< (\sum i \in ?d.\ e\ /$ *real-of-nat* (*card* *?d*))
using *setsum-strict-mono*[*of* *?d* $\lambda i.\ |((f \circ r)\ n - l)\ \$\ i|$ $\lambda i.\ e\ /$ (*real-of-nat* (*card* *?d*))] **by** *auto*
hence $(\sum i \in ?d.\ |((f \circ r)\ n - l)\ \$\ i|) < e$ **unfolding** *setsum-constant* **by** *auto*
hence *dist* $((f \circ r)\ n)\ l < e$ **unfolding** *dist-def* **using** *norm-le-l1*[*of* $(f \circ r)\ n - l$] **by** *auto* **}**
hence $\exists\, N.\ \forall\, n{\geq}N.$ *dist* $((f \circ r)\ n)\ l < e$ **by** *auto* **}**
hence $*$:$((f \circ r)\ ---{>}\ l)$ *sequentially* **unfolding** *Lim-sequentially* **by** *auto*
moreover have $l{\in}s$
using *assms*(*2*)[*unfolded closed-sequential-limits*, *THEN spec*[**where** $x{=}f \circ r$], *THEN spec*[**where** $x{=}l$]] **and** $*$ **and** *as* **by** *auto*
ultimately have $\exists\, l{\in}s.\ \exists\, r.\ (\forall\, m\ n.\ m < n \longrightarrow r\ m < r\ n) \wedge ((f \circ r)\ ---{>}\ l)$ *sequentially* **using** r **by** *auto* **}**
thus *?thesis* **unfolding** *compact-def* **by** *auto*
qed

64.22 Completeness.

definition *cauchy-def*:*cauchy* $s \longleftrightarrow (\forall\, e{>}0.\ \exists\, N.\ \forall\, m\ n.\ m \geq N \wedge n \geq N\ {-}{-}{>}$ $dist(s\ m)(s\ n) < e)$

definition *complete-def*:*complete* $s \longleftrightarrow (\forall\, f{::}(nat{=>}real\hat{}'a{::}finite).\ (\forall\, n.\ f\ n \in s) \wedge$ *cauchy* f
$${-}{-}{>}\ (\exists\, l \in s.\ (f\ ---{>}\ l)\ \textit{sequentially}))$$

lemma *cauchy*: *cauchy* $s \longleftrightarrow (\forall\, e{>}0.\exists\ N{::}nat.\ \forall\, n{\geq}N.\ dist(s\ n)(s\ N) < e)$ (**is** *?lhs* = *?rhs*)
proof−
{ assume *?rhs*
{ fix e::*real*
assume $e{>}0$
with ‹*?rhs*› **obtain** N **where** N:$\forall\, n{\geq}N.$ *dist* $(s\ n)\ (s\ N) < e/2$
by (*erule-tac* $x{=}e/2$ **in** *allE*) *auto*
{ fix $n\ m$
assume nm:$N \leq m \wedge N \leq n$
hence *dist* $(s\ m)\ (s\ n) < e$ **using** N
using *dist-triangle-half-l*[*of* $s\ m\ s\ N\ e\ s\ n$]

```
        by blast
    }
    hence ∃N. ∀m n. N ≤ m ∧ N ≤ n ⟶ dist (s m) (s n) < e
      by blast
  }
  hence ?lhs
    unfolding cauchy-def
    by blast
 }
 thus ?thesis
   unfolding cauchy-def
   using dist-triangle-half-l
   by blast
qed

lemma convergent-imp-cauchy:
 (s ---> l) sequentially ==> cauchy s
proof(simp only: cauchy-def, rule, rule)
 fix e::real assume e>0 (s ---> l) sequentially
 then obtain N::nat where N:∀n≥N. dist (s n) l < e/2 unfolding Lim-sequentially
by(erule-tac x=e/2 in allE) auto
 thus ∃N. ∀m n. N ≤ m ∧ N ≤ n ⟶ dist (s m) (s n) < e using dist-triangle-half-l[of
- l e -] by (rule-tac x=N in exI) auto
qed

lemma cauchy-imp-bounded: assumes cauchy s shows bounded {y. (∃n::nat. y
= s n)}
proof-
 from assms obtain N::nat where ∀m n. N ≤ m ∧ N ≤ n ⟶ dist (s m) (s
n) < 1 unfolding cauchy-def apply(erule-tac x= 1 in allE) by auto
 hence N:∀n. N ≤ n ⟶ dist (s N) (s n) < 1 by auto
 { fix n::nat assume n≥N
   hence norm (s n) ≤ norm (s N) + 1 using N apply(erule-tac x=n in allE)
unfolding dist-def
    using norm-triangle-sub[of s N s n] by (auto, metis dist-def dist-sym le-add-right-mono
norm-triangle-sub real-less-def)
 }
 hence ∀n≥N. norm (s n) ≤ norm (s N) + 1 by auto
 moreover
 have bounded (s ' {0..N}) using finite-imp-bounded[of s ' {1..N}] by auto
 then obtain a where a:∀x∈s ' {0..N}. norm x ≤ a unfolding bounded-def
by auto
 ultimately show ?thesis unfolding bounded-def
   apply(rule-tac x=max a (norm (s N) + 1) in exI) apply auto
   apply(erule-tac x=n in allE) apply(erule-tac x=n in ballE) by auto
qed

lemma compact-imp-complete: assumes compact s shows complete s
proof-
```

{ fix *f* **assume** *as*: (∀ *n*::*nat*. *f n* ∈ *s*) *cauchy f*
from *as*(*1*) **obtain** *l r* **where** *lr*: *l*∈*s* (∀ *m n*. *m* < *n* ⟶ *r m* < *r n*) ((*f* ∘ *r*) −−−> *l*) *sequentially* **using** *assms* **unfolding** *compact-def* **by** *blast*

{ fix *n* :: *nat* **have** *lr'*:*n* ≤ *r n*
proof (*induct n*)
show *0* ≤ *r 0* **using** *lr*(*2*) **by** *blast*
next fix *na* **assume** *na* ≤ *r na* **moreover have** *na* < *Suc na* ⟶ *r na* < *r* (*Suc na*) **using** *lr*(*2*) **by** *blast*
ultimately show *Suc na* ≤ *r* (*Suc na*) **by** *auto*
qed } note *lr'* = *this*

{ fix *e*::*real* **assume** *e*>*0*
from *as*(*2*) **obtain** *N* **where** *N*:∀ *m n*. *N* ≤ *m* ∧ *N* ≤ *n* ⟶ *dist* (*f m*) (*f n*) < *e*/*2* **unfolding** *cauchy-def* **using** ‹*e*>*0*› **apply** (*erule-tac x*=*e*/*2* **in** *allE*) **by** *auto*
from *lr*(*3*)[*unfolded Lim-sequentially*, *THEN spec*[**where** *x*=*e*/*2*]] **obtain** *M* **where** *M*:∀ *n*≥*M*. *dist* ((*f* ∘ *r*) *n*) *l* < *e*/*2* **using** ‹*e*>*0*› **by** *auto*
{ fix *n*::*nat* **assume** *n*:*n* ≥ *max N M*
have *dist* ((*f* ∘ *r*) *n*) *l* < *e*/*2* **using** *n M* **by** *auto*
moreover have *r n* ≥ *N* **using** *lr'*[*of n*] *n* **by** *auto*
hence *dist* (*f n*) ((*f* ∘ *r*) *n*) < *e* / *2* **using** *N* **using** *n* **by** *auto*
ultimately have *dist* (*f n*) *l* < *e* **using** *dist-triangle-half-r*[*of f* (*r n*) *f n e l*] **by** (*auto simp add*: *dist-sym*) **}**
hence ∃ *N*. ∀ *n*≥*N*. *dist* (*f n*) *l* < *e* **by** *blast* **}**
hence ∃ *l*∈*s*. (*f* −−−> *l*) *sequentially* **using** ‹*l*∈*s*› **unfolding** *Lim-sequentially* **by** *auto* **}**
thus *?thesis* **unfolding** *complete-def* **by** *auto*
qed

lemma *complete-univ*:
complete UNIV
proof(*simp add*: *complete-def*, *rule*, *rule*)
fix *f*::*nat* ⇒ *real*^'*n*::*finite* **assume** *cauchy f*
hence *bounded* (*f*‘*UNIV*) **using** *cauchy-imp-bounded*[*of f*] **unfolding** *image-def* **by** *auto*
hence *compact* (*closure* (*f*‘*UNIV*)) **using** *bounded-closed-imp-compact*[*of closure* (*range f*)] **by** *auto*
hence *complete* (*closure* (*range f*)) **using** *compact-imp-complete* **by** *auto*
thus ∃ *l*. (*f* −−−> *l*) *sequentially* **unfolding** *complete-def*[*of closure* (*range f*)] **using** ‹*cauchy f*› **unfolding** *closure-def* **by** *auto*
qed

lemma *complete-eq-closed*: *complete s* ⟷ *closed s* (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ fix *x* **assume** *x islimpt s*
then obtain *f* **where** *f*:∀ *n*. *f n* ∈ *s* − {*x*} (*f* −−−> *x*) *sequentially* **unfolding** *islimpt-sequential* **by** *auto*

then obtain l **where** l: $l{\in}s$ $(f \longrightarrow l)$ *sequentially* **using** ‹*?lhs*›[*unfolded complete-def*] **using** *convergent-imp-cauchy*[*of f x*] **by** *auto*
hence $x \in s$ **using** *Lim-unique*[*of sequentially f l x*] *trivial-limit-sequentially f*(*2*) **by** *auto* **}**
thus *?rhs* **unfolding** *closed-limpt* **by** *auto*
next
assume *?rhs*
{ fix f **assume** *as*:$\forall\, n{::}nat.\ f\ n \in s$ *cauchy f*
then obtain l **where** $(f \longrightarrow l)$ *sequentially* **using** *complete-univ*[*unfolded complete-def*, *THEN spec*[**where** $x{=}f$]] **by** *auto*
hence $\exists\, l{\in}s.\ (f \longrightarrow l)$ *sequentially* **using** ‹*?rhs*›[*unfolded closed-sequential-limits*, *THEN spec*[**where** $x{=}f$], *THEN spec*[**where** $x{=}l$]] **using** *as*(*1*) **by** *auto* **}**
thus *?lhs* **unfolding** *complete-def* **by** *auto*
qed

lemma *convergent-eq-cauchy*: $(\exists\, l.\ (s \longrightarrow l)$ *sequentially*$) \longleftrightarrow$ *cauchy s* (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **then obtain** l **where** $(s \longrightarrow l)$ *sequentially* **by** *auto*
thus *?rhs* **using** *convergent-imp-cauchy* **by** *auto*
next
assume *?rhs* **thus** *?lhs* **using** *complete-univ*[*unfolded complete-def*, *THEN spec*[**where** $x{=}s$]] **by** *auto*
qed

lemma *convergent-imp-bounded*: $(s \longrightarrow l)$ *sequentially* $\Longrightarrow$ *bounded* $(s$ ‘ $(UNIV{::}(nat\ set)))$
using *convergent-eq-cauchy*[*of s*]
using *cauchy-imp-bounded*[*of s*]
unfolding *image-def*
by *auto*

64.23 Total boundedness.

fun *helper-1*::$((real\hat{}'n{::}finite)\ set) \Rightarrow real \Rightarrow nat \Rightarrow real\hat{}'n$ **where**
helper-1 s e n = $(SOME\ y{::}real\hat{}'n.\ y \in s \wedge (\forall\, m{<}n.\ \neg\ (dist\ (helper\text{-}1\ s\ e\ m)\ y < e)))$
declare *helper-1.simps*[*simp del*]

lemma *compact-imp-totally-bounded*:
assumes *compact s*
shows $\forall\, e{>}0.\ \exists\, k.$ *finite* $k \wedge k \subseteq s \wedge s \subseteq (\bigcup((\lambda x.\ ball\ x\ e)$ ‘ $k))$
proof(*rule*, *rule*, *rule ccontr*)
fix *e*::*real* **assume** *e*>*0* **and** *assm*:$\neg\ (\exists\, k.$ *finite* $k \wedge k \subseteq s \wedge s \subseteq \bigcup(\lambda x.\ ball\ x\ e)$ ‘ $k)$
def $x \equiv$ *helper-1 s e*
{ fix n
have $x\ n \in s \wedge (\forall\, m{<}n.\ \neg\ dist\ (x\ m)\ (x\ n) < e)$
proof(*induct-tac rule*:*nat-less-induct*)

fix n **def** $Q \equiv (\lambda y.\ y \in s \wedge (\forall m{<}n.\ \neg\ dist\ (x\ m)\ y < e))$
assume *as*:$\forall\, m{<}n.\ x\ m \in s \wedge (\forall\, ma{<}m.\ \neg\ dist\ (x\ ma)\ (x\ m) < e)$
have $\neg\ s \subseteq (\bigcup x{\in}x\ `\ \{0..{<}n\}.\ ball\ x\ e)$ **using** *assm* **apply** *simp* **apply**(*erule-tac x*=$x\ `\ \{0\ ..{<}\ n\}$ **in** *allE*) **using** *as* **by** *auto*
then obtain z **where** $z{:}z{\in}s\ z \notin (\bigcup x{\in}x\ `\ \{0..{<}n\}.\ ball\ x\ e)$ **unfolding** *subset-eq* **by** *auto*
have $Q\ (x\ n)$ **unfolding** *x-def* **and** *helper-1.simps*[*of s e n*]
apply(*rule someI2*[**where** $a{=}z$]) **unfolding** *x-def*[*symmetric*] **and** *Q-def* **using** z **by** *auto*
thus $x\ n \in s \wedge (\forall\, m{<}n.\ \neg\ dist\ (x\ m)\ (x\ n) < e)$ **unfolding** *Q-def* **by** *auto*
qed }
hence $\forall\, n{::}nat.\ x\ n \in s$ **and** $x{:}\forall\, n.\ \forall\, m < n.\ \neg\ (dist\ (x\ m)\ (x\ n) < e)$ **by** *blast*+
then obtain $l\ r$ **where** $l{\in}s$ **and** $r{:}\forall\, m\ n.\ m < n \longrightarrow r\ m < r\ n$ **and** $((x \circ r)\ ---{>}\ l)\ sequentially$ **using** *assms*(*1*)[*unfolded compact-def*, *THEN spec*[**where** $x{=}x$]] **by** *auto*
from *this*(*3*) **have** *cauchy* $(x \circ r)$ **using** *convergent-imp-cauchy* **by** *auto*
then obtain $N{::}nat$ **where** $N{:}\forall\, m\ n.\ N \leq m \wedge N \leq n \longrightarrow dist\ ((x \circ r)\ m)\ ((x \circ r)\ n) < e$ **unfolding** *cauchy-def* **using** ‹*e*>*0*› **by** *auto*
show *False*
using *N*[*THEN spec*[**where** $x{=}N$], *THEN spec*[**where** $x{=}N{+}1$]]
using *r*[*THEN spec*[**where** $x{=}N$], *THEN spec*[**where** $x{=}N{+}1$]]
using *x*[*THEN spec*[**where** $x{=}r\ (N{+}1)$], *THEN spec*[**where** $x{=}r\ (N)$]] **by** *auto*
qed

64.24 Heine-Borel theorem (following Burkill & Burkill vol. 2)

lemma *heine-borel-lemma*: **fixes** *s*::(*real*^'*n*::*finite*) *set*
assumes *compact s* $s \subseteq (\bigcup\ t)$ $\forall\, b \in t.\ open\ b$
shows $\exists\, e{>}0.\ \forall\, x \in s.\ \exists\, b \in t.\ ball\ x\ e \subseteq b$
proof(*rule ccontr*)
assume $\neg\ (\exists\, e{>}0.\ \forall\, x{\in}s.\ \exists\, b{\in}t.\ ball\ x\ e \subseteq b)$
hence *cont*:$\forall\, e{>}0.\ \exists\, x{\in}s.\ \forall\, xa{\in}t.\ \neg\ (ball\ x\ e \subseteq xa)$ **by** *auto*
{ **fix** *n*::*nat*
have *1 / real* $(n + 1) > 0$ **by** *auto*
hence $\exists\, x.\ x{\in}s \wedge (\forall\, xa{\in}t.\ \neg\ (ball\ x\ (inverse\ (real\ (n{+}1))) \subseteq xa))$ **using** *cont* **unfolding** *Bex-def* **by** *auto* }
hence $\forall\, n{::}nat.\ \exists\, x.\ x \in s \wedge (\forall\, xa{\in}t.\ \neg\ ball\ x\ (inverse\ (real\ (n + 1))) \subseteq xa)$ **by** *auto*
then obtain f **where** $f{:}\forall\, n{::}nat.\ f\ n \in s \wedge (\forall\, xa{\in}t.\ \neg\ ball\ (f\ n)\ (inverse\ (real\ (n + 1))) \subseteq xa)$
using *choice*[*of* $\lambda n{::}nat.\ \lambda x.\ x{\in}s \wedge (\forall\, xa{\in}t.\ \neg\ ball\ x\ (inverse\ (real\ (n + 1))) \subseteq xa)$] **by** *auto*

then obtain $l\ r$ **where** $l{:}l{\in}s$ **and** $r{:}\forall\, m\ n.\ m < n \longrightarrow r\ m < r\ n$ **and** $lr{:}((f \circ r)\ ---{>}\ l)\ sequentially$
using *assms*(*1*)[*unfolded compact-def*, *THEN spec*[**where** $x{=}f$]] **by** *auto*

obtain b **where** $l{\in}b$ $b{\in}t$ **using** *assms*(*2*) **and** l **by** *auto*
then obtain e **where** $e{>}0$ **and** $e{:}\forall z.\ dist\ z\ l < e \longrightarrow z{\in}b$
using *assms*(*3*)[*THEN bspec*[**where** $x{=}b$]] **unfolding** *open-def* **by** *auto*

then obtain *N1* **where** $N1{:}\forall n{\geq}N1.\ dist\ ((f \circ r)\ n)\ l < e\ /\ 2$
using *lr*[*unfolded Lim-sequentially*, *THEN spec*[**where** $x{=}e/2$]] **by** *auto*

obtain *N2*::*nat* **where** *N2*:$N2{>}0$ *inverse* (*real N2*) $<$ e */2* **using** *real-arch-inv*[*of* $e/2$] **and** ‹$e{>}0$› **by** *auto*
have *N2'*:*inverse* (*real* (*r* (*N1* + *N2*) +*1*)) $<$ $e/2$
apply(*rule order-less-trans*) **apply**(*rule less-imp-inverse-less*) **using** *N2*
using *monotone-bigger*[*OF r*, *of N1* + *N2*] **by** *auto*

def $x \equiv (f\ (r\ (N1 + N2)))$
have $x{:}\neg$ *ball x* (*inverse* (*real* (*r* (*N1* + *N2*) + *1*))) $\subseteq$ b **unfolding** *x-def*
using f[*THEN spec*[**where** $x{=}r$ (*N1* + *N2*)]] **using** ‹$b{\in}t$› **by** *auto*
have $\exists\, y{\in}ball\ x$ (*inverse* (*real* (*r* (*N1* + *N2*) + *1*))). $y{\notin}b$ **apply**(*rule ccontr*) **using** x **by** *auto*
then obtain y **where** $y{:}y \in ball\ x$ (*inverse* (*real* (*r* (*N1* + *N2*) + *1*))) $y \notin b$ **by** *auto*

have *dist x l* $<$ *e/2* **using** *N1* **unfolding** *x-def o-def* **by** *auto*
hence *dist y l* $<$ *e* **using** *y N2'* **using** *dist-triangle*[*of y l x*]**by** (*auto simp add*:*dist-sym*)

thus *False* **using** *e* **and** ‹$y{\notin}b$› **by** *auto*
qed

lemma *compact-imp-heine-borel*: *compact s* ==> $(\forall f.\ (\forall\, t \in f.\ open\ t) \wedge s \subseteq (\bigcup f)$
$\longrightarrow (\exists f'.\ f' \subseteq f \wedge finite\ f' \wedge s \subseteq (\bigcup\ f')))$
proof *clarify*
fix f **assume** *compact s* $\forall\, t{\in}f.\ open\ t\ s \subseteq \bigcup f$
then obtain *e*::*real* **where** $e{>}0$ **and** $\forall\, x{\in}s.\ \exists\, b{\in}f.\ ball\ x\ e \subseteq b$ **using** *heine-borel-lemma*[*of s f*] **by** *auto*
hence $\forall\, x{\in}s.\ \exists\, b.\ b{\in}f \wedge ball\ x\ e \subseteq b$ **by** *auto*
hence $\exists\, bb.\ \forall\, x{\in}s.\ bb\ x \in f \wedge ball\ x\ e \subseteq bb\ x$ **using** *bchoice*[*of s* $\lambda x\ b.\ b{\in}f \wedge ball\ x\ e \subseteq b$] **by** *auto*
then obtain *bb* **where** $bb{:}\forall\, x{\in}s.\ (bb\ x) \in f \wedge ball\ x\ e \subseteq (bb\ x)$ **by** *blast*

from ‹*compact s*› **have** $\exists\ k.\ finite\ k \wedge k \subseteq s \wedge s \subseteq \bigcup(\lambda x.\ ball\ x\ e)\ `\ k$ **using** *compact-imp-totally-bounded*[*of s*] ‹$e{>}0$› **by** *auto*
then obtain k **where** $k{:}finite\ k\ k \subseteq s\ s \subseteq \bigcup(\lambda x.\ ball\ x\ e)\ `\ k$ **by** *auto*

have *finite* (*bb* ` *k*) **using** *k*(*1*) **by** *auto*
moreover
{ fix x **assume** $x{\in}s$
hence $x{\in}\bigcup(\lambda x.\ ball\ x\ e)\ `\ k$ **using** *k*(*3*) **unfolding** *subset-eq* **by** *auto*
hence $\exists\, X{\in}bb\ `\ k.\ x \in X$ **using** *bb k*(*2*) **by** *blast*

hence $x \in \bigcup (bb \mathbin{`} k)$ **using** *Union-iff*[*of x bb ` k*] **by** *auto*
}
ultimately show $\exists f' \subseteq f.$ *finite* $f' \wedge s \subseteq \bigcup f'$ **using** *bb k*(*2*) **by** (*rule-tac x=bb ` k* **in** *exI*) *auto*
qed

64.25 Bolzano-Weierstrass property.

lemma *heine-borel-imp-bolzano-weierstrass*:
assumes $\forall f.\ (\forall t \in f.\ open\ t) \wedge s \subseteq (\bigcup f) \longrightarrow (\exists f'.\ f' \subseteq f \wedge finite\ f' \wedge s \subseteq (\bigcup f'))$
infinite t $t \subseteq s$
shows $\exists x \in s.\ x\ islimpt\ t$
proof(*rule ccontr*)
assume $\neg\ (\exists x \in s.\ x\ islimpt\ t)$
then obtain f **where** $f{:}\forall x{\in}s.\ x \in f\,x \wedge open\ (f\,x) \wedge (\forall y{\in}t.\ y \in f\,x \longrightarrow y = x)$ **unfolding** *islimpt-def*
using *bchoice*[*of s* $\lambda\ x\ T.\ x \in T \wedge open\ T \wedge (\forall y{\in}t.\ y \in T \longrightarrow y = x)$] **by** *auto*
obtain g **where** $g{:}g \subseteq \{t.\ \exists x.\ x \in s \wedge t = f\,x\}$ *finite g* $s \subseteq \bigcup g$
using *assms*(*1*)[*THEN spec*[**where** $x{=}\{t.\ \exists x.\ x{\in}s \wedge t = f\,x\}$]] **using** f **by** *auto*
from *g*(*1*,*3*) **have** $g'{:}\forall x{\in}g.\ \exists xa \in s.\ x = f\,xa$ **by** *auto*
{ **fix** *x y* **assume** $x{\in}t\ y{\in}t\ f\,x = f\,y$
hence $x \in f\,x\ \ y \in f\,x \longrightarrow y = x$ **using** *f*[*THEN bspec*[**where** *x=x*]] **and** ‹$t \subseteq s$› **by** *auto*
hence $x = y$ **using** ‹$f\,x = f\,y$› **and** *f*[*THEN bspec*[**where** *x=y*]] **and** ‹$y{\in}t$› **and** ‹$t \subseteq s$› **by** *auto* }
hence *infinite* (*f ` t*) **using** *assms*(*2*) **using** *finite-imageD*[*unfolded inj-on-def*, *of f t*] **by** *auto*
moreover
{ **fix** *x* **assume** $x{\in}t\ f\,x \notin g$
from *g*(*3*) *assms*(*3*) ‹$x{\in}t$› **obtain** *h* **where** $h{\in}g$ **and** $x{\in}h$ **by** *auto*
then obtain *y* **where** $y{\in}s\ h = f\,y$ **using** g'[*THEN bspec*[**where** *x=h*]] **by** *auto*
hence $y = x$ **using** *f*[*THEN bspec*[**where** *x=y*]] **and** ‹$x{\in}t$› **and** ‹$x{\in}h$›[*unfolded* ‹$h = f\,y$›] **by** *auto*
hence *False* **using** ‹$f\,x \notin g$› ‹$h{\in}g$› **unfolding** ‹$h = f\,y$› **by** *auto* }
hence $f \mathbin{`} t \subseteq g$ **by** *auto*
ultimately show *False* **using** *g*(*2*) **using** *finite-subset* **by** *auto*
qed

64.26 Complete the chain of compactness variants.

primrec *helper-2*::(*real* ⇒ *real^'n*::*finite*) ⇒ *nat* ⇒ *real ^'n* **where**
helper-2 beyond 0 = beyond 0 |
helper-2 beyond (*Suc n*) = *beyond* (*norm* (*helper-2 beyond n*) + *1*)

lemma *bolzano-weierstrass-imp-bounded*: **fixes** *s*::(*real^'n*::*finite*) *set*
assumes $\forall t.\ infinite\ t \wedge t \subseteq s \longrightarrow (\exists x \in s.\ x\ islimpt\ t)$

shows *bounded s*
proof(*rule ccontr*)
assume ¬ *bounded s*
then obtain *beyond* **where** ∀ *a*. *beyond a* ∈*s* ∧ ¬ *norm* (*beyond a*) ≤ *a*
unfolding *bounded-def* **apply** *simp* **using** *choice*[*of* λ*a x*. *x*∈*s* ∧ ¬ *norm x* ≤ *a*] **by** *auto*
hence *beyond*:⋀*a*. *beyond a* ∈*s* ⋀*a*. *norm* (*beyond a*) > *a* **unfolding** *linorder-not-le* **by** *auto*
def *x* ≡ *helper-2 beyond*

{ **fix** *m n* ::*nat* **assume** *m*<*n*
hence *norm* (*x m*) + *1* < *norm* (*x n*)
proof(*induct n*)
case *0* **thus** *?case* **by** *auto*
next
case (*Suc n*)
have ∗:*norm* (*x n*) + *1* < *norm* (*x* (*Suc n*)) **unfolding** *x-def* **and** *helper-2.simps*
using *beyond*(*2*)[*of norm* (*helper-2 beyond n*) + *1*] **by** *auto*
thus *?case* **proof**(*cases m* < *n*)
case *True* **thus** *?thesis* **using** *Suc* **and** ∗ **by** *auto*
next
case *False* **hence** *m* = *n* **using** *Suc*(*2*) **by** *auto*
thus *?thesis* **using** ∗ **by** *auto*
qed
qed } **note** ∗ = *this*
{ **fix** *m n* ::*nat* **assume** *m*≠*n*
have *1* < *dist* (*x m*) (*x n*)
proof(*cases m*<*n*)
case *True*
hence *1* < *norm* (*x n*) − *norm* (*x m*) **using** ∗[*of m n*] **by** *auto*
thus *?thesis* **unfolding** *dist-sym*[*of x m x n*] **unfolding** *dist-def* **using** *norm-triangle-sub*[*of x n x m*] **by** *auto*
next
case *False* **hence** *n*<*m* **using** ‹*m*≠*n*› **by** *auto*
hence *1* < *norm* (*x m*) − *norm* (*x n*) **using** ∗[*of n m*] **by** *auto*
thus *?thesis* **unfolding** *dist-sym*[*of x n x m*] **unfolding** *dist-def* **using** *norm-triangle-sub*[*of x m x n*] **by** *auto*
qed } **note** ∗∗ = *this*
{ **fix** *a b* **assume** *x a* = *x b a* ≠ *b*
hence *False* **using** ∗∗[*of a b*] **unfolding** *dist-eq-0*[*THEN sym*] **by** *auto* }
hence *inj x* **unfolding** *inj-on-def* **by** *auto*
moreover
{ **fix** *n*::*nat*
have *x n* ∈ *s*
proof(*cases n* = *0*)
case *True* **thus** *?thesis* **unfolding** *x-def* **using** *beyond* **by** *auto*
next
case *False* **then obtain** *z* **where** *n* = *Suc z* **using** *not0-implies-Suc* **by** *auto*
thus *?thesis* **unfolding** *x-def* **using** *beyond* **by** *auto*

qed }
ultimately have *infinite* (*range x*) ∧ *range x* ⊆ *s* **unfolding** *x-def* **using** *range-inj-infinite*[*of helper-2 beyond*] **using** *beyond*(*1*) **by** *auto*

then obtain *l* **where** *l*∈*s* **and** *l*:*l islimpt range x* **using** *assms*[*THEN spec*[**where** *x*=*range x*]] **by** *auto*
then obtain *y* **where** *x y* ≠ *l* **and** *y*:*dist* (*x y*) *l* < *1*/*2* **unfolding** *islimpt-approachable* **apply**(*erule-tac x*=*1*/*2* **in** *allE*) **by** *auto*
then obtain *z* **where** *x z* ≠ *l* **and** *z*:*dist* (*x z*) *l* < *dist* (*x y*) *l* **using** *l*[*unfolded islimpt-approachable*, *THEN spec*[**where** *x*=*dist* (*x y*) *l*]]
unfolding *dist-nz* **by** *auto*
show *False* **using** *y* **and** *z* **and** *dist-triangle-half-l*[*of x y l 1 x z*] **and** ∗∗[*of y z*] **by** *auto*
qed

lemma *sequence-infinite-lemma*:
assumes ∀ *n*::*nat*. (*f n* ≠ *l*) (*f* −−−> *l*) *sequentially*
shows *infinite* {*y*::*real*^'*a*::*finite*. (∃ *n*. *y* = *f n*)}
proof(*rule ccontr*)
let *?A* = (λ*x*. *dist x l*) ‘ {*y*. ∃ *n*. *y* = *f n*}
assume ¬ *infinite* {*y*. ∃ *n*. *y* = *f n*}
hence ∗∗:*finite ?A ?A* ≠ {} **by** *auto*
obtain *k* **where** *k*:*dist* (*f k*) *l* = *Min ?A* **using** *Min-in*[*OF* ∗∗] **by** *auto*
have *0* < *Min ?A* **using** *assms*(*1*) **unfolding** *dist-nz* **unfolding** *Min-gr-iff*[*OF* ∗∗] **by** *auto*
then obtain *N* **where** *dist* (*f N*) *l* < *Min ?A* **using** *assms*(*2*)[*unfolded Lim-sequentially*, *THEN spec*[**where** *x*=*Min ?A*]] **by** *auto*
moreover have *dist* (*f N*) *l* ∈ *?A* **by** *auto*
ultimately show *False* **using** *Min-le*[*OF* ∗∗(*1*), *of dist* (*f N*) *l*] **by** *auto*
qed

lemma *sequence-unique-limpt*:
assumes ∀ *n*::*nat*. (*f n* ≠ *l*) (*f* −−−> *l*) *sequentially* *l'* *islimpt* {*y*. (∃ *n*. *y* = *f n*)}
shows *l'* = *l*
proof(*rule ccontr*)
def *e* ≡ *dist l' l*
assume *l'* ≠ *l* **hence** *e*>*0* **unfolding** *dist-nz e-def* **by** *auto*
then obtain *N*::*nat* **where** *N*:∀ *n*≥*N*. *dist* (*f n*) *l* < *e* / *2*
using *assms*(*2*)[*unfolded Lim-sequentially*, *THEN spec*[**where** *x*=*e*/*2*]] **by** *auto*
def *d* ≡ *Min* (*insert* (*e*/*2*) ((λ*n*. *if dist* (*f n*) *l'* = *0 then e*/*2 else dist* (*f n*) *l'*) ‘ {*0* .. *N*}))
have *d*>*0* **using** ‹*e*>*0*› **unfolding** *d-def e-def* **using** *dist-pos-le*[*of - l'*, *unfolded order-le-less*] **by** *auto*
obtain *k* **where** *k*:*f k* ≠ *l'* *dist* (*f k*) *l'* < *d* **using** ‹*d*>*0*› **and** *assms*(*3*)[*unfolded islimpt-approachable*, *THEN spec*[**where** *x*=*d*]] **by** *auto*
have *k*≥*N* **using** *k*(*1*)[*unfolded dist-nz*] **using** *k*(*2*)[*unfolded d-def*]
by *force*
hence *dist l' l* < *e* **using** *N*[*THEN spec*[**where** *x*=*k*]] **using** *k*(*2*)[*unfolded d-def*]

and *dist-triangle-half-r*[*of f k l' e l*] **by** *auto*
thus *False* **unfolding** *e-def* **by** *auto*
qed

lemma *bolzano-weierstrass-imp-closed*:
assumes $\forall t.$ *infinite* $t \wedge t \subseteq s \longrightarrow (\exists x \in s.\ x$ *islimpt* $t)$
shows *closed s*
proof−
{ fix *x l* **assume** *as*: $\forall n$::*nat*. *x n* $\in s$ (*x* −−−> *l*) *sequentially*
hence $l \in s$
proof(*cases* $\forall n.\ x\ n \neq l$)
case *False* **thus** *l*∈*s* **using** *as*(*1*) **by** *auto*
next
case *True* **note** *cas* = *this*
with *as*(*2*) **have** *infinite* $\{y.\ \exists n.\ y = x\ n\}$ **using** *sequence-infinite-lemma*[*of x l*] **by** *auto*
then obtain *l'* **where** *l'*∈*s l' islimpt* $\{y.\ \exists n.\ y = x\ n\}$ **using** *assms*[*THEN spec*[**where** *x*=$\{y.\ \exists n.\ y = x\ n\}$]] *as*(*1*) **by** *auto*
thus *l*∈*s* **using** *sequence-unique-limpt*[*of x l l'*] **using** *as cas* **by** *auto*
qed **}**
thus *?thesis* **unfolding** *closed-sequential-limits* **by** *auto*
qed

Hence express everything as an equivalence.

lemma *compact-eq-heine-borel*: *compact s* ⟷
$(\forall f.\ (\forall t \in f.$ *open* $t) \wedge s \subseteq (\bigcup f)$
−−> $(\exists f'.\ f' \subseteq f \wedge$ *finite* $f' \wedge s \subseteq (\bigcup f'))$) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **using** *compact-imp-heine-borel*[*of s*] **by** *blast*
next
assume *?rhs*
hence $\forall t.$ *infinite* $t \wedge t \subseteq s \longrightarrow (\exists x \in s.\ x$ *islimpt* $t)$ **using** *heine-borel-imp-bolzano-weierstrass*[*of s*] **by** *blast*
thus *?lhs* **using** *bolzano-weierstrass-imp-bounded*[*of s*] *bolzano-weierstrass-imp-closed*[*of s*] *bounded-closed-imp-compact*[*of s*] **by** *blast*
qed

lemma *compact-eq-bolzano-weierstrass*:
compact s ⟷ $(\forall t.$ *infinite* $t \wedge t \subseteq s$ −−> $(\exists x \in s.\ x$ *islimpt* $t))$ (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **unfolding** *compact-eq-heine-borel* **using** *heine-borel-imp-bolzano-weierstrass*[*of s*] **by** *auto*
next
assume *?rhs* **thus** *?lhs* **using** *bolzano-weierstrass-imp-bounded bolzano-weierstrass-imp-closed bounded-closed-imp-compact* **by** *auto*
qed

lemma *compact-eq-bounded-closed*:

compact s ⟷ *bounded s* ∧ *closed s* (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **unfolding** *compact-eq-bolzano-weierstrass* **using** *bolzano-weierstrass-imp-bounded bolzano-weierstrass-imp-closed* **by** *auto*
next
assume *?rhs* **thus** *?lhs* **using** *bounded-closed-imp-compact* **by** *auto*
qed

lemma *compact-imp-bounded*:
compact s ==> *bounded s*
unfolding *compact-eq-bounded-closed*
by *simp*

lemma *compact-imp-closed*:
compact s ==> *closed s*
unfolding *compact-eq-bounded-closed*
by *simp*

In particular, some common special cases.

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

lemma *compact-union*[*intro*]:
compact s ⟹ *compact t* ==> *compact* (*s* ∪ *t*)
unfolding *compact-eq-bounded-closed*
using *bounded-Un*[*of s t*]
using *closed-Un*[*of s t*]
by *simp*

lemma *compact-inter*[*intro*]:
compact s ⟹ *compact t* ==> *compact* (*s* ∩ *t*)
unfolding *compact-eq-bounded-closed*
using *bounded-Int*[*of s t*]
using *closed-Int*[*of s t*]
by *simp*

lemma *compact-inter-closed*[*intro*]:
compact s ⟹ *closed t* ==> *compact* (*s* ∩ *t*)
unfolding *compact-eq-bounded-closed*
using *closed-Int*[*of s t*]
using *bounded-subset*[*of s* ∩ *t s*]
by *blast*

lemma *closed-inter-compact*[*intro*]:
closed s ⟹ *compact t* ==> *compact* (*s* ∩ *t*)
proof−

assume *closed s compact t*
moreover
have $s \cap t = t \cap s$ **by** *auto* **ultimately**
show *?thesis*
using *compact-inter-closed*[*of t s*]
by *auto*
qed

lemma *finite-imp-closed*:
finite s ==> closed s
proof−
assume *finite s* **hence** ¬(∃*t*. $t \subseteq s \wedge$ *infinite t*) **using** *finite-subset* **by** *auto*
thus *?thesis* **using** *bolzano-weierstrass-imp-closed*[*of s*] **by** *auto*
qed

lemma *finite-imp-compact*:
finite s ==> compact s
unfolding *compact-eq-bounded-closed*
using *finite-imp-closed finite-imp-bounded*
by *blast*

lemma *compact-sing*[*simp*]:
compact {*a*}
using *finite-imp-compact*[*of* {*a*}]
by *blast*

lemma *closed-sing*[*simp*]:
closed {*a*}
using *compact-eq-bounded-closed compact-sing*[*of a*]
by *blast*

lemma *compact-cball*[*simp*]:
compact(*cball x e*)
using *compact-eq-bounded-closed bounded-cball closed-cball*
by *blast*

lemma *compact-frontier-bounded*[*intro*]:
bounded s ==> compact(*frontier s*)
unfolding *frontier-def*
using *compact-eq-bounded-closed*
by *blast*

lemma *compact-frontier*[*intro*]:
compact s ==> compact (*frontier s*)
using *compact-eq-bounded-closed compact-frontier-bounded*
by *blast*

lemma *frontier-subset-compact*:
compact s ==> frontier s $\subseteq$ *s*

using *frontier-subset-closed compact-eq-bounded-closed*
by *blast*

lemma *open-delete*:
open s ==> open$(s - \{x\})$
using *open-diff*[*of s* $\{x\}$] *closed-sing*
by *blast*

Finite intersection property. I could make it an equivalence in fact.

lemma *compact-imp-fip*:
assumes *compact s* $\forall t \in f$. *closed t*
$\forall f'$. *finite* $f' \wedge f' \subseteq f$ $--> (s \cap (\bigcap f') \neq \{\})$
shows $s \cap (\bigcap f) \neq \{\}$
proof
assume *as*:$s \cap (\bigcap f) = \{\}$
hence $s \subseteq \bigcup op - UNIV$ ' f **by** *auto*
moreover have *Ball* $(op - UNIV$ ' $f)$ *open* **using** *open-diff closed-diff* **using** *assms*(*2*) **by** *auto*
ultimately obtain f' **where** f':$f' \subseteq op - UNIV$ ' f *finite* f' $s \subseteq \bigcup f'$ **using** *assms*(*1*)[*unfolded compact-eq-heine-borel*, *THEN spec*[**where** x=$(\lambda t.\ UNIV - t)$ ' f]] **by** *auto*
hence *finite* $(op - UNIV$ ' $f') \wedge op - UNIV$ ' $f' \subseteq f$ **by**(*auto simp add*: *Diff-Diff-Int*)
hence $s \cap \bigcap op - UNIV$ ' $f' \neq \{\}$ **using** *assms*(*3*)[*THEN spec*[**where** x=*op* − *UNIV* ' f']] **by** *auto*
thus *False* **using** f'(*3*) **unfolding** *subset-eq* **and** *Union-iff* **by** *blast*
qed

64.27 Bounded closed nest property (proof does not use Heine-Borel).

lemma *bounded-closed-nest*:
assumes $\forall n.$ *closed*$(s\ n)$ $\forall n.$ $(s\ n \neq \{\})$
$(\forall m\ n.\ m \leq n$ $--> s\ n \subseteq s\ m)$ *bounded*$(s\ 0)$
shows $\exists$ a::*real*^'a::*finite*. $\forall n$::*nat*. $a \in s(n)$
proof−
from *assms*(*2*) **obtain** x **where** x:$\forall n$::*nat*. $x\ n \in s\ n$ **using** *choice*[*of* $\lambda n\ x.\ x \in s\ n$] **by** *auto*
from *assms*(*4*,*1*) **have** $*$:*compact* $(s\ 0)$ **using** *bounded-closed-imp-compact*[*of s 0*] **by** *auto*

then obtain $l\ r$ **where** lr:$l \in s\ 0$ $\forall m\ n.\ m < n \longrightarrow r\ m < r\ n$ $((x \circ r)$ $---> l)$ *sequentially*
unfolding *compact-def* **apply**(*erule-tac* x=x **in** *allE*) **using** x **using** *assms*(*3*)
by *blast*

{ fix n::*nat*
{ fix e::*real* **assume** e>*0*
with lr(*3*) **obtain** N **where** N:$\forall m \geq N.$ *dist* $((x \circ r)\ m)\ l < e$ **unfolding**

Lim-sequentially **by** *auto*
hence *dist* ((*x* ∘ *r*) (*max N n*)) *l* < *e* **by** *auto*
moreover
have *r* (*max N n*) ≥ *n* **using** *lr*(*2*) **using** *monotone-bigger*[*of r max N n*] **by** *auto*
hence (*x* ∘ *r*) (*max N n*) ∈ *s n*
using *x* **apply**(*erule-tac x*=*n* **in** *allE*)
using *x* **apply**(*erule-tac x*=*r* (*max N n*) **in** *allE*)
using *assms*(*3*) **apply**(*erule-tac x*=*n* **in** *allE*)**apply**(*erule-tac x*=*r* (*max N n*) **in** *allE*) **by** *auto*
ultimately have ∃ *y*∈*s n*. *dist y l* < *e* **by** *auto*
}
hence *l* ∈ *s n* **using** *closed-approachable*[*of s n l*] *assms*(*1*) **by** *blast*
}
thus *?thesis* **by** *auto*
qed

Decreasing case does not even need compactness, just completeness.

lemma *decreasing-closed-nest*:
assumes ∀ *n*. *closed*(*s n*)
∀ *n*. (*s n* ≠ {})
∀ *m n*. *m* ≤ *n* −−> *s n* ⊆ *s m*
∀ *e*>*0*. ∃ *n*. ∀ *x* ∈ (*s n*). ∀ *y* ∈ (*s n*). *dist x y* < *e*
shows ∃ *a*::*real*^'*a*::*finite*. ∀ *n*::*nat*. *a* ∈ *s n*
proof−
have ∀ *n*. ∃ *x*. *x*∈*s n* **using** *assms*(*2*) **by** *auto*
hence ∃ *t*. ∀ *n*. *t n* ∈ *s n* **using** *choice*[*of* λ *n x*. *x* ∈ *s n*] **by** *auto*
then obtain *t* **where** *t*: ∀ *n*. *t n* ∈ *s n* **by** *auto*
{ **fix** *e*::*real* **assume** *e*>*0*
then obtain *N* **where** *N*:∀ *x*∈*s N*. ∀ *y*∈*s N*. *dist x y* < *e* **using** *assms*(*4*) **by** *auto*
{ **fix** *m n* ::*nat* **assume** *N* ≤ *m* ∧ *N* ≤ *n*
hence *t m* ∈ *s N t n* ∈ *s N* **using** *assms*(*3*) *t* **unfolding** *subset-eq t* **by** *blast*+
hence *dist* (*t m*) (*t n*) < *e* **using** *N* **by** *auto*
}
hence ∃ *N*. ∀ *m n*. *N* ≤ *m* ∧ *N* ≤ *n* ⟶ *dist* (*t m*) (*t n*) < *e* **by** *auto*
}
hence *cauchy t* **unfolding** *cauchy-def* **by** *auto*
then obtain *l* **where** *l*:(*t* −−−> *l*) *sequentially* **using** *complete-univ* **unfolding** *complete-def* **by** *auto*
{ **fix** *n*::*nat*
{ **fix** *e*::*real* **assume** *e*>*0*
then obtain *N*::*nat* **where** *N*:∀ *n*≥*N*. *dist* (*t n*) *l* < *e* **using** *l*[*unfolded Lim-sequentially*] **by** *auto*
have *t* (*max n N*) ∈ *s n* **using** *assms*(*3*) **unfolding** *subset-eq* **apply**(*erule-tac x*=*n* **in** *allE*) **apply** (*erule-tac x*=*max n N* **in** *allE*) **using** *t* **by** *auto*
hence ∃ *y*∈*s n*. *dist y l* < *e* **apply**(*rule-tac x*=*t* (*max n N*) **in** *bexI*) **using** *N* **by** *auto*

```
   }
   hence l ∈ s n using closed-approachable[of s n l] assms(1) by auto
  }
  then show ?thesis by auto
qed
```

Strengthen it to the intersection actually being a singleton.

```
lemma decreasing-closed-nest-sing:
  assumes ∀ n. closed(s n)
          ∀ n. s n ≠ {}
          ∀ m n. m ≤ n −−> s n ⊆ s m
          ∀ e>0. ∃ n. ∀ x ∈ (s n). ∀ y∈(s n). dist x y < e
  shows ∃ a::real^'a::finite. ⋂ {t. (∃ n::nat. t = s n)} = {a}
proof−
  obtain a where a:∀ n. a ∈ s n using decreasing-closed-nest[of s] using assms
by auto
  { fix b assume b:b ∈ ⋂{t. ∃ n. t = s n}
   { fix e::real assume e>0
    hence dist a b < e using assms(4 )using b using a by blast
   }
   hence dist a b = 0 by (metis dist-eq-0 dist-nz real-less-def)
  }
  with a have ⋂{t. ∃ n. t = s n} = {a}  unfolding dist-eq-0 by auto
  thus ?thesis by auto
qed
```

Cauchy-type criteria for uniform convergence.

```
lemma uniformly-convergent-eq-cauchy: fixes s::nat ⇒ 'b ⇒ real^'a::finite shows
 (∃ l. ∀ e>0. ∃ N. ∀ n x. N ≤ n ∧ P x −−> dist(s n x)(l x) < e) ⟷
  (∀ e>0. ∃ N. ∀ m n x. N ≤ m ∧ N ≤ n ∧ P x  −−> dist (s m x) (s n x) < e)
(is ?lhs = ?rhs)
proof(rule)
  assume ?lhs
  then obtain l where l:∀ e>0. ∃ N. ∀ n x. N ≤ n ∧ P x ⟶ dist (s n x) (l x)
< e by auto
  { fix e::real assume e>0
   then obtain N::nat where N:∀ n x. N ≤ n ∧ P x ⟶ dist (s n x) (l x) < e
/ 2 using l[THEN spec[where x=e/2]] by auto
   { fix n m::nat and x::'b assume N ≤ m ∧ N ≤ n ∧ P x
    hence dist (s m x) (s n x) < e
     using N[THEN spec[where x=m], THEN spec[where x=x]]
     using N[THEN spec[where x=n], THEN spec[where x=x]]
     using dist-triangle-half-l[of s m x l x e s n x] by auto  }
   hence ∃ N. ∀ m n x. N ≤ m ∧ N ≤ n ∧ P x  −−> dist (s m x) (s n x) < e
by auto  }
  thus ?rhs by auto
next
  assume ?rhs
  hence ∀ x. P x ⟶ cauchy (λn. s n x) unfolding cauchy-def apply auto by
(erule-tac x=e in allE)auto
```

then obtain l **where** $l{:}\forall x.\ P\ x \longrightarrow ((\lambda n.\ s\ n\ x) \longrightarrow l\ x)$ *sequentially* **unfolding** *convergent-eq-cauchy*[*THEN sym*]
using *choice*[*of* $\lambda x\ l.\ P\ x \longrightarrow ((\lambda n.\ s\ n\ x) \longrightarrow l)$ *sequentially*] **by** *auto*
{ fix $e{::}$*real* **assume** $e{>}0$
then obtain N **where** $N{:}\forall m\ n\ x.\ N \leq m \wedge N \leq n \wedge P\ x \longrightarrow$ *dist* $(s\ m\ x)\ (s\ n\ x) < e/2$
using ‹*?rhs*›[*THEN spec*[**where** $x{=}e/2$]] **by** *auto*
{ fix x **assume** $P\ x$
then obtain M **where** $M{:}\forall n{\geq}M.$ *dist* $(s\ n\ x)\ (l\ x) < e/2$
using l[*THEN spec*[**where** $x{=}x$], *unfolded Lim-sequentially*] **using** ‹$e{>}0$› **by**(*auto elim*!: *allE*[**where** $x{=}e/2$])
fix $n{::}$*nat* **assume** $n{\geq}N$
hence *dist*$(s\ n\ x)(l\ x) < e$ **using** ‹$P\ x$›**and** N[*THEN spec*[**where** $x{=}n$], *THEN spec*[**where** $x{=}N{+}M$], *THEN spec*[**where** $x{=}x$]]
using M[*THEN spec*[**where** $x{=}N{+}M$]] **and** *dist-triangle-half-l*[*of* $s\ n\ x\ s\ (N{+}M)\ x\ e\ l\ x$] **by** (*auto simp add*: *dist-sym*) **}**
hence $\exists N.\ \forall n\ x.\ N \leq n \wedge P\ x \longrightarrow$ *dist*$(s\ n\ x)(l\ x) < e$ **by** *auto* **}**
thus *?lhs* **by** *auto*
qed

lemma *uniformly-cauchy-imp-uniformly-convergent*:
assumes $\forall e{>}0.\exists N.\ \forall m\ (n{::}nat)\ x.\ N \leq m \wedge N \leq n \wedge P\ x \dashrightarrow$ *dist*$(s\ m\ x)(s\ n\ x) < e$
$\forall x.\ P\ x \dashrightarrow (\forall e{>}0.\ \exists N.\ \forall n.\ N \leq n \dashrightarrow$ *dist*$(s\ n\ x)(l\ x) < e)$
shows $\forall e{>}0.\ \exists N.\ \forall n\ x.\ N \leq n \wedge P\ x \dashrightarrow$ *dist*$(s\ n\ x)(l\ x) < e$
proof−
obtain l' **where** $l{:}\forall e{>}0.\ \exists N.\ \forall n\ x.\ N \leq n \wedge P\ x \longrightarrow$ *dist* $(s\ n\ x)\ (l'\ x) < e$
using *assms*(*1*) **unfolding** *uniformly-convergent-eq-cauchy*[*THEN sym*] **by** *auto*
moreover
{ fix x **assume** $P\ x$
hence $l\ x = l'\ x$ **using** *Lim-unique*[*OF trivial-limit-sequentially*, *of* $\lambda n.\ s\ n\ x\ l\ x\ l'\ x$]
using l **and** *assms*(*2*) **unfolding** *Lim-sequentially* **by** *blast* **}**
ultimately show *?thesis* **by** *auto*
qed

64.28 Define continuity over a net to take in restrictions of the set.

definition *continuous net* $f \longleftrightarrow (f \dashrightarrow f(\mathit{netlimit\ net}))$ *net*

lemma *continuous-trivial-limit*:
trivial-limit net ==> *continuous net f*
unfolding *continuous-def tendsto-def eventually-def* **by** *auto*

lemma *continuous-within*: *continuous* (*at x within s*) $f \longleftrightarrow (f \dashrightarrow f(x))$ (*at x within s*)
unfolding *continuous-def*

unfolding *tendsto-def*
using *netlimit-within*[*of x s*]
unfolding *eventually-def*
by (*cases trivial-limit* (*at x within s*)) *auto*

lemma *continuous-at*: *continuous* (*at x*) *f* ⟷ (*f* −−−> *f*(*x*)) (*at x*) **using** *within-UNIV*[*of x*]
using *continuous-within*[*of x UNIV f*] **by** *auto*

lemma *continuous-at-within*:
assumes *continuous* (*at x*) *f* **shows** *continuous* (*at x within s*) *f*
proof(*cases x islimpt s*)
case *True* **show** *?thesis* **using** *assms* **unfolding** *continuous-def* **and** *netlimit-at*
using *Lim-at-within*[*of f f x x s*]
unfolding *netlimit-within*[*unfolded trivial-limit-within not-not*, *OF True*] **by** *blast*
next
case *False* **thus** *?thesis* **unfolding** *continuous-def* **and** *netlimit-at*
unfolding *Lim* **and** *trivial-limit-within* **by** *auto*
qed

Derive the epsilon-delta forms, which we often use as "definitions"

lemma *continuous-within-eps-delta*:
continuous (*at x within s*) *f* ⟷ (∀ *e*>*0*. ∃ *d*>*0*. ∀ *x'*∈ *s*. *dist x' x* < *d* −−> *dist* (*f x'*) (*f x*) < *e*)
unfolding *continuous-within* **and** *Lim-within*
apply *auto* **unfolding** *dist-nz*[*THEN sym*] **apply**(*auto elim*!:*allE*) **apply**(*rule-tac x*=*d* **in** *exI*) **by** *auto*

lemma *continuous-at-eps-delta*: *continuous* (*at x*) *f* ⟷ (∀ *e*>*0*. ∃ *d*>*0*. ∀ *x'*. *dist x' x* < *d* −−> *dist*(*f x'*)(*f x*) < *e*)
using *continuous-within-eps-delta*[*of x UNIV f*]
unfolding *within-UNIV* **by** *blast*

Versions in terms of open balls.

lemma *continuous-within-ball*:
continuous (*at x within s*) *f* ⟷ (∀ *e*>*0*. ∃ *d*>*0*. *f* ‘ (*ball x d* ∩ *s*) ⊆ *ball* (*f x*) *e*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ fix *e*::*real* **assume** *e*>*0*
then obtain *d* **where** *d*: *d*>*0* ∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *dist* (*f xa*) (*f x*) < *e*
using ‹*?lhs*›[*unfolded continuous-within Lim-within*] **by** *auto*
{ fix *y* **assume** *y*∈*f* ‘ (*ball x d* ∩ *s*)
hence *y* ∈ *ball* (*f x*) *e* **using** *d*(*2*) **unfolding** *dist-nz*[*THEN sym*]
apply (*auto simp add*: *dist-sym mem-ball*) **apply**(*erule-tac x*=*xa* **in** *ballE*)
apply *auto* **unfolding** *dist-refl* **using** ‹*e*>*0*› **by** *auto*
}

hence $\exists\, d{>}0.\ f\ `\ (ball\ x\ d \cap s) \subseteq ball\ (f\ x)\ e$ **using** ‹$d{>}0$› **unfolding** *subset-eq ball-def* **by** (*auto simp add*: *dist-sym*) **}**
thus *?rhs* **by** *auto*
next
assume *?rhs* **thus** *?lhs* **unfolding** *continuous-within Lim-within ball-def subset-eq*
apply (*auto simp add*: *dist-sym*) **apply**(*erule-tac x=e* **in** *allE*) **by** *auto*
qed

lemma *continuous-at-ball*: **fixes** *f*::*real^'a*::*finite* ⇒ *real^'a*
shows *continuous* (*at x*) *f* ⟷ (∀ *e*>0. ∃ *d*>0. *f* ` (*ball x d*) ⊆ *ball* (*f x*) *e*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **thus** *?rhs* **unfolding** *continuous-at Lim-at subset-eq Ball-def Bex-def image-iff mem-ball*
apply *auto* **apply**(*erule-tac x=e* **in** *allE*) **apply** *auto* **apply**(*rule-tac x=d* **in** *exI*) **apply** *auto* **apply**(*erule-tac x=xa* **in** *allE*) **apply** (*auto simp add*: *dist-refl dist-sym dist-nz*)
unfolding *dist-nz*[*THEN sym*] **by** (*auto simp add*: *dist-refl*)
next
assume *?rhs* **thus** *?lhs* **unfolding** *continuous-at Lim-at subset-eq Ball-def Bex-def image-iff mem-ball*
apply *auto* **apply**(*erule-tac x=e* **in** *allE*) **apply** *auto* **apply**(*rule-tac x=d* **in** *exI*) **apply** *auto* **apply**(*erule-tac x=f xa* **in** *allE*) **by** (*auto simp add*: *dist-refl dist-sym dist-nz*)
qed

For setwise continuity, just start from the epsilon-delta definitions.

definition *continuous-on s f* ⟷ (∀ *x* ∈ *s*. ∀ *e*>0. ∃ *d*::*real*>0. ∀ *x'* ∈ *s*. *dist x' x* < *d* −−> *dist* (*f x'*) (*f x*) < *e*)

definition *uniformly-continuous-on s f* ⟷
(∀ *e*>0. ∃ *d*>0. ∀ *x*∈*s*. ∀ *x'*∈*s*. *dist x' x* < *d* −−> *dist* (*f x'*) (*f x*) < *e*)

Some simple consequential lemmas.

lemma *uniformly-continuous-imp-continuous*:
uniformly-continuous-on s f ==> *continuous-on s f*
unfolding *uniformly-continuous-on-def continuous-on-def* **by** *blast*

lemma *continuous-at-imp-continuous-within*:
continuous (*at x*) *f* ==> *continuous* (*at x within s*) *f*
unfolding *continuous-within continuous-at* **using** *Lim-at-within* **by** *auto*

lemma *continuous-at-imp-continuous-on*: **assumes** (∀ *x* ∈ *s*. *continuous* (*at x*) *f*)
shows *continuous-on s f*
proof(*simp add*: *continuous-at continuous-on-def*, *rule*, *rule*, *rule*)
fix *x* **and** *e*::*real* **assume** *x*∈*s e*>0
hence *eventually* (λ*xa*. *dist* (*f xa*) (*f x*) < *e*) (*at x*) **using** *assms* **unfolding** *continuous-at tendsto-def* **by** *auto*

then obtain d **where** $d{:}d{>}0\ \forall xa.\ 0 < dist\ xa\ x \wedge dist\ xa\ x < d \longrightarrow dist\ (f\ xa)\ (f\ x) < e$ **unfolding** *eventually-at* **by** *auto*
{ fix x' **assume** $\neg\ 0 < dist\ x'\ x$
hence $x{=}x'$
using *dist-nz*[*of* $x'\ x$] **by** *auto*
hence $dist\ (f\ x')\ (f\ x) < e$ **using** *dist-refl*[*of* $f\ x'$] ‹$e{>}0$› **by** *auto*
}
thus $\exists\, d{>}0.\ \forall\, x'{\in}s.\ dist\ x'\ x < d \longrightarrow dist\ (f\ x')\ (f\ x) < e$ **using** d **by** *auto*
qed

lemma *continuous-on-eq-continuous-within*:
continuous-on s f $\longleftrightarrow$ $(\forall\, x \in s.\ continuous\ (at\ x\ within\ s)\ f)$ (**is** *?lhs* = *?rhs*)
proof
assume *?rhs*
{ fix x **assume** $x{\in}s$
fix $e{::}real$ **assume** $e{>}0$
assume $\exists\, d{>}0.\ \forall\, xa{\in}s.\ 0 < dist\ xa\ x \wedge dist\ xa\ x < d \longrightarrow dist\ (f\ xa)\ (f\ x) < e$
then obtain d **where** $d{>}0$ **and** $d{:}\forall\, xa{\in}s.\ 0 < dist\ xa\ x \wedge dist\ xa\ x < d \longrightarrow dist\ (f\ xa)\ (f\ x) < e$ **by** *auto*
{ fix x' **assume** *as*:$x'{\in}s$ $dist\ x'\ x < d$
hence $dist\ (f\ x')\ (f\ x) < e$ **using** *dist-refl*[*of f x'*] ‹$e{>}0$› d ‹$x'{\in}s$› *dist-eq-0*[*of* $x'\ x$] *dist-pos-le*[*of* $x'\ x$] *as*(*2*) **by** (*metis dist-eq-0 dist-nz*) **}**
hence $\exists\, d{>}0.\ \forall\, x'{\in}s.\ dist\ x'\ x < d \longrightarrow dist\ (f\ x')\ (f\ x) < e$ **using** ‹$d{>}0$› **by** (*auto simp add*: *dist-refl*)
}
thus *?lhs* **using** ‹*?rhs*› **unfolding** *continuous-on-def continuous-within Lim-within* **by** *auto*
next
assume *?lhs*
thus *?rhs* **unfolding** *continuous-on-def continuous-within Lim-within* **by** *blast*
qed

lemma *continuous-on*:
continuous-on s f $\longleftrightarrow$ $(\forall\, x \in s.\ (f\ ---{>}\ f(x))\ (at\ x\ within\ s))$
by (*auto simp add*: *continuous-on-eq-continuous-within continuous-within*)

lemma *continuous-on-eq-continuous-at*:
open s ==> (*continuous-on s f* $\longleftrightarrow$ $(\forall\, x \in s.\ continuous\ (at\ x)\ f)$)
by (*auto simp add*: *continuous-on continuous-at Lim-within-open*)

lemma *continuous-within-subset*:
continuous $(at\ x\ within\ s)\ f \Longrightarrow t \subseteq s$
==> *continuous* $(at\ x\ within\ t)\ f$
unfolding *continuous-within* **by**(*metis Lim-within-subset*)

lemma *continuous-on-subset*:
continuous-on s f $\Longrightarrow$ *t* $\subseteq$ *s* ==> *continuous-on t f*
unfolding *continuous-on* **by** (*metis subset-eq Lim-within-subset*)

lemma *continuous-on-interior*:
continuous-on s f $\Longrightarrow$ *x* $\in$ *interior s* ==> *continuous* (*at x*) *f*
unfolding *interior-def*
apply *simp*
by (*meson continuous-on-eq-continuous-at continuous-on-subset*)

lemma *continuous-on-eq*:
($\forall$ *x* $\in$ *s*. *f x* = *g x*) $\Longrightarrow$ *continuous-on s f*
==> *continuous-on s g*
by (*simp add*: *continuous-on-def*)

Characterization of various kinds of continuity in terms of sequences.

lemma *continuous-within-sequentially*:
continuous (*at a within s*) *f* $\longleftrightarrow$
($\forall$ *x*. ($\forall$ *n*::*nat*. *x n* $\in$ *s*) $\wedge$ (*x* ---> *a*) *sequentially*
---> ((*f o x*) ---> *f a*) *sequentially*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ fix *x*::*nat* $\Rightarrow$ *real*^'*a* **assume** *x*:$\forall$ *n*. *x n* $\in$ *s* $\forall$ *e*>*0*. $\exists$ *N*. $\forall$ *n*$\geq$*N*. *dist* (*x n*) *a* < *e*
fix *e*::*real* **assume** *e*>*0*
from ‹*?lhs*› **obtain** *d* **where** *d*>*0* **and** *d*:$\forall$ *x*$\in$*s*. *0* < *dist x a* $\wedge$ *dist x a* < *d* $\longrightarrow$ *dist* (*f x*) (*f a*) < *e* **unfolding** *continuous-within Lim-within* **using** ‹*e*>*0*› **by** *auto*
from *x*(*2*) ‹*d*>*0*› **obtain** *N* **where** *N*:$\forall$ *n*$\geq$*N*. *dist* (*x n*) *a* < *d* **by** *auto*
hence $\exists$ *N*. $\forall$ *n*$\geq$*N*. *dist* ((*f* $\circ$ *x*) *n*) (*f a*) < *e*
apply(*rule-tac x*=*N* **in** *exI*) **using** *N d* **apply** *auto* **using** *x*(*1*)
apply(*erule-tac x*=*n* **in** *allE*) **apply**(*erule-tac x*=*n* **in** *allE*)
apply(*erule-tac x*=*x n* **in** *ballE*) **apply** *auto* **unfolding** *dist-nz*[*THEN sym*] **apply** *auto* **unfolding** *dist-refl* **using** ‹*e*>*0*› **by** *auto*
}
thus *?rhs* **unfolding** *continuous-within* **unfolding** *Lim-sequentially* **by** *simp*
next
assume *?rhs*
{ fix *e*::*real* **assume** *e*>*0*
assume ¬ ($\exists$ *d*>*0*. $\forall$ *x*$\in$*s*. *0* < *dist x a* $\wedge$ *dist x a* < *d* $\longrightarrow$ *dist* (*f x*) (*f a*) < *e*)
hence $\forall$ *d*. $\exists$ *x*. *d*>*0* $\longrightarrow$ *x*$\in$*s* $\wedge$ (*0* < *dist x a* $\wedge$ *dist x a* < *d* $\wedge$ ¬ *dist* (*f x*) (*f a*) < *e*) **by** *blast*
then obtain *x* **where** *x*:$\forall$ *d*>*0*. *x d* $\in$ *s* $\wedge$ (*0* < *dist* (*x d*) *a* $\wedge$ *dist* (*x d*) *a* < *d* $\wedge$ ¬ *dist* (*f* (*x d*)) (*f a*) < *e*)
using *choice*[*of* λ*d x*.*0*<*d* $\longrightarrow$ *x*$\in$*s* $\wedge$ (*0* < *dist x a* $\wedge$ *dist x a* < *d* $\wedge$ ¬ *dist* (*f x*) (*f a*) < *e*)] **by** *auto*
{ fix *d*::*real* **assume** *d*>*0*
hence $\exists$ *N*::*nat*. *inverse* (*real* (*N* + *1*)) < *d* **using** *real-arch-inv*[*of d*] **by** (*auto*, *rule-tac x*=*n* − *1* **in** *exI*)*auto*
then obtain *N*::*nat* **where** *N*:*inverse* (*real* (*N* + *1*)) < *d* **by** *auto*
{ fix *n*::*nat* **assume** *n*:*n*$\geq$*N*
hence *dist* (*x* (*inverse* (*real* (*n* + *1*)))) *a* < *inverse* (*real* (*n* + *1*)) **using** *x*[*THEN spec*[**where** *x*=*inverse* (*real* (*n* + *1*))]] **by** *auto*

moreover have *inverse* (*real* (*n* + *1*)) < *d* **using** *N n* **by** (*auto*, *metis Suc-le-mono le-SucE less-imp-inverse-less nat-le-real-less order-less-trans real-of-nat-Suc real-of-nat-Suc-gt-zero*)
ultimately have *dist* (*x* (*inverse* (*real* (*n* + *1*)))) *a* < *d* **by** *auto*
}
hence ∃ *N*::*nat*. ∀ *n*≥*N*. *dist* (*x* (*inverse* (*real* (*n* + *1*)))) *a* < *d* **by** *auto*
}
hence (∀ *n*::*nat*. *x* (*inverse* (*real* (*n* + *1*))) ∈ *s*) ∧ (∀ *e*>*0*. ∃ *N*::*nat*. ∀ *n*≥*N*. *dist* (*x* (*inverse* (*real* (*n* + *1*)))) *a* < *e*) **using** *x* **by** *auto*
hence ∀ *e*>*0*. ∃ *N*::*nat*. ∀ *n*≥*N*. *dist* (*f* (*x* (*inverse* (*real* (*n* + *1*))))) (*f a*) < *e* **using** ‹*?rhs*›[*THEN spec*[**where** *x*=λ*n*::*nat*. *x* (*inverse* (*real* (*n*+*1*)))], *unfolded Lim-sequentially*] **by** *auto*
hence *False* **apply**(*erule-tac x*=*e* **in** *allE*) **using** ‹*e*>*0*› **using** *x* **by** *auto*
}
thus *?lhs* **unfolding** *continuous-within* **unfolding** *Lim-within* **unfolding** *Lim-sequentially* **by** *blast*
qed

lemma *continuous-at-sequentially*:
continuous (*at a*) *f* ⟷ (∀ *x*. (*x* −−−> *a*) *sequentially* −−> ((*f o x*) −−−> *f a*) *sequentially*)
using *continuous-within-sequentially*[*of a UNIV f*] **unfolding** *within-UNIV* **by** *auto*

lemma *continuous-on-sequentially*:
continuous-on s f ⟷ (∀ *x*. ∀ *a* ∈ *s*. (∀ *n*. *x*(*n*) ∈ *s*) ∧ (*x* −−−> *a*) *sequentially* −−> ((*f o x*) −−−> *f*(*a*)) *sequentially*) (**is** *?lhs* = *?rhs*)
proof
assume *?rhs* **thus** *?lhs* **using** *continuous-within-sequentially*[*of - s f*] **unfolding** *continuous-on-eq-continuous-within* **by** *auto*
next
assume *?lhs* **thus** *?rhs* **unfolding** *continuous-on-eq-continuous-within* **using** *continuous-within-sequentially*[*of - s f*] **by** *auto*
qed

lemma *uniformly-continuous-on-sequentially*:
uniformly-continuous-on s f ⟷ (∀ *x y*. (∀ *n*. *x n* ∈ *s*) ∧ (∀ *n*. *y n* ∈ *s*) ∧ ((λ*n*. *x n* − *y n*) −−−> *0*) *sequentially* ⟶ ((λ*n*. *f*(*x n*) − *f*(*y n*)) −−−> *0*) *sequentially*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ **fix** *x y* **assume** *x*:∀ *n*. *x n* ∈ *s* **and** *y*:∀ *n*. *y n* ∈ *s* **and** *xy*:((λ*n*. *x n* − *y n*) −−−> *0*) *sequentially*
{ **fix** *e*::*real* **assume** *e*>*0*
then obtain *d* **where** *d*>*0* **and** *d*:∀ *x*∈*s*. ∀ *x*′∈*s*. *dist x*′ *x* < *d* ⟶ *dist* (*f x*′) (*f x*) < *e*
using ‹*?lhs*›[*unfolded uniformly-continuous-on-def*, *THEN spec*[**where** *x*=*e*]] **by** *auto*

obtain N **where** $N{:}\forall\, n{\geq}N.\ norm\ (x\ n\ -\ y\ n\ -\ 0) < d$ **using** $xy[unfolded\ Lim\text{-}sequentially\ dist\text{-}def]$ **and** $\langle d{>}0\rangle$ **by** *auto*
{ fix n **assume** $n{\geq}N$
hence $norm\ (f\ (x\ n) - f\ (y\ n) - 0) < e$
using $N[THEN\ spec[\textbf{where}\ x{=}n]]$ **using** $d[THEN\ bspec[\textbf{where}\ x{=}x\ n], THEN\ bspec[\textbf{where}\ x{=}y\ n]]$ **using** x **and** y
unfolding *dist-sym* **and** *dist-def* **by** *simp* **}**
hence $\exists\, N.\ \forall\, n{\geq}N.\ norm\ (f\ (x\ n) - f\ (y\ n) - 0) < e$ **by** *auto* **}**
hence $((\lambda n.\ f(x\ n) - f(y\ n)) \longrightarrow 0)\ sequentially$ **unfolding** *Lim-sequentially* **and** *dist-def* **by** *auto* **}**
thus *?rhs* **by** *auto*
next
assume *?rhs*
{ assume $\neg$ *?lhs*
then obtain e **where** $e{>}0\ \forall\, d{>}0.\ \exists\, x{\in}s.\ \exists\, x'{\in}s.\ dist\ x'\ x < d \wedge \neg\ dist\ (f\ x')\ (f\ x) < e$ **unfolding** *uniformly-continuous-on-def* **by** *auto*
then obtain fa **where** $fa{:}\forall\, x.\ \ 0 < x \longrightarrow fst\ (fa\ x) \in s \wedge snd\ (fa\ x) \in s \wedge dist\ (fst\ (fa\ x))\ (snd\ (fa\ x)) < x \wedge \neg\ dist\ (f\ (fst\ (fa\ x)))\ (f\ (snd\ (fa\ x))) < e$
using $choice[of\ \lambda d\ x.\ d{>}0 \longrightarrow fst\ x \in s \wedge snd\ x \in s \wedge dist\ (snd\ x)\ (fst\ x) < d \wedge \neg\ dist\ (f\ (snd\ x))\ (f\ (fst\ x)) < e]$ **unfolding** *Bex-def*
by (*auto simp add*: *dist-sym*)
def $x \equiv \lambda n{::}nat.\ fst\ (fa\ (inverse\ (real\ n\ +\ 1)))$
def $y \equiv \lambda n{::}nat.\ snd\ (fa\ (inverse\ (real\ n\ +\ 1)))$
have $xyn{:}\forall\, n.\ x\ n \in s \wedge y\ n \in s$ **and** $xy0{:}\forall\, n.\ dist\ (x\ n)\ (y\ n) < inverse\ (real\ n\ +\ 1)$ **and** $fxy{:}\forall\, n.\ \neg\ dist\ (f\ (x\ n))\ (f\ (y\ n)) < e$
unfolding *x-def* **and** *y-def* **using** *fa* **by** *auto*
have $*{:}\bigwedge x\ y.\ dist\ (x\ -\ y)\ 0 = dist\ x\ y$ **unfolding** *dist-def* **by** *auto*
{ fix $e{::}real$ **assume** $e{>}0$
then obtain $N{::}nat$ **where** $N \neq 0$ **and** $N{:}0 < inverse\ (real\ N) \wedge inverse\ (real\ N) < e$ **unfolding** $real\text{-}arch\text{-}inv[of\ e]$ **by** *auto*
{ fix $n{::}nat$ **assume** $n{\geq}N$
hence $inverse\ (real\ n\ +\ 1) < inverse\ (real\ N)$ **using** *real-of-nat-ge-zero* **and** $\langle N{\neq}0\rangle$ **by** *auto*
also have $\ldots < e$ **using** N **by** *auto*
finally have $inverse\ (real\ n\ +\ 1) < e$ **by** *auto*
hence $dist\ (x\ n\ -\ y\ n)\ 0 < e$ **unfolding** $*$ **using** $xy0[THEN\ spec[\textbf{where}\ x{=}n]]$ **by** *auto* **}**
hence $\exists\, N.\ \forall\, n{\geq}N.\ dist\ (x\ n\ -\ y\ n)\ 0 < e$ **by** *auto* **}**
hence $\forall\, e{>}0.\ \exists\, N.\ \forall\, n{\geq}N.\ dist\ (f\ (x\ n) - f\ (y\ n))\ 0 < e$ **using** $\langle ?rhs\rangle[THEN\ spec[\textbf{where}\ x{=}x], THEN\ spec[\textbf{where}\ x{=}y]]$ **and** xyn **unfolding** *Lim-sequentially* **by** *auto*
hence *False* **unfolding** $*$ **using** fxy **and** $\langle e{>}0\rangle$ **by** *auto* **}**
thus *?lhs* **unfolding** *uniformly-continuous-on-def* **by** *blast*
qed

The usual transformation theorems.

lemma *continuous-transform-within*:
assumes $0 < d\ x \in s\ \forall\, x' \in s.\ dist\ x'\ x < d \longrightarrow f\ x' = g\ x'$
$continuous\ (at\ x\ within\ s)\ f$

shows *continuous* (*at x within s*) *g*
proof−
{ fix *e*::*real* **assume** *e*>*0*
then obtain *d'* **where** *d'*:*d'*>*0* ∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < *d'* ⟶ *dist* (*f xa*) (*f x*) < *e* **using** *assms*(*4*) **unfolding** *continuous-within Lim-within* **by** *auto*
{ fix *x'* **assume** *x'*∈*s* *0* < *dist x' x dist x' x* < (*min d d'*)
hence *dist* (*f x'*) (*g x*) < *e* **using** *assms*(*2*,*3*) **apply**(*erule-tac x*=*x* **in** *ballE*) **unfolding** *dist-refl* **using** *d'* **by** *auto* **}**
hence ∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < (*min d d'*) ⟶ *dist* (*f xa*) (*g x*) < *e* **by** *blast*
hence ∃ *d*>*0*. ∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *dist* (*f xa*) (*g x*) < *e* **using** ‹*d*>*0*› ‹*d'*>*0*› **by**(*rule-tac x*=*min d d'* **in** *exI*)*auto* **}**
hence (*f* −−−> *g x*) (*at x within s*) **unfolding** *Lim-within* **using** *assms*(*1*) **by** *auto*
thus *?thesis* **unfolding** *continuous-within* **using** *Lim-transform-within*[*of d s x f g g x*] **using** *assms* **by** *blast*
qed

lemma *continuous-transform-at*:
assumes *0* < *d* ∀ *x'*. *dist x' x* < *d* −−> *f x'* = *g x'*
continuous (*at x*) *f*
shows *continuous* (*at x*) *g*
proof−
{ fix *e*::*real* **assume** *e*>*0*
then obtain *d'* **where** *d'*:*d'*>*0* ∀ *xa*. *0* < *dist xa x* ∧ *dist xa x* < *d'* ⟶ *dist* (*f xa*) (*f x*) < *e* **using** *assms*(*3*) **unfolding** *continuous-at Lim-at* **by** *auto*
{ fix *x'* **assume** *0* < *dist x' x dist x' x* < (*min d d'*)
hence *dist* (*f x'*) (*g x*) < *e* **using** *assms*(*2*) **apply**(*erule-tac x*=*x* **in** *allE*) **unfolding** *dist-refl* **using** *d'* **by** *auto*
}
hence ∀ *xa*. *0* < *dist xa x* ∧ *dist xa x* < (*min d d'*) ⟶ *dist* (*f xa*) (*g x*) < *e* **by** *blast*
hence ∃ *d*>*0*. ∀ *xa*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *dist* (*f xa*) (*g x*) < *e* **using** ‹*d*>*0*› ‹*d'*>*0*› **by**(*rule-tac x*=*min d d'* **in** *exI*)*auto*
}
hence (*f* −−−> *g x*) (*at x*) **unfolding** *Lim-at* **using** *assms*(*1*) **by** *auto*
thus *?thesis* **unfolding** *continuous-at* **using** *Lim-transform-at*[*of d x f g g x*] **using** *assms* **by** *blast*
qed

Combination results for pointwise continuity.

lemma *continuous-const*: *continuous net* (λ*x*::'*a*::*zero-neq-one*. *c*)
by(*auto simp add*: *continuous-def Lim-const*)

lemma *continuous-cmul*:
continuous net f ==> *continuous net* (λ*x*. *c* ∗*s f x*)
by(*auto simp add*: *continuous-def Lim-cmul*)

lemma *continuous-neg*:
continuous net f ==> continuous net $(\lambda x.\ -(f\ x))$
by(*auto simp add*: *continuous-def Lim-neg*)

lemma *continuous-add*:
continuous net f $\Longrightarrow$ *continuous net g*
$\quad$ *==> continuous net* $(\lambda x.\ f\ x + g\ x)$
by(*auto simp add*: *continuous-def Lim-add*)

lemma *continuous-sub*:
continuous net f $\Longrightarrow$ *continuous net g*
$\quad$ *==> continuous net* $(\lambda x.\ f(x) - g(x))$
by(*auto simp add*: *continuous-def Lim-sub*)

Same thing for setwise continuity.

lemma *continuous-on-const*:
continuous-on s $(\lambda x.\ c)$
unfolding *continuous-on-eq-continuous-within* **using** *continuous-const* **by** *blast*

lemma *continuous-on-cmul*:
continuous-on s f ==> continuous-on s $(\lambda x.\ c *s (f\ x))$
unfolding *continuous-on-eq-continuous-within* **using** *continuous-cmul* **by** *blast*

lemma *continuous-on-neg*:
continuous-on s f ==> continuous-on s $(\lambda x.\ -(f\ x))$
unfolding *continuous-on-eq-continuous-within* **using** *continuous-neg* **by** *blast*

lemma *continuous-on-add*:
continuous-on s f $\Longrightarrow$ *continuous-on s g*
$\quad$ *==> continuous-on s* $(\lambda x.\ f\ x + g\ x)$
unfolding *continuous-on-eq-continuous-within* **using** *continuous-add* **by** *blast*

lemma *continuous-on-sub*:
continuous-on s f $\Longrightarrow$ *continuous-on s g*
$\quad$ *==> continuous-on s* $(\lambda x.\ f(x) - g(x))$
unfolding *continuous-on-eq-continuous-within* **using** *continuous-sub* **by** *blast*

Same thing for uniform continuity, using sequential formulations.

lemma *uniformly-continuous-on-const*:
uniformly-continuous-on s $(\lambda x.\ c)$
unfolding *uniformly-continuous-on-sequentially* **using** *Lim-const*[*of 0*] **by** *auto*

lemma *uniformly-continuous-on-cmul*:
assumes *uniformly-continuous-on s f*
shows *uniformly-continuous-on s* $(\lambda x.\ c *s f(x))$
proof−
{ fix *x y* **assume** $((\lambda n.\ f\ (x\ n) - f\ (y\ n)) \longrightarrow 0)$ *sequentially*
hence $((\lambda n.\ c *s f\ (x\ n) - c *s f\ (y\ n)) \longrightarrow 0)$ *sequentially*
using *Lim-cmul*[*of* $(\lambda n.\ f\ (x\ n) - f\ (y\ n))$ *0 sequentially c*]

unfolding *vector-smult-rzero vector-ssub-ldistrib*[*of c*] **by** *auto*
}
thus *?thesis* **using** *assms* **unfolding** *uniformly-continuous-on-sequentially* **by** *auto*
qed

lemma *uniformly-continuous-on-neg*:
uniformly-continuous-on s f
==> *uniformly-continuous-on s* (λ*x*. −(*f x*))
using *uniformly-continuous-on-cmul*[*of s f* −*1*] **unfolding** *pth-3* **by** *auto*

lemma *uniformly-continuous-on-add*:
assumes *uniformly-continuous-on s f uniformly-continuous-on s g*
shows *uniformly-continuous-on s* (λ*x*. *f*(*x*) + *g*(*x*) ::*real*^'*n*::*finite*)
proof−
have ∗:⋀*fx fy gx gy*::*real*^'*n*. *fx* − *fy* + (*gx* − *gy*) = *fx* + *gx* − (*fy* + *gy*) **by** *auto*
{ **fix** *x y* **assume** ((λ*n*. *f* (*x n*) − *f* (*y n*)) −−−> *0*) *sequentially*
((λ*n*. *g* (*x n*) − *g* (*y n*)) −−−> *0*) *sequentially*
hence ((λ*xa*. *f* (*x xa*) − *f* (*y xa*) + (*g* (*x xa*) − *g* (*y xa*))) −−−> *0* + *0*) *sequentially*
using *Lim-add*[*of* λ *n*. *f* (*x n*) − *f* (*y n*) *0* *sequentially* λ *n*. *g* (*x n*) − *g* (*y n*) *0*] **by** *auto*
hence ((λ*n*. *f* (*x n*) + *g* (*x n*) − (*f* (*y n*) + *g* (*y n*))) −−−> *0*) *sequentially* **unfolding** *Lim-sequentially* **and** ∗ **by** *auto* }
thus *?thesis* **using** *assms* **unfolding** *uniformly-continuous-on-sequentially* **by** *auto*
qed

lemma *uniformly-continuous-on-sub*:
uniformly-continuous-on s f ⟹ *uniformly-continuous-on s g*
==> *uniformly-continuous-on s* (λ*x*. *f x* − *g x*)
unfolding *ab-diff-minus*
using *uniformly-continuous-on-add*[*of s f* λ*x*. − *g x*]
using *uniformly-continuous-on-neg*[*of s g*] **by** *auto*

Identity function is continuous in every sense.

lemma *continuous-within-id*:
continuous (*at a within s*) (λ*x*. *x*)
unfolding *continuous-within Lim-within* **by** *auto*

lemma *continuous-at-id*:
continuous (*at a*) (λ*x*. *x*)
unfolding *continuous-at Lim-at* **by** *auto*

lemma *continuous-on-id*:
continuous-on s (λ*x*. *x*)
unfolding *continuous-on Lim-within* **by** *auto*

lemma *uniformly-continuous-on-id*:
uniformly-continuous-on s (λ*x*. *x*)
unfolding *uniformly-continuous-on-def* **by** *auto*

Continuity of all kinds is preserved under composition.

lemma *continuous-within-compose*:
assumes *continuous* (*at x within s*) *f* *continuous* (*at* (*f x*) *within f* ' *s*) *g*
shows *continuous* (*at x within s*) (*g o f*)
proof−
{ fix *e*::*real* **assume** *e*>*0*
with *assms*(*2*)[*unfolded continuous-within Lim-within*] **obtain** *d* **where** *d*>*0* **and** *d*:∀ *xa*∈*f* ' *s*. *0* < *dist xa* (*f x*) ∧ *dist xa* (*f x*) < *d* ⟶ *dist* (*g xa*) (*g* (*f x*)) < *e* **by** *auto*
from *assms*(*1*)[*unfolded continuous-within Lim-within*] **obtain** *d'* **where** *d'*>*0* **and** *d'*:∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < *d'* ⟶ *dist* (*f xa*) (*f x*) < *d* **using** ‹*d*>*0*› **by** *auto*
{ fix *y* **assume** *as*:*y*∈*s* *0* < *dist y x* *dist y x* < *d'*
hence *dist* (*f y*) (*f x*) < *d* **using** *d'*[*THEN bspec*[**where** *x*=*y*]] **by** (*auto simp add*:*dist-sym*)
hence *dist* (*g* (*f y*)) (*g* (*f x*)) < *e* **using** *as*(*1*) *d*[*THEN bspec*[**where** *x*=*f y*]] **unfolding** *dist-nz*[*THEN sym*] **using** ‹*e*>*0*› **by** (*auto simp add*: *dist-refl*) **}**
hence ∃ *d*>*0*. ∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *dist* (*g* (*f xa*)) (*g* (*f x*)) < *e* **using** ‹*d'*>*0*› **by** *auto* **}**
thus *?thesis* **unfolding** *continuous-within Lim-within* **by** *auto*
qed

lemma *continuous-at-compose*:
assumes *continuous* (*at x*) *f* *continuous* (*at* (*f x*)) *g*
shows *continuous* (*at x*) (*g o f*)
proof−
have *continuous* (*at* (*f x*) *within range f*) *g* **using** *assms*(*2*) **using** *continuous-within-subset*[*of f x UNIV g range f*, *unfolded within-UNIV*] **by** *auto*
thus *?thesis* **using** *assms*(*1*) **using** *continuous-within-compose*[*of x UNIV f g*, *unfolded within-UNIV*] **by** *auto*
qed

lemma *continuous-on-compose*:
continuous-on s f ⟹ *continuous-on* (*f* ' *s*) *g* ⟹ *continuous-on s* (*g o f*)
unfolding *continuous-on-eq-continuous-within* **using** *continuous-within-compose*[*of - s f g*] **by** *auto*

lemma *uniformly-continuous-on-compose*:
assumes *uniformly-continuous-on s f* *uniformly-continuous-on* (*f* ' *s*) *g*
shows *uniformly-continuous-on s* (*g o f*)
proof−
{ fix *e*::*real* **assume** *e*>*0*
then obtain *d* **where** *d*>*0* **and** *d*:∀ *x*∈*f* ' *s*. ∀ *x'*∈*f* ' *s*. *dist x' x* < *d* ⟶ *dist* (*g x'*) (*g x*) < *e* **using** *assms*(*2*) **unfolding** *uniformly-continuous-on-def* **by** *auto*
obtain *d'* **where** *d'*>*0* ∀ *x*∈*s*. ∀ *x'*∈*s*. *dist x' x* < *d'* ⟶ *dist* (*f x'*) (*f x*) < *d*

using ‹*d*>*0*› **using** *assms*(*1*) **unfolding** *uniformly-continuous-on-def* **by** *auto*
hence ∃ *d*>*0*. ∀ *x*∈*s*. ∀ *x'*∈*s*. *dist x' x* < *d* ⟶ *dist* ((*g* ∘ *f*) *x'*) ((*g* ∘ *f*) *x*) < *e* **using** ‹*d*>*0*› **using** *d* **by** *auto* **}**
thus *?thesis* **using** *assms* **unfolding** *uniformly-continuous-on-def* **by** *auto*
qed

Continuity in terms of open preimages.

lemma *continuous-at-open*:
continuous (*at x*) *f* ⟷ (∀ *t*. *open t* ∧ *f x* ∈ *t* −−> (∃ *s*. *open s* ∧ *x* ∈ *s* ∧ (∀ *x'* ∈ *s*. (*f x'*) ∈ *t*))) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ fix *t* **assume** *as*: *open t f x* ∈ *t*
then obtain *e* **where** *e*>*0* **and** *e*:*ball* (*f x*) *e* ⊆ *t* **unfolding** *open-contains-ball* **by** *auto*

obtain *d* **where** *d*>*0* **and** *d*:∀ *y*. *0* < *dist y x* ∧ *dist y x* < *d* ⟶ *dist* (*f y*) (*f x*) < *e* **using** ‹*e*>*0*› **using** ‹*?lhs*›[*unfolded continuous-at Lim-at open-def*] **by** *auto*

have *open* (*ball x d*) **using** *open-ball* **by** *auto*
moreover have *x* ∈ *ball x d* **unfolding** *centre-in-ball* **using** ‹*d*>*0*› **by** *simp*
moreover
{ fix *x'* **assume** *x'*∈*ball x d* **hence** *f x'* ∈ *t*
using *e*[*unfolded subset-eq Ball-def mem-ball*, *THEN spec*[**where** *x*=*f x'*]] *d*[*THEN spec*[**where** *x*=*x'*]]
unfolding *mem-ball* **apply** (*auto simp add*: *dist-sym*)
unfolding *dist-nz*[*THEN sym*] **using** *as*(*2*) **by** *auto* **}**
hence ∀ *x'*∈*ball x d*. *f x'* ∈ *t* **by** *auto*
ultimately have ∃ *s*. *open s* ∧ *x* ∈ *s* ∧ (∀ *x'*∈*s*. *f x'* ∈ *t*)
apply(*rule-tac x*=*ball x d* **in** *exI*) **by** *simp* **}**
thus *?rhs* **by** *auto*
next
assume *?rhs*
{ fix *e*::*real* **assume** *e*>*0*
then obtain *s* **where** *s*: *open s* *x* ∈ *s* ∀ *x'*∈*s*. *f x'* ∈ *ball* (*f x*) *e* **using** ‹*?rhs*›[*unfolded continuous-at Lim-at*, *THEN spec*[**where** *x*=*ball* (*f x*) *e*]]
unfolding *centre-in-ball*[*of f x e*, *THEN sym*] **by** *auto*
then obtain *d* **where** *d*>*0* **and** *d*:*ball x d* ⊆ *s* **unfolding** *open-contains-ball* **by** *auto*
{ fix *y* **assume** *0* < *dist y x* ∧ *dist y x* < *d*
hence *dist* (*f y*) (*f x*) < *e* **using** *d*[*unfolded subset-eq Ball-def mem-ball*, *THEN spec*[**where** *x*=*y*]]
using *s*(*3*)[*THEN bspec*[**where** *x*=*y*], *unfolded mem-ball*] **by** (*auto simp add*: *dist-sym*) **}**
hence ∃ *d*>*0*. ∀ *xa*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *dist* (*f xa*) (*f x*) < *e* **using** ‹*d*>*0*› **by** *auto* **}**
thus *?lhs* **unfolding** *continuous-at Lim-at* **by** *auto*
qed

lemma *continuous-on-open*:
continuous-on s f ⟷
(∀ *t*. *openin* (*subtopology euclidean* (*f* ‘ *s*)) *t*
−−> *openin* (*subtopology euclidean s*) {*x* ∈ *s*. *f x* ∈ *t*}) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ fix *t* **assume** *as*:*openin* (*subtopology euclidean* (*f* ‘ *s*)) *t*
have {*x* ∈ *s*. *f x* ∈ *t*} ⊆ *s* **using** *as*[*unfolded openin-euclidean-subtopology-iff*]
by *auto*
moreover
{ fix *x* **assume** *as′*:*x*∈{*x* ∈ *s*. *f x* ∈ *t*}
then obtain *e* **where** *e*: *e*>*0* ∀ *x′*∈*f* ‘ *s*. *dist x′* (*f x*) < *e* ⟶ *x′* ∈ *t* **using**
as[*unfolded openin-euclidean-subtopology-iff*, *THEN conjunct2*, *THEN bspec*[**where**
x=*f x*]] **by** *auto*
from *this*(*1*) **obtain** *d* **where** *d*: *d*>*0* ∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < *d*
⟶ *dist* (*f xa*) (*f x*) < *e* **using** ‹*?lhs*›[*unfolded continuous-on Lim-within*, *THEN*
bspec[**where** *x*=*x*]] **using** *as′* **by** *auto*
have ∃ *e*>*0*. ∀ *x′*∈*s*. *dist x′ x* < *e* ⟶ *x′* ∈ {*x* ∈ *s*. *f x* ∈ *t*} **using** *d e*
unfolding *dist-nz*[*THEN sym*] **by** (*rule-tac x*=*d* **in** *exI*, *auto simp add*: *dist-refl*)
}
ultimately have *openin* (*subtopology euclidean s*) {*x* ∈ *s*. *f x* ∈ *t*} **unfolding**
openin-euclidean-subtopology-iff **by** *auto* **}**
thus *?rhs* **unfolding** *continuous-on Lim-within* **using** *openin* **by** *auto*
next
assume *?rhs*
{ fix *e*::*real* **and** *x* **assume** *x*∈*s e*>*0*
{ fix *xa x′* **assume** *dist* (*f xa*) (*f x*) < *e xa* ∈ *s x′* ∈ *s dist* (*f xa*) (*f x′*) < *e* −
dist (*f xa*) (*f x*)
hence *dist* (*f x′*) (*f x*) < *e* **using** *dist-triangle*[*of f x′ f x f xa*]
by (*auto simp add*: *dist-sym*) **}**
hence *ball* (*f x*) *e* ∩ *f* ‘ *s* ⊆ *f* ‘ *s* ∧ (∀ *xa*∈*ball* (*f x*) *e* ∩ *f* ‘ *s*. ∃ *ea*>*0*. ∀ *x′*∈*f* ‘
s. *dist x′ xa* < *ea* ⟶ *x′* ∈ *ball* (*f x*) *e* ∩ *f* ‘ *s*) **apply** *auto*
apply(*rule-tac x*=*e* − *dist* (*f xa*) (*f x*) **in** *exI*) **using** ‹*e*>*0*› **by** (*auto simp*
add: *dist-sym*)
hence ∀ *xa*∈{*xa* ∈ *s*. *f xa* ∈ *ball* (*f x*) *e* ∩ *f* ‘ *s*}. ∃ *ea*>*0*. ∀ *x′*∈*s*. *dist x′ xa* <
ea ⟶ *x′* ∈ {*xa* ∈ *s*. *f xa* ∈ *ball* (*f x*) *e* ∩ *f* ‘ *s*}
using ‹*?rhs*›[*unfolded openin-euclidean-subtopology-iff*, *THEN spec*[**where**
x=*ball* (*f x*) *e* ∩ *f* ‘ *s*]] **by** *auto*
hence ∃ *d*>*0*. ∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *dist* (*f xa*) (*f x*) < *e*
apply(*erule-tac x*=*x* **in** *ballE*) **apply** *auto* **unfolding** *dist-refl* **using** ‹*e*>*0*› ‹*x*∈*s*›
by (*auto simp add*: *dist-sym*) **}**
thus *?lhs* **unfolding** *continuous-on Lim-within* **by** *auto*
qed

lemma *continuous-on-closed*:

continuous-on s f $\longleftrightarrow$ $(\forall t.$ *closedin* (*subtopology euclidean* (*f ' s*)) *t* −−> *closedin* (*subtopology euclidean s*) $\{x \in s.\ f\ x \in t\}$) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
{ fix *t*
have ∗:$s - \{x \in s.\ f\ x \in f\ `\ s - t\} = \{x \in s.\ f\ x \in t\}$ **by** *auto*
have ∗∗:$f\ `\ s - (f\ `\ s - (f\ `\ s - t)) = f\ `\ s - t$ **by** *auto*
assume *as*:*closedin* (*subtopology euclidean* (*f ' s*)) *t*
hence *closedin* (*subtopology euclidean* (*f ' s*)) (*f ' s* − (*f ' s* − *t*)) **unfolding** *closedin-def topspace-euclidean-subtopology* **unfolding** ∗∗ **by** *auto*
hence *closedin* (*subtopology euclidean s*) $\{x \in s.\ f\ x \in t\}$ **using** ‹*?lhs*›[*unfolded continuous-on-open*, *THEN spec*[**where** *x*=(*f ' s*) − *t*]]
unfolding *openin-closedin-eq topspace-euclidean-subtopology* **unfolding** ∗ **by** *auto* **}**
thus *?rhs* **by** *auto*
next
assume *?rhs*
{ fix *t*
have ∗:$s - \{x \in s.\ f\ x \in f\ `\ s - t\} = \{x \in s.\ f\ x \in t\}$ **by** *auto*
assume *as*:*openin* (*subtopology euclidean* (*f ' s*)) *t*
hence *openin* (*subtopology euclidean s*) $\{x \in s.\ f\ x \in t\}$ **using** ‹*?rhs*›[*THEN spec*[**where** *x*=(*f ' s*) − *t*]]
unfolding *openin-closedin-eq topspace-euclidean-subtopology* ∗[*THEN sym*] *closedin-subtopology* **by** *auto* **}**
thus *?lhs* **unfolding** *continuous-on-open* **by** *auto*
qed

Half-global and completely global cases.

lemma *continuous-open-in-preimage*:
assumes *continuous-on s f open t*
shows *openin* (*subtopology euclidean s*) $\{x \in s.\ f\ x \in t\}$
proof−
have ∗:$\forall x.\ x \in s \wedge f\ x \in t \longleftrightarrow x \in s \wedge f\ x \in (t \cap f\ `\ s)$ **by** *auto*
have *openin* (*subtopology euclidean* (*f ' s*)) ($t \cap f\ `\ s$)
using *openin-open-Int*[*of t f ' s*, *OF assms*(*2*)] **unfolding** *openin-open* **by** *auto*
thus *?thesis* **using** *assms*(*1*)[*unfolded continuous-on-open*, *THEN spec*[**where** *x*=$t \cap f\ `\ s$]] **using** ∗ **by** *auto*
qed

lemma *continuous-closed-in-preimage*:
assumes *continuous-on s f closed t*
shows *closedin* (*subtopology euclidean s*) $\{x \in s.\ f\ x \in t\}$
proof−
have ∗:$\forall x.\ x \in s \wedge f\ x \in t \longleftrightarrow x \in s \wedge f\ x \in (t \cap f\ `\ s)$ **by** *auto*
have *closedin* (*subtopology euclidean* (*f ' s*)) ($t \cap f\ `\ s$)
using *closedin-closed-Int*[*of t f ' s*, *OF assms*(*2*)] **unfolding** *Int-commute* **by** *auto*
thus *?thesis*
using *assms*(*1*)[*unfolded continuous-on-closed*, *THEN spec*[**where** *x*=$t \cap f\ `$

s]] **using** ∗ **by** *auto*
qed

lemma *continuous-open-preimage*:
assumes *continuous-on s f open s open t*
shows *open* {*x* ∈ *s*. *f x* ∈ *t*}
proof−
obtain *T* **where** *T*: *open T* {*x* ∈ *s*. *f x* ∈ *t*} = *s* ∩ *T*
using *continuous-open-in-preimage*[*OF assms*(*1*,*3*)] **unfolding** *openin-open* **by** *auto*
thus *?thesis* **using** *open-inter*[*of s T*, *OF assms*(*2*)] **by** *auto*
qed

lemma *continuous-closed-preimage*:
assumes *continuous-on s f closed s closed t*
shows *closed* {*x* ∈ *s*. *f x* ∈ *t*}
proof−
obtain *T* **where** *T*: *closed T* {*x* ∈ *s*. *f x* ∈ *t*} = *s* ∩ *T*
using *continuous-closed-in-preimage*[*OF assms*(*1*,*3*)] **unfolding** *closedin-closed* **by** *auto*
thus *?thesis* **using** *closed-Int*[*of s T*, *OF assms*(*2*)] **by** *auto*
qed

lemma *continuous-open-preimage-univ*:
∀ *x*. *continuous* (*at x*) *f* ⟹ *open s* ⟹ *open* {*x*. *f x* ∈ *s*}
using *continuous-open-preimage*[*of UNIV f s*] *open-UNIV continuous-at-imp-continuous-on* **by** *auto*

lemma *continuous-closed-preimage-univ*:
(∀ *x*. *continuous* (*at x*) *f*) ⟹ *closed s* ==> *closed* {*x*. *f x* ∈ *s*}
using *continuous-closed-preimage*[*of UNIV f s*] *closed-UNIV continuous-at-imp-continuous-on* **by** *auto*

Equality of continuous functions on closure and related results.

lemma *continuous-closed-in-preimage-constant*:
continuous-on s f ==> *closedin* (*subtopology euclidean s*) {*x* ∈ *s*. *f x* = *a*}
using *continuous-closed-in-preimage*[*of s f* {*a*}] *closed-sing* **by** *auto*

lemma *continuous-closed-preimage-constant*:
continuous-on s f ⟹ *closed s* ==> *closed* {*x* ∈ *s*. *f x* = *a*}
using *continuous-closed-preimage*[*of s f* {*a*}] *closed-sing* **by** *auto*

lemma *continuous-constant-on-closure*:
assumes *continuous-on* (*closure s*) *f*
∀ *x* ∈ *s*. *f x* = *a*
shows ∀ *x* ∈ (*closure s*). *f x* = *a*
using *continuous-closed-preimage-constant*[*of closure s f a*]
assms closure-minimal[*of s* {*x* ∈ *closure s*. *f x* = *a*}] *closure-subset* **unfolding** *subset-eq* **by** *auto*

lemma *image-closure-subset*:
assumes *continuous-on* (*closure s*) *f* *closed t* (*f ' s*) ⊆ *t*
shows *f ' (closure s)* ⊆ *t*
proof−
have *s* ⊆ {*x* ∈ *closure s*. *f x* ∈ *t*} **using** *assms*(*3*) *closure-subset* **by** *auto*
moreover have *closed* {*x* ∈ *closure s*. *f x* ∈ *t*}
using *continuous-closed-preimage*[*OF assms*(*1*)] **and** *assms*(*2*) **by** *auto*
ultimately have *closure s* = {*x* ∈ *closure s* . *f x* ∈ *t*}
using *closure-minimal*[*of s* {*x* ∈ *closure s*. *f x* ∈ *t*}] **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *continuous-on-closure-norm-le*:
assumes *continuous-on* (*closure s*) *f* ∀ *y* ∈ *s*. *norm*(*f y*) ≤ *b* *x* ∈ (*closure s*)
shows *norm*(*f x*) ≤ *b*
proof−
have ∗:*f ' s* ⊆ *cball 0 b* **using** *assms*(*2*)[*unfolded mem-cball-0*[*THEN sym*]] **by** *auto*
show *?thesis*
using *image-closure-subset*[*OF assms*(*1*) *closed-cball*[*of 0 b*] ∗] *assms*(*3*)
unfolding *subset-eq* **apply**(*erule-tac x*=*f x* **in** *ballE*) **by** (*auto simp add*: *dist-def*)
qed

Making a continuous function avoid some value in a neighbourhood.

lemma *continuous-within-avoid*:
assumes *continuous* (*at x within s*) *f* *x* ∈ *s* *f x* ≠ *a*
shows ∃ *e*>*0*. ∀ *y* ∈ *s*. *dist x y* < *e* −−> *f y* ≠ *a*
proof−
obtain *d* **where** *d*>*0* **and** *d*:∀ *xa*∈*s*. *0* < *dist xa x* ∧ *dist xa x* < *d* ⟶ *dist* (*f xa*) (*f x*) < *dist* (*f x*) *a*
using *assms*(*1*)[*unfolded continuous-within Lim-within*, *THEN spec*[**where** *x*=*dist* (*f x*) *a*]] *assms*(*3*)[*unfolded dist-nz*] **by** *auto*
{ **fix** *y* **assume** *y*∈*s* *dist x y* < *d*
hence *f y* ≠ *a* **using** *d*[*THEN bspec*[**where** *x*=*y*]] *assms*(*3*)[*unfolded dist-nz*]
apply *auto* **unfolding** *dist-nz*[*THEN sym*] **by** (*auto simp add*: *dist-sym*) }
thus *?thesis* **using** ‹*d*>*0*› **by** *auto*
qed

lemma *continuous-at-avoid*:
assumes *continuous* (*at x*) *f* *f x* ≠ *a*
shows ∃ *e*>*0*. ∀ *y*. *dist x y* < *e* ⟶ *f y* ≠ *a*
using *assms* **using** *continuous-within-avoid*[*of x UNIV f a*, *unfolded within-UNIV*]
by *auto*

lemma *continuous-on-avoid*:
assumes *continuous-on s f* *x* ∈ *s* *f x* ≠ *a*
shows ∃ *e*>*0*. ∀ *y* ∈ *s*. *dist x y* < *e* ⟶ *f y* ≠ *a*

using *assms(1)*[*unfolded continuous-on-eq-continuous-within*, *THEN bspec*[**where** *x=x*], *OF assms(2)*] *continuous-within-avoid*[*of x s f a*] *assms(2,3)* **by** *auto*

lemma *continuous-on-open-avoid*:
assumes *continuous-on s f* *open s* $x \in s$ $f\,x \neq a$
shows $\exists\, e{>}0.\ \forall\, y.\ dist\ x\ y < e \longrightarrow f\ y \neq a$
using *assms(1)*[*unfolded continuous-on-eq-continuous-at*[*OF assms(2)*], *THEN bspec*[**where** *x=x*], *OF assms(3)*] *continuous-at-avoid*[*of x f a*] *assms(3,4)* **by** *auto*

Proving a function is constant by proving open-ness of level set.

lemma *continuous-levelset-open-in-cases*:
connected s $\Longrightarrow$ *continuous-on s f* $\Longrightarrow$
openin (*subtopology euclidean s*) $\{x \in s.\ f\,x = a\}$
$==>$ $(\forall\, x \in s.\ f\,x \neq a) \vee (\forall\, x \in s.\ f\,x = a)$
unfolding *connected-clopen* **using** *continuous-closed-in-preimage-constant* **by** *auto*

lemma *continuous-levelset-open-in*:
connected s $\Longrightarrow$ *continuous-on s f* $\Longrightarrow$
openin (*subtopology euclidean s*) $\{x \in s.\ f\,x = a\} \Longrightarrow$
$(\exists\, x \in s.\ f\,x = a)$ $==>$ $(\forall\, x \in s.\ f\,x = a)$
using *continuous-levelset-open-in-cases*[*of s f*]
by *meson*

lemma *continuous-levelset-open*:
assumes *connected s* *continuous-on s f* *open* $\{x \in s.\ f\,x = a\}$ $\exists\, x \in s.\ f\,x = a$
shows $\forall\, x \in s.\ f\,x = a$
using *continuous-levelset-open-in*[*OF assms(1,2)*, *of a*, *unfolded openin-open*] **using** *assms (3,4)* **by** *auto*

Some arithmetical combinations (more to prove).

lemma *open-scaling*[*intro*]:
assumes $c \neq 0$ *open s*
shows *open*$((\lambda x.\ c *s x)\ `\ s)$
proof−
{ fix *x* **assume** $x \in s$
then obtain *e* **where** *e>0* **and** *e*:$\forall\, x'.\ dist\ x'\ x < e \longrightarrow x' \in s$ **using** *assms(2)*[*unfolded open-def*, *THEN bspec*[**where** *x=x*]] **by** *auto*
have $e * abs\ c > 0$ **using** *assms(1)*[*unfolded zero-less-abs-iff*[*THEN sym*]] **using** *real-mult-order*[*OF* ‹*e>0*›] **by** *auto*
moreover
{ fix *y* **assume** $dist\ y\ (c *s x) < e * |c|$
hence *norm* $((1\ /\ c) *s\ y - x) < e$ **unfolding** *dist-def*
using *norm-mul*[*of c* $(1\ /\ c) *s\ y - x$, *unfolded vector-ssub-ldistrib*, *unfolded vector-smult-assoc*] *assms(1)*
assms(1)[*unfolded zero-less-abs-iff*[*THEN sym*]] **by** (*simp del*:*zero-less-abs-iff*)
hence $y \in op *s\ c\ `\ s$ **using** *rev-image-eqI*[*of* $(1\ /\ c) *s\ y\ s\ y$ *op* $*s$ *c*] *e*[*THEN spec*[**where** *x*=$(1\ /\ c) *s\ y$]] *assms(1)* **unfolding** *dist-def vector-smult-assoc* **by** *auto* **}**

ultimately have $\exists e{>}0.\ \forall x'.\ dist\ x'\ (c\ *s\ x) < e \longrightarrow x' \in op\ *s\ c\ `\ s$ **apply**(*rule-tac* $x{=}e * abs\ c$ **in** *exI*) **by** *auto* **}**
thus *?thesis* **unfolding** *open-def* **by** *auto*
qed

lemma *open-negations*:
$open\ s \Longrightarrow open\ ((\lambda\ x.\ -x)\ `\ s)$ **unfolding** *pth-3* **by** *auto*

lemma *open-translation*:
assumes *open s* **shows** $open((\lambda x.\ a + x)\ `\ s)$
proof−
{ fix x **have** *continuous* $(at\ x)\ (\lambda x.\ x - a)$ **using** *continuous-sub*[*of at x* $\lambda x.\ x$ $\lambda x.\ a$] *continuous-at-id*[*of x*] *continuous-const*[*of at x a*] **by** *auto* **}**
moreover have $\{x.\ x - a \in s\} = op + a\ `\ s$ **apply** *auto* **unfolding** *image-iff* **apply**(*rule-tac* $x{=}x - a$ **in** *bexI*) **by** *auto*
ultimately show *?thesis* **using** *continuous-open-preimage-univ*[*of* $\lambda x.\ x - a$ *s*] **using** *assms* **by** *auto*
qed

lemma *open-affinity*:
assumes *open s* $c \neq 0$
shows *open* $((\lambda x.\ a + c\ *s\ x)\ `\ s)$
proof−
have $*{:}(\lambda x.\ a + c\ *s\ x) = (\lambda x.\ a + x) \circ (\lambda x.\ c\ *s\ x)$ **unfolding** *o-def* **..**
have $op + a\ `\ op\ *s\ c\ `\ s = (op + a \circ op\ *s\ c)\ `\ s$ **by** *auto*
thus *?thesis* **using** *assms open-translation*[*of op* $*s$ *c* ` *s a*] **unfolding** $*$ **by** *auto*
qed

lemma *interior-translation*: $interior\ ((\lambda x.\ a + x)\ `\ s) = (\lambda x.\ a + x)\ `\ (interior\ s)$
proof (*rule set-ext*, *rule*)
fix x **assume** $x \in interior\ (op + a\ `\ s)$
then obtain e **where** $e{>}0$ **and** $e{:}ball\ x\ e \subseteq op + a\ `\ s$ **unfolding** *mem-interior* **by** *auto*
hence $ball\ (x - a)\ e \subseteq s$ **unfolding** *subset-eq Ball-def mem-ball dist-def* **apply** *auto* **apply**(*erule-tac* $x{=}a + xa$ **in** *allE*) **unfolding** *ab-group-add-class.diff-diff-eq*[*THEN sym*] **by** *auto*
thus $x \in op + a\ `\ interior\ s$ **unfolding** *image-iff* **apply**(*rule-tac* $x{=}x - a$ **in** *bexI*) **unfolding** *mem-interior* **using** ‹$e > 0$› **by** *auto*
next
fix x **assume** $x \in op + a\ `\ interior\ s$
then obtain $y\ e$ **where** $e{>}0$ **and** $e{:}ball\ y\ e \subseteq s$ **and** $y{:}x = a + y$ **unfolding** *image-iff Bex-def mem-interior* **by** *auto*
{ fix z **have** $*{:}a + y - z = y + a - z$ **by** *auto*
assume $z{\in}ball\ x\ e$
hence $z - a \in s$ **using** e[*unfolded subset-eq*, *THEN bspec*[**where** $x{=}z - a$]] **unfolding** *mem-ball dist-def y ab-group-add-class.diff-diff-eq2* $*$ **by** *auto*
hence $z \in op + a\ `\ s$ **unfolding** *image-iff* **by**(*auto intro*!: *bexI*[**where** $x{=}z - a$]) **}**

hence *ball x e* ⊆ *op + a ' s* **unfolding** *subset-eq* **by** *auto*
thus *x* ∈ *interior* (*op + a ' s*) **unfolding** *mem-interior* **using** ‹*e>0*› **by** *auto*
qed

64.29 Preservation of compactness and connectedness under continuous function.

lemma *compact-continuous-image*:
assumes *continuous-on s f compact s*
shows *compact*(*f ' s*)
proof−
{ fix *x* **assume** *x*:∀ *n*::*nat*. *x n* ∈ *f ' s*
then obtain *y* **where** *y*:∀ *n*. *y n* ∈ *s* ∧ *x n* = *f* (*y n*) **unfolding** *image-iff Bex-def* **using** *choice*[*of* λ*n xa*. *xa* ∈ *s* ∧ *x n* = *f xa*] **by** *auto*
then obtain *l r* **where** *l*∈*s* **and** *r*:∀ *m n*. *m* < *n* ⟶ *r m* < *r n* **and** *lr*:((*y* ∘ *r*) −−−> *l*) *sequentially* **using** *assms*(*2*)[*unfolded compact-def*, *THEN spec*[**where** *x*=*y*]] **by** *auto*
{ fix *e*::*real* **assume** *e>0*
then obtain *d* **where** *d>0* **and** *d*:∀ *x'*∈*s*. *dist x' l* < *d* ⟶ *dist* (*f x'*) (*f l*) < *e* **using** *assms*(*1*)[*unfolded continuous-on-def*, *THEN bspec*[**where** *x*=*l*], *OF* ‹*l*∈*s*›] **by** *auto*
then obtain *N*::*nat* **where** *N*:∀ *n*≥*N*. *dist* ((*y* ∘ *r*) *n*) *l* < *d* **using** *lr*[*unfolded Lim-sequentially*, *THEN spec*[**where** *x*=*d*]] **by** *auto*
{ fix *n*::*nat* **assume** *n*≥*N* **hence** *dist* ((*x* ∘ *r*) *n*) (*f l*) < *e* **using** *N*[*THEN spec*[**where** *x*=*n*]] *d*[*THEN bspec*[**where** *x*=*y* (*r n*)]] *y*[*THEN spec*[**where** *x*=*r n*]] **by** *auto* **}**
hence ∃ *N*. ∀ *n*≥*N*. *dist* ((*x* ∘ *r*) *n*) (*f l*) < *e* **by** *auto* **}**
hence ∃ *l*∈*f ' s*. ∃ *r*. (∀ *m n*. *m* < *n* ⟶ *r m* < *r n*) ∧ ((*x* ∘ *r*) −−−> *l*) *sequentially* **unfolding** *Lim-sequentially* **using** *r lr* ‹*l*∈*s*› **by** *auto* **}**
thus *?thesis* **unfolding** *compact-def* **by** *auto*
qed

lemma *connected-continuous-image*:
assumes *continuous-on s f connected s*
shows *connected*(*f ' s*)
proof−
{ fix *T* **assume** *as*: *T* ≠ {} *T* ≠ *f ' s openin* (*subtopology euclidean* (*f ' s*)) *T closedin* (*subtopology euclidean* (*f ' s*)) *T*
have {*x* ∈ *s*. *f x* ∈ *T*} = {} ∨ {*x* ∈ *s*. *f x* ∈ *T*} = *s*
using *assms*(*1*)[*unfolded continuous-on-open*, *THEN spec*[**where** *x*=*T*]]
using *assms*(*1*)[*unfolded continuous-on-closed*, *THEN spec*[**where** *x*=*T*]]
using *assms*(*2*)[*unfolded connected-clopen*, *THEN spec*[**where** *x*={*x* ∈ *s*. *f x* ∈ *T*}]] *as*(*3*,*4*) **by** *auto*
hence *False* **using** *as*(*1*,*2*)
using *as*(*4*)[*unfolded closedin-def topspace-euclidean-subtopology*] **by** *auto* **}**
thus *?thesis* **unfolding** *connected-clopen* **by** *auto*
qed

Continuity implies uniform continuity on a compact domain.

lemma *compact-uniformly-continuous*:
assumes *continuous-on s f* *compact s*
shows *uniformly-continuous-on s f*
proof−
{ fix *x* **assume** *x*:*x*∈*s*
hence ∀*xa*. ∃*y*. 0 < *xa* ⟶ (*y* > 0 ∧ (∀*x'*∈*s*. *dist x' x* < *y* ⟶ *dist* (*f x'*) (*f x*) < *xa*)) **using** *assms*(*1*)[*unfolded continuous-on-def*, *THEN bspec*[**where** *x*=*x*]] **by** *auto*
hence ∃*fa*. ∀*xa*>*0*. ∀*x'*∈*s*. *fa xa* > *0* ∧ (*dist x' x* < *fa xa* ⟶ *dist* (*f x'*) (*f x*) < *xa*) **using** *choice*[*of* λ*e d*. *e*>*0* ⟶ *d*>*0* ∧(∀*x'*∈*s*. (*dist x' x* < *d* ⟶ *dist* (*f x'*) (*f x*) < *e*))] **by** *auto* **}**
then have ∀*x*∈*s*. ∃*y*. ∀*xa*. *0* < *xa* ⟶ (∀*x'*∈*s*. *y xa* > *0* ∧ (*dist x' x* < *y xa* ⟶ *dist* (*f x'*) (*f x*) < *xa*)) **by** *auto*
then obtain *d* **where** *d*:∀*e*>*0*. ∀*x*∈*s*. ∀*x'*∈*s*. *d x e* > *0* ∧ (*dist x' x* < *d x e* ⟶ *dist* (*f x'*) (*f x*) < *e*)
using *bchoice*[*of s* λ*x fa*. ∀*xa*>*0*. ∀*x'*∈*s*. *fa xa* > *0* ∧ (*dist x' x* < *fa xa* ⟶ *dist* (*f x'*) (*f x*) < *xa*)] **by** *blast*

{ fix *e*::*real* **assume** *e*>*0*

{ fix *x* **assume** *x*∈*s* **hence** *x* ∈ *ball x* (*d x* (*e* / *2*)) **unfolding** *centre-in-ball* **using** *d*[*THEN spec*[**where** *x*=*e*/*2*]] **using** ‹*e*>*0*› **by** *auto* **}**
hence *s* ⊆ ⋃{*ball x* (*d x* (*e* / *2*)) |*x*. *x* ∈ *s*} **unfolding** *subset-eq* **by** *auto*
moreover
{ fix *b* **assume** *b*∈{*ball x* (*d x* (*e* / *2*)) |*x*. *x* ∈ *s*} **hence** *open b* **by** *auto* **}**
ultimately obtain *ea* **where** *ea*>*0* **and** *ea*:∀*x*∈*s*. ∃*b*∈{*ball x* (*d x* (*e* / *2*)) |*x*. *x* ∈ *s*}. *ball x ea* ⊆ *b* **using** *heine-borel-lemma*[*OF assms*(*2*), *of* {*ball x* (*d x* (*e* / *2*)) | *x*. *x*∈*s* }] **by** *auto*

{ fix *x y* **assume** *x*∈*s* *y*∈*s* **and** *as*:*dist y x* < *ea*
obtain *z* **where** *z*∈*s* **and** *z*:*ball x ea* ⊆ *ball z* (*d z* (*e* / *2*)) **using** *ea*[*THEN bspec*[**where** *x*=*x*]] **and** ‹*x*∈*s*› **by** *auto*
hence *x*∈*ball z* (*d z* (*e* / *2*)) **using** ‹*ea*>*0*› **unfolding** *subset-eq* **by** *auto*
hence *dist* (*f z*) (*f x*) < *e* / *2* **using** *d*[*THEN spec*[**where** *x*=*e*/*2*]] **and** ‹*e*>*0*› **and** ‹*x*∈*s*› **and** ‹*z*∈*s*›
by (*auto* *simp add*: *dist-sym*)
moreover have *y*∈*ball z* (*d z* (*e* / *2*)) **using** *as* **and** ‹*ea*>*0*› **and** *z*[*unfolded subset-eq*]
by (*auto simp add*: *dist-sym*)
hence *dist* (*f z*) (*f y*) < *e* / *2* **using** *d*[*THEN spec*[**where** *x*=*e*/*2*]] **and** ‹*e*>*0*› **and** ‹*y*∈*s*› **and** ‹*z*∈*s*›
by (*auto* *simp add*: *dist-sym*)
ultimately have *dist* (*f y*) (*f x*) < *e* **using** *dist-triangle-half-r*[*of f z f x e f y*]
by (*auto simp add*: *dist-sym*) **}**
then have ∃*d*>*0*. ∀*x*∈*s*. ∀*x'*∈*s*. *dist x' x* < *d* ⟶ *dist* (*f x'*) (*f x*) < *e* **using** ‹*ea*>*0*› **by** *auto* **}**
thus *?thesis* **unfolding** *uniformly-continuous-on-def* **by** *auto*
qed

Continuity of inverse function on compact domain.

lemma *continuous-on-inverse*:
assumes *continuous-on s f* *compact s* $\forall x \in s.\ g\ (f\ x) = x$
shows *continuous-on* $(f\ `\ s)\ g$
proof−
have $*$:$g\ `\ f\ `\ s = s$ **using** *assms*(3) **by** (*auto simp add*: *image-iff*)
{ fix *t* **assume** *t*:*closedin* (*subtopology euclidean* $(g\ `\ f\ `\ s))\ t$
then obtain *T* **where** *T*: *closed T t* $= s \cap T$ **unfolding** *closedin-closed* **unfolding** $*$ **by** *auto*
have *continuous-on* $(s \cap T)\ f$ **using** *continuous-on-subset*[*OF assms*(1), *of s* $\cap\ t$]
unfolding *T*(2) **and** *Int-left-absorb* **by** *auto*
moreover have *compact* $(s \cap T)$
using *assms*(2) **unfolding** *compact-eq-bounded-closed*
using *bounded-subset*[*of s s* $\cap$ *T*] **and** *T*(1) **by** *auto*
ultimately have *closed* $(f\ `\ t)$ **using** *T*(1) **unfolding** *T*(2)
using *compact-continuous-image* **unfolding** *compact-eq-bounded-closed* **by** *auto*
moreover have $\{x \in f\ `\ s.\ g\ x \in t\} = f\ `\ s \cap f\ `\ t$ **using** *assms*(3) **unfolding** *T*(2) **by** *auto*
ultimately have *closedin* (*subtopology euclidean* $(f\ `\ s))\ \{x \in f\ `\ s.\ g\ x \in t\}$
unfolding *closedin-closed* **by** *auto* **}**
thus *?thesis* **unfolding** *continuous-on-closed* **by** *auto*
qed

64.30 A uniformly convergent limit of continuous functions is continuous.

lemma *continuous-uniform-limit*:
assumes $\neg$ (*trivial-limit net*) *eventually* $(\lambda n.$ *continuous-on s* $(f\ n))$ *net*
$\forall\, e{>}0.$ *eventually* $(\lambda n.\ \forall x \in s.\ norm(f\ n\ x - g\ x) < e)$ *net*
shows *continuous-on s g*
proof−
{ fix *x* **and** *e*::*real* **assume** *x*∈*s e*>*0*
have *eventually* $(\lambda n.\ \forall x{\in}s.\ norm\ (f\ n\ x - g\ x) < e\ /\ 3)$ *net* **using** ‹*e*>*0*› *assms*(3)[*THEN spec*[**where** *x*=*e*/*3*]] **by** *auto*
then obtain *n* **where** *n*:$\forall$ *xa*∈*s*. *norm* $(f\ n\ xa - g\ xa) < e\ /\ 3$ *continuous-on s* $(f\ n)$
using *eventually-and*[*of* $(\lambda n.\ \forall x{\in}s.\ norm\ (f\ n\ x - g\ x) < e\ /\ 3)$ $(\lambda n.$ *continuous-on s* $(f\ n))$ *net*] *assms*(1,2) *eventually-happens* **by** *blast*
have $e\ /\ 3 > 0$ **using** ‹*e*>*0*› **by** *auto*
then obtain *d* **where** *d*>*0* **and** *d*:$\forall\, x'{\in}s.$ *dist x' x* $< d \longrightarrow$ *dist* $(f\ n\ x')\ (f\ n\ x) < e\ /\ 3$
using *n*(2)[*unfolded continuous-on-def*, *THEN bspec*[**where** *x*=*x*], *OF* ‹*x*∈*s*›, *THEN spec*[**where** *x*=*e*/*3*]] **by** *blast*
{ fix *y* **assume** *y*∈*s dist y x* $< d$
hence *dist* $(f\ n\ y)\ (f\ n\ x) < e\ /\ 3$ **using** *d*[*THEN bspec*[**where** *x*=*y*]] **by** *auto*
hence *norm* $(f\ n\ y - g\ x) < 2 * e\ /\ 3$ **using** *norm-triangle-lt*[*of f n y* − *f n x f n x* − *g x* 2∗*e*/*3*]

using *n(1)*[*THEN bspec*[**where** *x=x*], *OF* ‹*x∈s*›] **unfolding** *dist-def* **unfolding** *ab-group-add-class.ab-diff-minus* **by** *auto*
hence *dist* (*g y*) (*g x*) < *e* **unfolding** *dist-def* **using** *n(1)*[*THEN bspec*[**where** *x=y*], *OF* ‹*y∈s*›]
unfolding *norm-minus-cancel*[*of f n y − g y*, *THEN sym*] **using** *norm-triangle-lt*[*of f n y − g x g y − f n y e*] **by** (*auto simp add*: *uminus-add-conv-diff*) **}**
hence ∃ *d>0*. ∀ *x'∈s*. *dist x' x* < *d* ⟶ *dist* (*g x'*) (*g x*) < *e* **using** ‹*d>0*› **by** *auto* **}**
thus *?thesis* **unfolding** *continuous-on-def* **by** *auto*
qed

64.31 Topological properties of linear functions.

lemma *linear-lim-0*: **fixes** *f*::*real^'a*::*finite* ⇒ *real^'b*::*finite*
assumes *linear f* **shows** (*f* −−−> *0*) (*at* (*0*))
proof−
obtain *B* **where** *B>0* **and** *B*:∀ *x*. *norm* (*f x*) ≤ *B* ∗ *norm x* **using** *linear-bounded-pos*[*OF assms*] **by** *auto*
{ fix *e*::*real* **assume** *e>0*
{ fix *x*::*real^'a* **assume** *norm x* < *e* / *B*
hence *B* ∗ *norm x* < *e* **using** ‹*B>0*› **using** *mult-strict-right-mono*[*of norm x e* / *B B*] **unfolding** *real-mult-commute* **by** *auto*
hence *norm* (*f x*) < *e* **using** *B*[*THEN spec*[**where** *x=x*]] ‹*B>0*› **using** *order-le-less-trans*[*of norm* (*f x*) *B* ∗ *norm x e*] **by** *auto* **}**
moreover have *e* / *B* > *0* **using** ‹*e>0*› ‹*B>0*› *divide-pos-pos* **by** *auto*
ultimately have ∃ *d>0*. ∀ *x*. *0* < *dist x 0* ∧ *dist x 0* < *d* ⟶ *dist* (*f x*) *0* < *e* **unfolding** *dist-def* **by** *auto* **}**
thus *?thesis* **unfolding** *Lim-at* **by** *auto*
qed

lemma *linear-continuous-at*:
assumes *linear f* **shows** *continuous* (*at a*) *f*
unfolding *continuous-at Lim-at-zero*[*of f f a a*] **using** *linear-lim-0*[*OF assms*]
unfolding *Lim-null*[*of* λ*x*. *f* (*a* + *x*)] **unfolding** *linear-sub*[*OF assms*, *THEN sym*] **by** *auto*

lemma *linear-continuous-within*:
linear f ==> *continuous* (*at x within s*) *f*
using *continuous-at-imp-continuous-within*[*of x f s*] **using** *linear-continuous-at*[*of f*] **by** *auto*

lemma *linear-continuous-on*:
linear f ==> *continuous-on s f*
using *continuous-at-imp-continuous-on*[*of s f*] **using** *linear-continuous-at*[*of f*] **by** *auto*

Also bilinear functions, in composition form.

lemma *bilinear-continuous-at-compose*:
continuous (*at x*) *f* ⟹ *continuous* (*at x*) *g* ⟹ *bilinear h*
==> *continuous* (*at x*) (λ*x*. *h* (*f x*) (*g x*))

unfolding *continuous-at* **using** *Lim-bilinear*[*of f f f x* (*at x*) *g g x h*] **by** *auto*

lemma *bilinear-continuous-within-compose*:
continuous (*at x within s*) *f* ⟹ *continuous* (*at x within s*) *g* ⟹ *bilinear h*
==> *continuous* (*at x within s*) (λ*x*. *h* (*f x*) (*g x*))
unfolding *continuous-within* **using** *Lim-bilinear*[*of f f f x*] **by** *auto*

lemma *bilinear-continuous-on-compose*:
continuous-on s f ⟹ *continuous-on s g* ⟹ *bilinear h*
==> *continuous-on s* (λ*x*. *h* (*f x*) (*g x*))
unfolding *continuous-on-eq-continuous-within* **apply** *auto* **apply**(*erule-tac x*=*x* **in** *ballE*) **apply** *auto* **apply**(*erule-tac x*=*x* **in** *ballE*) **apply** *auto*
using *bilinear-continuous-within-compose*[*of - s f g h*] **by** *auto*

64.32 Topological stuff lifted from and dropped to R

lemma *open-vec1*:
open(*vec1* ' *s*) ⟷
(∀ *x* ∈ *s*. ∃ *e*>*0*. ∀ *x'*. *abs*(*x'* − *x*) < *e* −−> *x'* ∈ *s*) (**is** *?lhs* = *?rhs*)
unfolding *open-def* **apply** *simp* **unfolding** *forall-vec1 dist-vec1 vec1-in-image-vec1* **by** *simp*

lemma *islimpt-approachable-vec1*:
(*vec1 x*) *islimpt* (*vec1* ' *s*) ⟷
(∀ *e*>*0*. ∃ *x'*∈ *s*. *x'* ≠ *x* ∧ *abs*(*x'* − *x*) < *e*)
by (*auto simp add*: *islimpt-approachable dist-vec1 vec1-eq*)

lemma *closed-vec1*:
closed (*vec1* ' *s*) ⟷
(∀ *x*. (∀ *e*>*0*. ∃ *x'* ∈ *s*. *x'* ≠ *x* ∧ *abs*(*x'* − *x*) < *e*)
−−> *x* ∈ *s*)
unfolding *closed-limpt islimpt-approachable forall-vec1* **apply** *simp*
unfolding *dist-vec1 vec1-in-image-vec1 abs-minus-commute* **by** *auto*

lemma *continuous-at-vec1-range*:
continuous (*at x*) (*vec1 o f*) ⟷ (∀ *e*>*0*. ∃ *d*>*0*.
∀ *x'*. *norm*(*x'* − *x*) < *d* −−> *abs*(*f x'* − *f x*) < *e*)
unfolding *continuous-at* **unfolding** *Lim-at* **apply** *simp* **unfolding** *dist-vec1* **unfolding** *dist-nz*[*THEN sym*] **unfolding** *dist-def* **apply** *auto*
apply(*erule-tac x*=*e* **in** *allE*) **apply** *auto* **apply** (*rule-tac x*=*d* **in** *exI*) **apply** *auto* **apply** (*erule-tac x*=*x'* **in** *allE*) **apply** *auto*
apply(*erule-tac x*=*e* **in** *allE*) **by** *auto*

lemma *continuous-on-vec1-range*:
continuous-on s (*vec1 o f*) ⟷ (∀ *x* ∈ *s*. ∀ *e*>*0*. ∃ *d*>*0*. (∀ *x'* ∈ *s*. *norm*(*x'* − *x*) < *d* −−> *abs*(*f x'* − *f x*) < *e*))
unfolding *continuous-on-def* **apply** (*simp del*: *dist-sym*) **unfolding** *dist-vec1* **unfolding** *dist-def* **..**

lemma *continuous-at-vec1-norm*:
$\forall x.$ *continuous* (*at* x) (*vec1 o norm*)
unfolding *continuous-at-vec1-range* **using** *real-abs-sub-norm order-le-less-trans*
by *blast*

lemma *continuous-on-vec1-norm*:
$\forall s.$ *continuous-on* s (*vec1 o norm*)
unfolding *continuous-on-vec1-range norm-vec1*[*THEN sym*] **by** (*metis norm-vec1 order-le-less-trans real-abs-sub-norm*)

lemma *continuous-at-vec1-component*:
shows *continuous* (*at* (a::*real^'a*::*finite*)) (λ $x.$ *vec1*(x\$$i$))
proof−
{ fix e::*real* **and** x **assume** $0 <$ *dist* x a *dist* x $a < e$ $e>0$
hence $|x \$ i - a \$ i| < e$ **using** *component-le-norm*[*of* $x - a$ i] **unfolding** *dist-def* **by** *auto* **}**
thus *?thesis* **unfolding** *continuous-at tendsto-def eventually-at dist-vec1* **by** *auto*
qed

lemma *continuous-on-vec1-component*:
shows *continuous-on* s (λ x::*real^'a*::*finite*. *vec1*(x\$$i$))
proof−
{ fix e::*real* **and** x xa **assume** $x \in s$ $e>0$ $xa \in s$ $0 <$ *norm* ($xa - x$) $\wedge$ *norm* ($xa - x$) $< e$
hence $|xa \$ i - x \$ i| < e$ **using** *component-le-norm*[*of* $xa - x$ i] **by** *auto* **}**
thus *?thesis* **unfolding** *continuous-on Lim-within dist-vec1* **unfolding** *dist-def*
by *auto*
qed

lemma *continuous-at-vec1-infnorm*:
continuous (*at* x) (*vec1 o infnorm*)
unfolding *continuous-at Lim-at o-def* **unfolding** *dist-vec1* **unfolding** *dist-def*
apply *auto* **apply** (*rule-tac* x=e **in** *exI*) **apply** *auto*
using *order-trans*[*OF real-abs-sub-infnorm infnorm-le-norm*, *of* - x] **by** (*metis xt1*(*7*))

Hence some handy theorems on distance, diameter etc. of/from a set.

lemma *compact-attains-sup*:
assumes *compact* (*vec1* ' s) $s \neq \{\}$
shows $\exists x \in s.\ \forall y \in s.\ y \leq x$
proof−
from *assms*(*1*) **have** a:*bounded* (*vec1* ' s) *closed* (*vec1* ' s) **unfolding** *compact-eq-bounded-closed*
by *auto*
{ fix e::*real* **assume** *as*: $\forall x \in s.\ x \leq$ *rsup* s *rsup* $s \notin s$ $0 < e$ $\forall x' \in s.\ x' =$ *rsup* $s \vee \neg$ *rsup* $s - x' < e$
have *isLub UNIV* s (*rsup* s) **using** *rsup*[*OF assms*(*2*)] **unfolding** *setle-def*
using *as*(*1*) **by** *auto*
moreover have *isUb UNIV* s (*rsup* $s - e$) **unfolding** *isUb-def* **unfolding** *setle-def* **using** *as*(*4*,*2*) **by** *auto*

ultimately have *False* **using** *isLub-le-isUb*[*of UNIV s rsup s rsup s − e*] **using** ‹*e>0*› **by** *auto* **}**
thus *?thesis* **using** *bounded-has-rsup*(*1*)[*OF a*(*1*) *assms*(*2*)] **using** *a*(*2*)[*unfolded closed-vec1*, *THEN spec*[**where** *x=rsup s*]]
apply(*rule-tac x=rsup s* **in** *bexI*) **by** *auto*
qed

lemma *compact-attains-inf*:
assumes *compact* (*vec1 ' s*) *s* ≠ {} **shows** ∃ *x* ∈ *s*. ∀ *y* ∈ *s*. *x* ≤ *y*
proof−
from *assms*(*1*) **have** *a*:*bounded* (*vec1 ' s*) *closed* (*vec1 ' s*) **unfolding** *compact-eq-bounded-closed* **by** *auto*
{ fix *e*::*real* **assume** *as*: ∀ *x*∈*s*. *x* ≥ *rinf s* *rinf s* ∉ *s* *0* < *e*
∀ *x'*∈*s*. *x'* = *rinf s* ∨ ¬ *abs* (*x'* − *rinf s*) < *e*
have *isGlb UNIV s* (*rinf s*) **using** *rinf*[*OF assms*(*2*)] **unfolding** *setge-def* **using** *as*(*1*) **by** *auto*
moreover
{ fix *x* **assume** *x* ∈ *s*
hence ∗:*abs* (*x* − *rinf s*) = *x* − *rinf s* **using** *as*(*1*)[*THEN bspec*[**where** *x=x*]] **by** *auto*
have *rinf s* + *e* ≤ *x* **using** *as*(*4*)[*THEN bspec*[**where** *x=x*]] **using** *as*(*2*) ‹*x*∈*s*› **unfolding** ∗ **by** *auto* **}**
hence *isLb UNIV s* (*rinf s* + *e*) **unfolding** *isLb-def* **and** *setge-def* **by** *auto*
ultimately have *False* **using** *isGlb-le-isLb*[*of UNIV s rinf s rinf s* + *e*] **using** ‹*e>0*› **by** *auto* **}**
thus *?thesis* **using** *bounded-has-rinf*(*1*)[*OF a*(*1*) *assms*(*2*)] **using** *a*(*2*)[*unfolded closed-vec1*, *THEN spec*[**where** *x=rinf s*]]
apply(*rule-tac x=rinf s* **in** *bexI*) **by** *auto*
qed

lemma *continuous-attains-sup*:
compact s ⟹ *s* ≠ {} ⟹ *continuous-on s* (*vec1 o f*)
==> (∃ *x* ∈ *s*. ∀ *y* ∈ *s*. *f y* ≤ *f x*)
using *compact-attains-sup*[*of f ' s*]
using *compact-continuous-image*[*of s vec1* ∘ *f*] **unfolding** *image-compose* **by** *auto*

lemma *continuous-attains-inf*:
compact s ⟹ *s* ≠ {} ⟹ *continuous-on s* (*vec1 o f*)
==> (∃ *x* ∈ *s*. ∀ *y* ∈ *s*. *f x* ≤ *f y*)
using *compact-attains-inf*[*of f ' s*]
using *compact-continuous-image*[*of s vec1* ∘ *f*] **unfolding** *image-compose* **by** *auto*

lemma *distance-attains-sup*:
assumes *compact s s* ≠ {}
shows ∃ *x* ∈ *s*. ∀ *y* ∈ *s*. *dist a y* ≤ *dist a x*
proof−
{ fix *x* **assume** *x*∈*s* **fix** *e*::*real* **assume** *e>0*

{ fix x' **assume** $x'{\in}s$ **and** *as*:$norm\ (x' - x) < e$
hence $|norm\ (x' - a) - norm\ (x - a)| < e$
using *real-abs-sub-norm*[*of* $x' - a\ x - a$] **by** *auto* **}**
hence $\exists\, d{>}0.\ \forall\, x'{\in}s.\ norm\ (x' - x) < d \longrightarrow |dist\ x'\ a - dist\ x\ a| < e$ **using** ‹$e{>}0$› **unfolding** *dist-def* **by** *auto* **}**
thus *?thesis* **using** *assms*
using *continuous-attains-sup*[*of* $s\ \lambda x.\ dist\ a\ x$]
unfolding *continuous-on-vec1-range* **by** (*auto simp add*: *dist-sym*)
qed

For *minimal* distance, we only need closure, not compactness.

lemma *distance-attains-inf*:
assumes *closed* s $s \neq \{\}$
shows $\exists\, x \in s.\ \forall\, y \in s.\ dist\ a\ x \leq dist\ a\ y$
proof−
from *assms*(*2*) **obtain** b **where** $b{\in}s$ **by** *auto*
let *?B* $= cball\ a\ (dist\ b\ a) \cap s$
have $b \in$ *?B* **using** ‹$b{\in}s$› **by** (*simp add*: *dist-sym*)
hence *?B* $\neq \{\}$ **by** *auto*
moreover
{ fix x **assume** $x{\in}$*?B*
fix e::*real* **assume** $e{>}0$
{ fix x' **assume** $x'{\in}$*?B* **and** *as*:$norm\ (x' - x) < e$
hence $|norm\ (x' - a) - norm\ (x - a)| < e$
using *real-abs-sub-norm*[*of* $x' - a\ x - a$] **by** *auto* **}**
hence $\exists\, d{>}0.\ \forall\, x'{\in}$*?B*$.\ norm\ (x' - x) < d \longrightarrow |dist\ x'\ a - dist\ x\ a| < e$ **using** ‹$e{>}0$› **unfolding** *dist-def* **by** *auto* **}**
hence *continuous-on* $(cball\ a\ (dist\ b\ a) \cap s)\ (vec1 \circ dist\ a)$ **unfolding** *continuous-on-vec1-range* **by** (*auto simp add*: *dist-sym*)
moreover have *compact ?B* **using** *compact-cball*[*of a dist b a*] **unfolding** *compact-eq-bounded-closed* **using** *bounded-Int* **and** *closed-Int* **and** *assms*(*1*) **by** *auto*
ultimately obtain x **where** $x{\in}cball\ a\ (dist\ b\ a) \cap s\ \forall\, y{\in}cball\ a\ (dist\ b\ a) \cap s.$ *dist a x* $\leq$ *dist a y* **using** *continuous-attains-inf*[*of ?B dist a*] **by** *fastsimp*
thus *?thesis* **by** *fastsimp*
qed

64.33 We can now extend limit compositions to consider the scalar multiplier.

lemma *Lim-mul*:
assumes $((vec1\ o\ c) \longrightarrow vec1\ d)\ net\ \ (f \longrightarrow l)\ net$
shows $((\lambda x.\ c(x) *s f\ x) \longrightarrow (d *s l))\ net$
proof−
have *bilinear* $(\lambda x.\ op *s\ (dest\text{-}vec1\ (x{::}real\hat{}1)))$ **unfolding** *bilinear-def linear-def*
unfolding *dest-vec1-add dest-vec1-cmul*
apply *vector* **apply** *auto* **unfolding** *semiring-class.right-distrib semiring-class.left-distrib* **by** *auto*
thus *?thesis* **using** *Lim-bilinear*[*OF assms*, *of* $\lambda x\ y.\ (dest\text{-}vec1\ x) *s\ y$] **by** *auto*

qed

lemma *Lim-vmul*:
*((vec1 o c) ---> vec1 d) net ==> ((λx. c(x) *s v) ---> d *s v) net*
using *Lim-mul*[*of c d net λx. v v*] **using** *Lim-const*[*of v*] **by** *auto*

lemma *continuous-vmul*:
*continuous net (vec1 o c) ==> continuous net (λx. c(x) *s v)*
unfolding *continuous-def* **using** *Lim-vmul*[*of c*] **by** *auto*

lemma *continuous-mul*:
continuous net (vec1 o c) ⟹ continuous net f
*==> continuous net (λx. c(x) *s f x)*
unfolding *continuous-def* **using** *Lim-mul*[*of c*] **by** *auto*

lemma *continuous-on-vmul*:
*continuous-on s (vec1 o c) ==> continuous-on s (λx. c(x) *s v)*
unfolding *continuous-on-eq-continuous-within* **using** *continuous-vmul*[*of - c*] **by** *auto*

lemma *continuous-on-mul*:
continuous-on s (vec1 o c) ⟹ continuous-on s f
*==> continuous-on s (λx. c(x) *s f x)*
unfolding *continuous-on-eq-continuous-within* **using** *continuous-mul*[*of - c*] **by** *auto*

And so we have continuity of inverse.

lemma *Lim-inv*:
assumes *((vec1 o f) ---> vec1 l) (net::'a net) l ≠ 0*
shows *((vec1 o inverse o f) ---> vec1(inverse l)) net*
proof(*cases trivial-limit net*)
case *True* **thus** *?thesis* **unfolding** *tendsto-def* **unfolding** *eventually-def* **by** *auto*
next
case *False* **note** *ntriv = this*
{ fix *e::real* **assume** *e>0*
hence *0 < min (|l| / 2) (l² * e / 2)* **using** ‹*l≠0*› *mult-pos-pos*[*of l^2 e/2*] **by** *auto*
then obtain *y* **where** *y1:∃x. netord net x y* **and**
*y:∀x. netord net x y ⟶ dist ((vec1 ∘ f) x) (vec1 l) < min (|l| / 2) (l² * e / 2)* **using** *ntriv*
using *assms*(*1*)[*unfolded tendsto-def eventually-def*, *THEN spec*[**where** *x=min (abs l / 2) (l ^ 2 * e / 2)*]] **by** *auto*
{ fix *x* **assume** *netord net x y*
hence *∗:|f x − l| < min (|l| / 2) (l² * e / 2)* **using** *y*[*THEN spec*[**where** *x=x*]] **unfolding** *o-def dist-vec1* **by** *auto*
hence *fx0:f x ≠ 0* **using** ‹*l ≠ 0*› **by** *auto*
hence *fxl0*: *(f x) * l ≠ 0* **using** ‹*l ≠ 0*› **by** *auto*
from *∗* **have** *∗∗:|f x − l| < l² * e / 2* **by** *auto*

have $|f\ x| * 2 \geq |l|$ **using** $*$ **by** (*auto simp del*: *less-divide-eq-number-of1*)
hence $|f\ x| * 2 * |l| \geq |l| * |l|$ **unfolding** *mult-le-cancel-right* **by** *auto*
hence $|f\ x * l| * 2 \geq |l|\ \hat{}\ 2$ **unfolding** *real-mult-commute* **and** *power2-eq-square* **by** *auto*
hence $***$:*inverse* $|f\ x * l| \leq$ *inverse* $(l^2\ /\ 2)$ **using** *fxl0*
using *le-imp-inverse-le*[*of* $l\hat{}2\ /\ 2\ |f\ x * l|$] **by** *auto*

have *dist* $((vec1 \circ inverse \circ f)\ x)\ (vec1\ (inverse\ l)) < e$ **unfolding** *o-def* **unfolding** *dist-vec1*
unfolding *inverse-diff-inverse*[*OF fx0* ‹$l \neq 0$›] **apply** *simp*
unfolding *mult-commute*[*of inverse* $(f\ x)$]
unfolding *real-divide-def*[*THEN sym*]
unfolding *divide-divide-eq-left*
unfolding *nonzero-abs-divide*[*OF fxl0*]
using *mult-less-le-imp-less*[*OF* $**$, *of inverse* $|f\ x * l|$, *of inverse* $(l\hat{}2\ /\ 2)$] **using** $***$ **using** *fx0* ‹$l \neq 0$›
unfolding *inverse-eq-divide* **using** ‹$e > 0$› **by** *auto* **}**
hence $(\exists\, y.\ (\exists\, x.$ *netord net x y*$) \wedge (\forall\, x.$ *netord net x y* $\longrightarrow$ *dist* $((vec1 \circ inverse \circ f)\ x)\ (vec1\ (inverse\ l)) < e))$
using *y1* **by** *auto* **}**
thus *?thesis* **unfolding** *tendsto-def eventually-def* **by** *auto*
qed

lemma *continuous-inv*:
continuous net $(vec1\ o\ f) \Longrightarrow f(netlimit\ net) \neq 0$
$==>$ *continuous net* $(vec1\ o\ inverse\ o\ f)$
unfolding *continuous-def* **using** *Lim-inv* **by** *auto*

lemma *continuous-at-within-inv*:
assumes *continuous* $(at\ a\ within\ s)\ (vec1\ o\ f)\ f\ a \neq 0$
shows *continuous* $(at\ a\ within\ s)\ (vec1\ o\ inverse\ o\ f)$
proof(*cases trivial-limit* $(at\ a\ within\ s)$)
case *True* **thus** *?thesis* **unfolding** *continuous-def tendsto-def eventually-def* **by** *auto*
next
case *False* **note** $cs = this$
thus *?thesis* **using** *netlimit-within*[*OF cs*] *assms*(*2*) *continuous-inv*[*OF assms*(*1*)] **by** *auto*
qed

lemma *continuous-at-inv*:
continuous $(at\ a)\ (vec1\ o\ f) \Longrightarrow f\ a \neq 0$
$==>$ *continuous* $(at\ a)\ (vec1\ o\ inverse\ o\ f)$
using *within-UNIV*[*THEN sym*, *of a*] **using** *continuous-at-within-inv*[*of a UNIV*] **by** *auto*

64.34 Preservation properties for pasted sets.

lemma *bounded-pastecart*:

assumes *bounded s bounded t*
shows *bounded* { *pastecart x y* | *x y* . (*x* ∈ *s* ∧ *y* ∈ *t*)}
proof−
obtain *a b* **where** *ab*:∀ *x*∈*s*. *norm x* ≤ *a* ∀ *x*∈*t*. *norm x* ≤ *b* **using** *assms*[*unfolded bounded-def*] **by** *auto*
{ fix *x y* **assume** *x*∈*s y*∈*t*
hence *norm x* ≤ *a norm y* ≤ *b* **using** *ab* **by** *auto*
hence *norm* (*pastecart x y*) ≤ *a* + *b* **using** *norm-pastecart*[*of x y*] **by** *auto* **}**
thus *?thesis* **unfolding** *bounded-def* **by** *auto*
qed

lemma *closed-pastecart*:
assumes *closed s closed t*
shows *closed* {*pastecart x y* | *x y* . *x* ∈ *s* ∧ *y* ∈ *t*}
proof−
{ fix *x l* **assume** *as*:∀ *n*::*nat*. *x n* ∈ {*pastecart x y* |*x y*. *x* ∈ *s* ∧ *y* ∈ *t*} (*x* −−−> *l*) *sequentially*
{ fix *n*::*nat* **have** *fstcart* (*x n*) ∈ *s sndcart* (*x n*) ∈ *t* **using** *as*(*1*)[*THEN spec*[**where** *x*=*n*]] **by** *auto* **} note** ∗ = *this*
moreover
{ fix *e*::*real* **assume** *e*>*0*
then obtain *N*::*nat* **where** *N*:∀ *n*≥*N*. *dist* (*x n*) *l* < *e* **using** *as*(*2*)[*unfolded Lim-sequentially*, *THEN spec*[**where** *x*=*e*]] **by** *auto*
{ fix *n*::*nat* **assume** *n*≥*N*
hence *dist* (*fstcart* (*x n*)) (*fstcart l*) < *e dist* (*sndcart* (*x n*)) (*sndcart l*) < *e*
using *N*[*THEN spec*[**where** *x*=*n*]] *dist-fstcart*[*of x n l*] *dist-sndcart*[*of x n l*] **by** *auto* **}**
hence ∃ *N*. ∀ *n*≥*N*. *dist* (*fstcart* (*x n*)) (*fstcart l*) < *e* ∃ *N*. ∀ *n*≥*N*. *dist* (*sndcart* (*x n*)) (*sndcart l*) < *e* **by** *auto* **}**
ultimately have *fstcart l* ∈ *s sndcart l* ∈ *t*
using *assms*(*1*)[*unfolded closed-sequential-limits*, *THEN spec*[**where** *x*=λ*n*. *fstcart* (*x n*)], *THEN spec*[**where** *x*=*fstcart l*]]
using *assms*(*2*)[*unfolded closed-sequential-limits*, *THEN spec*[**where** *x*=λ*n*. *sndcart* (*x n*)], *THEN spec*[**where** *x*=*sndcart l*]]
unfolding *Lim-sequentially* **by** *auto*
hence *l* ∈ {*pastecart x y* |*x y*. *x* ∈ *s* ∧ *y* ∈ *t*} **using** *pastecart-fst-snd*[*THEN sym*, *of l*] **by** *auto* **}**
thus *?thesis* **unfolding** *closed-sequential-limits* **by** *auto*
qed

lemma *compact-pastecart*:
compact s ⟹ *compact t* ==> *compact* {*pastecart x y* | *x y* . *x* ∈ *s* ∧ *y* ∈ *t*}
unfolding *compact-eq-bounded-closed* **using** *bounded-pastecart*[*of s t*] *closed-pastecart*[*of s t*] **by** *auto*

Hence some useful properties follow quite easily.

lemma *compact-scaling*:
assumes *compact s* **shows** *compact* ((λ*x*. *c* ∗*s x*) ‘ *s*)
proof−

let *?f* = $\lambda x.\ c *s\ x$
have $*$:*linear ?f* **unfolding** *linear-def vector-smult-assoc vector-add-ldistrib real-mult-commute*
by *auto*
show *?thesis* **using** *compact-continuous-image*[*of s ?f*] *continuous-at-imp-continuous-on*[*of s ?f*]
using *linear-continuous-at*[*OF* $*$] *assms* **by** *auto*
qed

lemma *compact-negations*:
assumes *compact s* **shows** *compact* ($\lambda x.\ -x$) ' *s*)
proof−
have *uminus* ' *s* = ($\lambda x.\ -1 *s\ x$) ' *s* **apply** *auto* **unfolding** *image-iff pth-3* **by** *auto*
thus *?thesis* **using** *compact-scaling*[*OF assms*, *of* −*1*] **by** *auto*
qed

lemma *compact-sums*:
assumes *compact s compact t* **shows** *compact* {$x + y \mid x\ y.\ x \in s \wedge y \in t$}
proof−
have $*$:{$x + y \mid x\ y.\ x \in s \wedge y \in t$} =($\lambda z.$ *fstcart z* + *sndcart z*) ' {*pastecart x y* | *x y.* $x \in s \wedge y \in t$}
apply *auto* **unfolding** *image-iff* **apply**(*rule-tac x*=*pastecart xa y* **in** *bexI*)
unfolding *fstcart-pastecart sndcart-pastecart* **by** *auto*
have *linear* (λz::*real*^('*a* + '*a*). *fstcart z* + *sndcart z*) **unfolding** *linear-def*
unfolding *fstcart-add sndcart-add* **apply** *auto*
unfolding *vector-add-ldistrib fstcart-cmul*[*THEN sym*] *sndcart-cmul*[*THEN sym*] **by** *auto*
hence *continuous-on* {*pastecart x y* |*x y.* $x \in s \wedge y \in t$} ($\lambda z.$ *fstcart z* + *sndcart z*)
using *continuous-at-imp-continuous-on linear-continuous-at* **by** *auto*
thus *?thesis* **unfolding** $*$ **using** *compact-continuous-image compact-pastecart*[*OF assms*] **by** *auto*
qed

lemma *compact-differences*:
assumes *compact s compact t* **shows** *compact* {$x - y \mid x\ y.\ x \in s \wedge y \in t$}
proof−
have {$x - y \mid x\ y$::*real*^'*a*. $x \in s \wedge y \in t$} = {$x + y \mid x\ y.\ x \in s \wedge y \in$ (*uminus* ' *t*)}
apply *auto* **apply**(*rule-tac x*= *xa* **in** *exI*) **apply** *auto* **apply**(*rule-tac x*=*xa* **in** *exI*) **by** *auto*
thus *?thesis* **using** *compact-sums*[*OF assms*(*1*) *compact-negations*[*OF assms*(*2*)]]
by *auto*
qed

lemma *compact-translation*:
assumes *compact s* **shows** *compact* ($\lambda x.\ a + x$) ' *s*)
proof−
have {$x + y \mid x\ y.\ x \in s \wedge y \in \{a\}$} = ($\lambda x.\ a + x$) ' *s* **by** *auto*

thus *?thesis* **using** *compact-sums*[*OF assms compact-sing*[*of a*]] **by** *auto*
qed

lemma *compact-affinity*:
assumes *compact s* **shows** *compact* ((λ*x*. *a* + *c* ∗*s x*) ‘ *s*)
proof−
have *op* + *a* ‘ *op* ∗*s c* ‘ *s* = (λ*x*. *a* + *c* ∗*s x*) ‘ *s* **by** *auto*
thus *?thesis* **using** *compact-translation*[*OF compact-scaling*[*OF assms*], *of a c*]
by *auto*
qed

Hence we get the following.

lemma *compact-sup-maxdistance*:
assumes *compact s* *s* ≠ {}
shows ∃ *x*∈*s*. ∃ *y*∈*s*. ∀ *u*∈*s*. ∀ *v*∈*s*. *norm*(*u* − *v*) ≤ *norm*(*x* − *y*)
proof−
have {*x* − *y* | *x y* . *x*∈*s* ∧ *y*∈*s*} ≠ {} **using** ‹*s* ≠ {}› **by** *auto*
then obtain *x* **where** *x*:*x*∈{*x* − *y* |*x y*. *x* ∈ *s* ∧ *y* ∈ *s*} ∀ *y*∈{*x* − *y* |*x y*. *x* ∈ *s* ∧ *y* ∈ *s*}. *norm y* ≤ *norm x*
using *compact-differences*[*OF assms*(*1*) *assms*(*1*)]
using *distance-attains-sup*[*unfolded dist-def*, *of* {*x* − *y* | *x y* . *x*∈*s* ∧ *y*∈*s*} *0*]
by(*auto simp add*: *norm-minus-cancel*)
from *x*(*1*) **obtain** *a b* **where** *a*∈*s b*∈*s x* = *a* − *b* **by** *auto*
thus *?thesis* **using** *x*(*2*)[*unfolded* ‹*x* = *a* − *b*›] **by** *blast*
qed

We can state this in terms of diameter of a set.

definition *diameter s* = (*if s* = {} *then 0*::*real else rsup* {*norm*(*x* − *y*) | *x y*. *x* ∈ *s* ∧ *y* ∈ *s*})

lemma *diameter-bounded*:
assumes *bounded s*
shows ∀ *x*∈*s*. ∀ *y*∈*s*. *norm*(*x* − *y*) ≤ *diameter s*
∀ *d*>*0*. *d* < *diameter s* −−> (∃ *x*∈*s*. ∃ *y*∈*s*. *norm*(*x* − *y*) > *d*)
proof−
let *?D* = {*norm* (*x* − *y*) |*x y*. *x* ∈ *s* ∧ *y* ∈ *s*}
obtain *a* **where** *a*:∀ *x*∈*s*. *norm x* ≤ *a* **using** *assms*[*unfolded bounded-def*] **by** *auto*
{ fix *x y* **assume** *x* ∈ *s y* ∈ *s*
hence *norm* (*x* − *y*) ≤ *2* ∗ *a* **using** *norm-triangle-ineq*[*of x* −*y*, *unfolded norm-minus-cancel*] *a*[*THEN bspec*[**where** *x*=*x*]] *a*[*THEN bspec*[**where** *x*=*y*]] **by** (*auto simp add*: *ring-simps*) **}**
note ∗ = *this*
{ fix *x y* **assume** *x*∈*s y*∈*s* **hence** *s* ≠ {} **by** *auto*
have *lub*:*isLub UNIV ?D* (*rsup ?D*) **using** ∗ *rsup*[*of ?D*] **using** ‹*s*≠{}› **unfolding** *setle-def* **by** *auto*
have *norm*(*x* − *y*) ≤ *diameter s* **unfolding** *diameter-def* **using** ‹*s*≠{}› ∗[*OF* ‹*x*∈*s*› ‹*y*∈*s*›] ‹*x*∈*s*› ‹*y*∈*s*› *isLubD1*[*OF lub*] **unfolding** *setle-def* **by** *auto* **}**
moreover

{ fix *d*::*real* **assume** *d*>*0 d* < *diameter s*
hence *s*≠{} **unfolding** *diameter-def* **by** *auto*
hence *lub*:*isLub UNIV ?D* (*rsup ?D*) **using** ∗ *rsup*[*of ?D*] **unfolding** *setle-def* **by** *auto*
have ∃ *d′* ∈ *?D*. *d′* > *d*
proof(*rule ccontr*)
assume ¬ (∃ *d′*∈{*norm* (*x* − *y*) |*x y*. *x* ∈ *s* ∧ *y* ∈ *s*}. *d* < *d′*)
hence *as*:∀ *d′*∈*?D*. *d′* ≤ *d* **apply** *auto* **apply**(*erule-tac x*=*norm* (*x* − *y*) **in** *allE*) **by** *auto*
hence *isUb UNIV ?D d* **unfolding** *isUb-def* **unfolding** *setle-def* **by** *auto*
thus *False* **using** ‹*d* < *diameter s*› ‹*s*≠{}› *isLub-le-isUb*[*OF lub*, *of d*] **unfolding** *diameter-def* **by** *auto*
qed
hence ∃ *x*∈*s*. ∃ *y*∈*s*. *norm*(*x* − *y*) > *d* **by** *auto* **}**
ultimately show ∀ *x*∈*s*. ∀ *y*∈*s*. *norm*(*x* − *y*) ≤ *diameter s*
∀ *d*>*0*. *d* < *diameter s* −−> (∃ *x*∈*s*. ∃ *y*∈*s*. *norm*(*x* − *y*) > *d*) **by** *auto*
qed

lemma *diameter-bounded-bound*:
bounded s ⟹ *x* ∈ *s* ⟹ *y* ∈ *s* ==> *norm*(*x* − *y*) ≤ *diameter s*
using *diameter-bounded* **by** *blast*

lemma *diameter-compact-attained*:
assumes *compact s* *s* ≠ {}
shows ∃ *x*∈*s*. ∃ *y*∈*s*. (*norm*(*x* − *y*) = *diameter s*)
proof−
have *b*:*bounded s* **using** *assms*(*1*) *compact-eq-bounded-closed* **by** *auto*
then obtain *x y* **where** *xys*:*x*∈*s y*∈*s* **and** *xy*:∀ *u*∈*s*. ∀ *v*∈*s*. *norm* (*u* − *v*) ≤ *norm* (*x* − *y*) **using** *compact-sup-maxdistance*[*OF assms*] **by** *auto*
hence *diameter s* ≤ *norm* (*x* − *y*) **using** *rsup-le*[*of* {*norm* (*x* − *y*) |*x y*. *x* ∈ *s* ∧ *y* ∈ *s*} *norm* (*x* − *y*)]
unfolding *setle-def* **and** *diameter-def* **by** *auto*
thus *?thesis* **using** *diameter-bounded*(*1*)[*OF b*, *THEN bspec*[**where** *x*=*x*], *THEN bspec*[**where** *x*=*y*], *OF xys*] **and** *xys* **by** *auto*
qed

Related results with closure as the conclusion.

lemma *closed-scaling*:
assumes *closed s* **shows** *closed* ((λ*x*. *c* ∗*s x*) ‘ *s*)
proof(*cases s*={})
case *True* **thus** *?thesis* **by** *auto*
next
case *False*
show *?thesis*
proof(*cases c*=*0*)
have ∗:(λ*x*. *0*) ‘ *s* = {*0*} **using** ‹*s*≠{}› **by** *auto*
case *True* **thus** *?thesis* **apply** *auto* **unfolding** ∗ **using** *closed-sing* **by** *auto*
next
case *False*

```
  { fix x l assume as:∀ n::nat. x n ∈ op *s c ' s  (x ---> l) sequentially
    { fix n::nat have (1 / c) *s x n ∈ s using as(1)[THEN spec[where x=n]]
using ‹c≠0› by (auto simp add: vector-smult-assoc) }
    moreover
    { fix e::real assume e>0
      hence 0 < e *|c|  using ‹c≠0› mult-pos-pos[of e abs c] by auto
      then obtain N where ∀ n≥N. dist (x n) l < e * |c| using as(2)[unfolded
Lim-sequentially, THEN spec[where x=e * abs c]] by auto
       hence ∃ N. ∀ n≥N. dist ((1 / c) *s x n) ((1 / c) *s l) < e unfolding
dist-def unfolding vector-ssub-ldistrib[THEN sym] norm-mul
        using mult-imp-div-pos-less[of abs c - e] ‹c≠0› by auto  }
     hence ((λn. (1 / c) *s x n) ---> (1 / c) *s l) sequentially unfolding
Lim-sequentially by auto
   ultimately have l ∈ op *s c ' s using assms[unfolded closed-sequential-limits,
THEN spec[where x=λn. (1/c) *s x n], THEN spec[where x=(1/c) *s l]]
      unfolding image-iff using ‹c≠0› apply(rule-tac x=(1 / c) *s l in bexI)
apply auto unfolding vector-smult-assoc by auto }
  thus ?thesis unfolding closed-sequential-limits by auto
 qed
qed

lemma closed-negations:
 assumes closed s shows closed ((λx. -x) ' s)
 using closed-scaling[OF assms, of -1] unfolding pth-3 by auto

lemma compact-closed-sums:
 assumes compact s  closed t  shows closed {x + y | x y. x ∈ s ∧ y ∈ t}
proof-
 let ?S = {x + y |x y. x ∈ s ∧ y ∈ t}
 { fix x l assume as:∀ n. x n ∈ ?S  (x ---> l) sequentially
   from as(1) obtain f where f:∀ n. x n = fst (f n) + snd (f n)  ∀ n. fst (f n)
∈ s  ∀ n. snd (f n) ∈ t
     using choice[of λn y. x n = (fst y) + (snd y) ∧ fst y ∈ s ∧ snd y ∈ t] by
auto
   obtain l' r where l'∈s and r:∀ m n. m < n ⟶ r m < r n and lr:(((λn. fst
(f n)) ∘ r) ---> l') sequentially
     using assms(1)[unfolded compact-def, THEN spec[where x=λ n. fst (f n)]]
using f(2) by auto
   have ((λn. snd (f (r n))) ---> l - l') sequentially
    using Lim-sub[OF lim-subsequence[OF r as(2)] lr] and f(1) unfolding o-def
by auto
   hence l - l' ∈ t
     using assms(2)[unfolded closed-sequential-limits, THEN spec[where x=λ n.
snd (f (r n))], THEN spec[where x=l - l']]
     using f(3) by auto
    hence l ∈ ?S using ‹l' ∈ s› apply auto apply(rule-tac x=l' in exI) ap-
ply(rule-tac x=l - l' in exI) by auto
 }
 thus ?thesis unfolding closed-sequential-limits by auto
```

qed

lemma *closed-compact-sums*:
assumes *closed s compact t*
shows *closed* $\{x + y \mid x\ y.\ x \in s \wedge y \in t\}$
proof−
have $\{x + y \mid x\ y.\ x \in t \wedge y \in s\} = \{x + y \mid x\ y.\ x \in s \wedge y \in t\}$ **apply** *auto*
apply(*rule-tac x*=*y* **in** *exI*) **apply** *auto* **apply**(*rule-tac x*=*y* **in** *exI*) **by** *auto*
thus *?thesis* **using** *compact-closed-sums*[*OF assms*(*2*,*1*)] **by** *simp*
qed

lemma *compact-closed-differences*:
assumes *compact s closed t*
shows *closed* $\{x - y \mid x\ y.\ x \in s \wedge y \in t\}$
proof−
have $\{x + y \mid x\ y.\ x \in s \wedge y \in uminus\ `\ t\} = \{x - y \mid x\ y.\ x \in s \wedge y \in t\}$
apply *auto* **apply**(*rule-tac x*=*xa* **in** *exI*) **apply** *auto* **apply**(*rule-tac x*=*xa* **in** *exI*) **by** *auto*
thus *?thesis* **using** *compact-closed-sums*[*OF assms*(*1*) *closed-negations*[*OF assms*(*2*)]]
by *auto*
qed

lemma *closed-compact-differences*:
assumes *closed s compact t*
shows *closed* $\{x - y \mid x\ y.\ x \in s \wedge y \in t\}$
proof−
have $\{x + y \mid x\ y.\ x \in s \wedge y \in uminus\ `\ t\} = \{x - y \mid x\ y.\ x \in s \wedge y \in t\}$
apply *auto* **apply**(*rule-tac x*=*xa* **in** *exI*) **apply** *auto* **apply**(*rule-tac x*=*xa* **in** *exI*) **by** *auto*
thus *?thesis* **using** *closed-compact-sums*[*OF assms*(*1*) *compact-negations*[*OF assms*(*2*)]]
by *simp*
qed

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

lemma *translation-UNIV*:
$range\ (\lambda x{::}real\hat{}'a.\ a + x) = UNIV$
apply (*auto simp add*: *image-iff*) **apply**(*rule-tac x*=*x* − *a* **in** *exI*) **by** *auto*

lemma *translation-diff*: $(\lambda x{::}real\hat{}'a.\ a + x)\ `\ (s - t) = ((\lambda x.\ a + x)\ `\ s) - ((\lambda x.\ a + x)\ `\ t)$ **by** *auto*

lemma *closure-translation*:
$closure\ ((\lambda x.\ a + x)\ `\ s) = (\lambda x.\ a + x)\ `\ (closure\ s)$

proof−
have ∗:*op + a ' (UNIV − s) = UNIV − op + a ' s* **apply** *auto* **unfolding** *image-iff* **apply**(*rule-tac x=x − a* **in** *bexI*) **by** *auto*
show *?thesis* **unfolding** *closure-interior translation-diff translation-UNIV* **using** *interior-translation*[*of a UNIV − s*] **unfolding** ∗ **by** *auto*
qed

lemma *frontier-translation*:
frontier((λ*x. a + x*) ' *s*) = (λ*x. a + x*) ' (*frontier s*)
unfolding *frontier-def translation-diff interior-translation closure-translation* **by** *auto*

64.35 Separation between points and sets.

lemma *separate-point-closed*:
closed s ⟹ *a* ∉ *s* ==> (∃ *d*>*0*. ∀ *x*∈*s*. *d* ≤ *dist a x*)
proof(*cases s* = {})
case *True*
thus *?thesis* **by**(*auto intro*!: *exI*[**where** *x=1*])
next
case *False*
assume *closed s a* ∉ *s*
then obtain *x* **where** *x*∈*s* ∀ *y*∈*s*. *dist a x* ≤ *dist a y* **using** ‹*s* ≠ {}› *distance-attains-inf* [*of s a*] **by** *blast*
with ‹*x*∈*s*› **show** *?thesis* **using** *dist-pos-lt*[*of a x*] **and**‹*a* ∉ *s*› **by** *blast*
qed

lemma *separate-compact-closed*:
assumes *compact s* **and** *closed t* **and** *s* ∩ *t* = {}
shows ∃ *d*>*0*. ∀ *x*∈*s*. ∀ *y*∈*t*. *d* ≤ *dist x y*
proof−
have *0* ∉ {*x − y* |*x y*. *x* ∈ *s* ∧ *y* ∈ *t*} **using** *assms*(*3*) **by** *auto*
then obtain *d* **where** *d*>*0* **and** *d*:∀ *x*∈{*x − y* |*x y*. *x* ∈ *s* ∧ *y* ∈ *t*}. *d* ≤ *dist 0 x*
using *separate-point-closed*[*OF compact-closed-differences*[*OF assms*(*1*,*2*)], *of 0*] **by** *auto*
{ fix *x y* **assume** *x*∈*s y*∈*t*
hence *x − y* ∈ {*x − y* |*x y*. *x* ∈ *s* ∧ *y* ∈ *t*} **by** *auto*
hence *d* ≤ *dist* (*x − y*) *0* **using** *d*[*THEN bspec*[**where** *x=x − y*]] **using** *dist-sym*
by (*auto simp add*: *dist-sym*)
hence *d* ≤ *dist x y* **unfolding** *dist-def* **by** *auto* **}**
thus *?thesis* **using** ‹*d*>*0*› **by** *auto*
qed

lemma *separate-closed-compact*:
assumes *closed s* **and** *compact t* **and** *s* ∩ *t* = {}
shows ∃ *d*>*0*. ∀ *x*∈*s*. ∀ *y*∈*t*. *d* ≤ *dist x y*
proof−

have $*{:}t \cap s = \{\}$ **using** *assms*(*3*) **by** *auto*
show *?thesis* **using** *separate-compact-closed*[*OF assms*(*2*,*1*) $*$]
apply *auto* **apply**(*rule-tac x*=*d* **in** *exI*) **apply** *auto* **apply** (*erule-tac x*=*y* **in** *ballE*)
by (*auto simp add*: *dist-sym*)
qed

lemma *interval*: **fixes** $a :: 'a{::}ord\hat{}'n{::}finite$ **shows**
$\{a <..< b\} = \{x{::}'a\hat{}'n.\ \forall i.\ a\$i < x\$i \wedge x\$i < b\$i\}$ **and**
$\{a\ ..\ b\} = \{x{::}'a\hat{}'n.\ \forall i.\ a\$i \leq x\$i \wedge x\$i \leq b\$i\}$
by (*auto simp add*: *expand-set-eq vector-less-def vector-less-eq-def*)

lemma *mem-interval*: **fixes** $a :: 'a{::}ord\hat{}'n{::}finite$ **shows**
$x \in \{a<..<b\} \longleftrightarrow (\forall i.\ a\$i < x\$i \wedge x\$i < b\$i)$
$x \in \{a\ ..\ b\} \longleftrightarrow (\forall i.\ a\$i \leq x\$i \wedge x\$i \leq b\$i)$
using *interval*[*of a b*]
by(*auto simp add*: *expand-set-eq vector-less-def vector-less-eq-def*)

lemma *interval-eq-empty*: **fixes** $a :: real\hat{}'n{::}finite$ **shows**
$(\{a <..< b\} = \{\} \longleftrightarrow (\exists i.\ b\$i \leq a\$i))$ (**is** *?th1*) **and**
$(\{a\ ..\ b\} = \{\} \longleftrightarrow (\exists i.\ b\$i < a\$i))$ (**is** *?th2*)
proof−
{ fix *i x* **assume** *as*:$b\$i \leq a\$i$ **and** *x*:$x \in \{a <..< b\}$
hence $a\ \$\ i < x\ \$\ i \wedge x\ \$\ i < b\ \$\ i$ **unfolding** *mem-interval* **by** *auto*
hence $a\$i < b\$i$ **by** *auto*
hence *False* **using** *as* **by** *auto* **}**
moreover
{ assume *as*:$\forall i.\ \neg\ (b\$i \leq a\$i)$
let *?x* = $(1/2) *s (a + b)$
{ fix *i*
have $a\$i < b\$i$ **using** *as*[*THEN spec*[**where** *x*=*i*]] **by** *auto*
hence $a\$i < ((1/2) *s (a+b))\ \$\ i\ ((1/2) *s (a+b))\ \$\ i < b\$i$
unfolding *vector-smult-component* **and** *vector-add-component*
by (*auto simp add*: *less-divide-eq-number-of1*) **}**
hence $\{a <..< b\} \neq \{\}$ **using** *mem-interval*(*1*)[*of ?x a b*] **by** *auto* **}**
ultimately show *?th1* **by** *blast*

{ fix *i x* **assume** *as*:$b\$i < a\$i$ **and** *x*:$x \in \{a\ ..\ b\}$
hence $a\ \$\ i \leq x\ \$\ i \wedge x\ \$\ i \leq b\ \$\ i$ **unfolding** *mem-interval* **by** *auto*
hence $a\$i \leq b\$i$ **by** *auto*
hence *False* **using** *as* **by** *auto* **}**
moreover
{ assume *as*:$\forall i.\ \neg\ (b\$i < a\$i)$
let *?x* = $(1/2) *s (a + b)$
{ fix *i*
have $a\$i \leq b\$i$ **using** *as*[*THEN spec*[**where** *x*=*i*]] **by** *auto*
hence $a\$i \leq ((1/2) *s (a+b))\ \$\ i\ ((1/2) *s (a+b))\ \$\ i \leq b\$i$

unfolding *vector-smult-component* **and** *vector-add-component*
by (*auto simp add*: *less-divide-eq-number-of1*) **}**
hence $\{a\ ..\ b\} \neq \{\}$ **using** *mem-interval*(*2*)[*of ?x a b*] **by** *auto* **}**
ultimately show *?th2* **by** *blast*
qed

lemma *interval-ne-empty*: **fixes** *a* :: *real^'n*::*finite* **shows**
$\{a\ ..\ b\} \neq \{\} \longleftrightarrow (\forall i.\ a\$i \leq b\$i)$ **and**
$\{a\ <..<\ b\} \neq \{\} \longleftrightarrow (\forall i.\ a\$i < b\$i)$
unfolding *interval-eq-empty*[*of a b*] **by** (*auto simp add*: *not-less not-le*)

lemma *subset-interval-imp*: **fixes** *a* :: *real^'n*::*finite* **shows**
$(\forall i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i) \Longrightarrow \{c\ ..\ d\} \subseteq \{a\ ..\ b\}$ **and**
$(\forall i.\ a\$i < c\$i \wedge d\$i < b\$i) \Longrightarrow \{c\ ..\ d\} \subseteq \{a<..<b\}$ **and**
$(\forall i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i) \Longrightarrow \{c<..<d\} \subseteq \{a\ ..\ b\}$ **and**
$(\forall i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i) \Longrightarrow \{c<..<d\} \subseteq \{a<..<b\}$
unfolding *subset-eq*[*unfolded Ball-def*] **unfolding** *mem-interval*
by (*auto intro*: *order-trans less-le-trans le-less-trans less-imp-le*)

lemma *interval-sing*: **fixes** *a* :: *'a*::*linorder^'n*::*finite* **shows**
$\{a\ ..\ a\} = \{a\} \wedge \{a<..<a\} = \{\}$
apply(*auto simp add*: *expand-set-eq vector-less-def vector-less-eq-def Cart-eq*)
apply (*simp add*: *order-eq-iff*)
apply (*auto simp add*: *not-less less-imp-le*)
done

lemma *interval-open-subset-closed*: **fixes** *a* :: *'a*::*preorder^'n*::*finite* **shows**
$\{a<..<b\} \subseteq \{a\ ..\ b\}$
proof(*simp add*: *subset-eq*, *rule*)
fix *x*
assume *x*:$x \in \{a<..<b\}$
{ fix *i*
have $a\ \$\ i \leq x\ \$\ i$
using *x order-less-imp-le*[*of a*$*i x*$*i*]
by(*simp add*: *expand-set-eq vector-less-def vector-less-eq-def Cart-eq*)
}
moreover
{ fix *i*
have $x\ \$\ i \leq b\ \$\ i$
using *x order-less-imp-le*[*of x*$*i b*$*i*]
by(*simp add*: *expand-set-eq vector-less-def vector-less-eq-def Cart-eq*)
}
ultimately
show $a \leq x \wedge x \leq b$
by(*simp add*: *expand-set-eq vector-less-def vector-less-eq-def Cart-eq*)
qed

lemma *subset-interval*: **fixes** *a* :: *real^'n*::*finite* **shows**

$\{c\ ..\ d\} \subseteq \{a\ ..\ b\} \longleftrightarrow (\forall i.\ c\$i \leq d\$i) \longrightarrow (\forall i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i)$ (**is** *?th1*) **and**
$\{c\ ..\ d\} \subseteq \{a<..<b\} \longleftrightarrow (\forall i.\ c\$i \leq d\$i) \longrightarrow (\forall i.\ a\$i < c\$i \wedge d\$i < b\$i)$ (**is** *?th2*) **and**
$\{c<..<d\} \subseteq \{a\ ..\ b\} \longleftrightarrow (\forall i.\ c\$i < d\$i) \longrightarrow (\forall i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i)$ (**is** *?th3*) **and**
$\{c<..<d\} \subseteq \{a<..<b\} \longleftrightarrow (\forall i.\ c\$i < d\$i) \longrightarrow (\forall i.\ a\$i \leq c\$i \wedge d\$i \leq b\$i)$ (**is** *?th4*)
proof−
show *?th1* **unfolding** *subset-eq* **and** *Ball-def* **and** *mem-interval* **by** (*auto intro*: *order-trans*)
show *?th2* **unfolding** *subset-eq* **and** *Ball-def* **and** *mem-interval* **by** (*auto intro*: *le-less-trans less-le-trans order-trans less-imp-le*)
{ assume *as*: $\{c<..<d\} \subseteq \{a\ ..\ b\}$ $\forall i.\ c\$i < d\$i$
hence $\{c<..<d\} \neq \{\}$ **unfolding** *interval-eq-empty* **by** (*auto*, *drule-tac x=i* **in** *spec*, *simp*)
fix *i*

{ let *?x* = $(\chi\ j.\ (\mathit{if}\ j{=}i\ \mathit{then}\ ((\mathit{min}\ (a\$j)\ (d\$j)){+}c\$j)/2\ \mathit{else}\ (c\$j{+}d\$j)/2))$::*real^'n*
assume *as2*: $a\$i > c\$i$
{ fix *j*
have $c\ \$\ j < ?x\ \$\ j \wedge ?x\ \$\ j < d\ \$\ j$ **unfolding** *Cart-lambda-beta*
apply(*cases j=i*) **using** *as*(*2*)[*THEN spec*[**where** *x=j*]]
by (*auto simp add*: *less-divide-eq-number-of1 as2*) **}**
hence *?x*∈{*c*<..<*d*} **unfolding** *mem-interval* **by** *auto*
moreover
have *?x*∉{*a* .. *b*}
unfolding *mem-interval* **apply** *auto* **apply**(*rule-tac x=i* **in** *exI*)
using *as*(*2*)[*THEN spec*[**where** *x=i*]] **and** *as2*
by (*auto simp add*: *less-divide-eq-number-of1*)
ultimately have *False* **using** *as* **by** *auto* **}**
hence $a\$i \leq c\$i$ **by**(*rule ccontr*)*auto*
moreover
{ let *?x* = $(\chi\ j.\ (\mathit{if}\ j{=}i\ \mathit{then}\ ((\mathit{max}\ (b\$j)\ (c\$j)){+}d\$j)/2\ \mathit{else}\ (c\$j{+}d\$j)/2))$::*real^'n*
assume *as2*: $b\$i < d\$i$
{ fix *j*
have $d\ \$\ j > ?x\ \$\ j \wedge ?x\ \$\ j > c\ \$\ j$ **unfolding** *Cart-lambda-beta*
apply(*cases j=i*) **using** *as*(*2*)[*THEN spec*[**where** *x=j*]]
by (*auto simp add*: *less-divide-eq-number-of1 as2*) **}**
hence *?x*∈{*c*<..<*d*} **unfolding** *mem-interval* **by** *auto*
moreover
have *?x*∉{*a* .. *b*}
unfolding *mem-interval* **apply** *auto* **apply**(*rule-tac x=i* **in** *exI*)
using *as*(*2*)[*THEN spec*[**where** *x=i*]] **and** *as2*
by (*auto simp add*: *less-divide-eq-number-of1*)
ultimately have *False* **using** *as* **by** *auto* **}**
hence $b\$i \geq d\$i$ **by**(*rule ccontr*)*auto*
ultimately
have $a\$i \leq c\$i \wedge d\$i \leq b\$i$ **by** *auto*

} **note** *part1 = this*
thus *?th3* **unfolding** *subset-eq* **and** *Ball-def* **and** *mem-interval* **apply** *auto* **apply** (*erule-tac x=ia* **in** *allE*, *simp*)+ **by** (*erule-tac x=i* **in** *allE*, *erule-tac x=i* **in** *allE*, *simp*)+
{ **assume** *as*:$\{c<..<d\} \subseteq \{a<..<b\}$ $\forall i.\ c\$i < d\$i$
fix *i*
from *as*(*1*) **have** $\{c<..<d\} \subseteq \{a..b\}$ **using** *interval-open-subset-closed*[*of a b*] **by** *auto*
hence $a\$i \leq c\$i \wedge d\$i \leq b\$i$ **using** *part1* **and** *as*(*2*) **by** *auto* **}** **note** $* =$ *this*
thus *?th4* **unfolding** *subset-eq* **and** *Ball-def* **and** *mem-interval* **apply** *auto* **apply** (*erule-tac x=ia* **in** *allE*, *simp*)+ **by** (*erule-tac x=i* **in** *allE*, *erule-tac x=i* **in** *allE*, *simp*)+
qed

lemma *disjoint-interval*: **fixes** *a*::*real^'n*::*finite* **shows**
$\{a\ ..\ b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow (\exists i.\ (b\$i < a\$i \vee d\$i < c\$i \vee b\$i < c\$i \vee d\$i < a\$i))$ (**is** *?th1*) **and**
$\{a\ ..\ b\} \cap \{c<..<d\} = \{\} \longleftrightarrow (\exists i.\ (b\$i < a\$i \vee d\$i \leq c\$i \vee b\$i \leq c\$i \vee d\$i \leq a\$i))$ (**is** *?th2*) **and**
$\{a<..<b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow (\exists i.\ (b\$i \leq a\$i \vee d\$i < c\$i \vee b\$i \leq c\$i \vee d\$i \leq a\$i))$ (**is** *?th3*) **and**
$\{a<..<b\} \cap \{c<..<d\} = \{\} \longleftrightarrow (\exists i.\ (b\$i \leq a\$i \vee d\$i \leq c\$i \vee b\$i \leq c\$i \vee d\$i \leq a\$i))$ (**is** *?th4*)
proof−
let *?z* = $(\chi\ i.\ ((max\ (a\$i)\ (c\$i)) + (min\ (b\$i)\ (d\$i))) / 2)$::*real^'n*
show *?th1 ?th2 ?th3 ?th4*
unfolding *expand-set-eq* **and** *Int-iff* **and** *empty-iff* **and** *mem-interval* **and** *all-conj-distrib*[*THEN sym*] **and** *eq-False*
apply (*auto elim*!: *allE*[**where** *x=?z*])
apply ((*rule-tac x=x* **in** *exI*, *force*) | (*rule-tac x=i* **in** *exI*, *force*))+
done
qed

lemma *inter-interval*: **fixes** *a* :: *'a*::*linorder^'n*::*finite* **shows**
$\{a\ ..\ b\} \cap \{c\ ..\ d\} = \{(\chi\ i.\ max\ (a\$i)\ (c\$i))\ ..\ (\chi\ i.\ min\ (b\$i)\ (d\$i))\}$
unfolding *expand-set-eq* **and** *Int-iff* **and** *mem-interval*
by (*auto simp add*: *less-divide-eq-number-of1 intro*!: *bexI*)

lemma *open-interval-lemma*: **fixes** *x* :: *real* **shows**
$a < x \Longrightarrow x < b ==> (\exists d>0.\ \forall x'.\ abs(x' - x) < d --> a < x' \wedge x' < b)$
by(*rule-tac x=min* $(x - a)$ $(b - x)$ **in** *exI*, *auto*)

lemma *open-interval*: **fixes** *a* :: *real^'n*::*finite* **shows** *open* $\{a<..<b\}$
proof−
{ **fix** *x* **assume** *x*:$x \in \{a<..<b\}$
{ **fix** *i*

```
    have ∃ d>0. ∀ x'. abs (x' − (x$i)) < d ⟶ a$i < x' ∧ x' < b$i
      using x[unfolded mem-interval, THEN spec[where x=i]]
      using open-interval-lemma[of a$i x$i b$i] by auto }

  hence ∀ i. ∃ d>0. ∀ x'. abs (x' − (x$i)) < d ⟶ a$i < x' ∧ x' < b$i by auto
  then obtain d where d:∀ i. 0 < d i ∧ (∀ x'. |x' − x $ i| < d i ⟶ a $ i <
x' ∧ x' < b $ i)
    using bchoice[of UNIV λi d. d>0 ∧ (∀ x'. |x' − x $ i| < d ⟶ a $ i < x'
∧ x' < b $ i)] by auto

  let ?d = Min (range d)
  have **:finite (range d) range d ≠ {} by auto
  have ?d>0 unfolding Min-gr-iff[OF **] using d by auto
  moreover
  { fix x' assume as:dist x' x < ?d
    { fix i
      have |x'$i − x $ i| < d i
        using norm-bound-component-lt[OF as[unfolded dist-def], of i]
        unfolding vector-minus-component and Min-gr-iff[OF **] by auto
      hence a $ i < x' $ i x' $ i < b $ i using d[THEN spec[where x=i]] by
auto }
    hence a < x' ∧ x' < b unfolding vector-less-def by auto }
  ultimately have ∃ e>0. ∀ x'. dist x' x < e ⟶ x' ∈ {a<..<b} by (auto,
rule-tac x=?d in exI, simp)
 }
 thus ?thesis unfolding open-def using open-interval-lemma by auto
qed

lemma closed-interval: fixes a :: real^'n::finite shows closed {a .. b}
proof−
 { fix x i assume as:∀ e>0. ∃ x'∈{a..b}. x' ≠ x ∧ dist x' x < e
   { assume xa:a$i > x$i
    with as obtain y where y:y∈{a..b} y ≠ x dist y x < a$i − x$i by(erule-tac
x=a$i − x$i in allE)auto
     hence False unfolding mem-interval and dist-def
       using component-le-norm[of y−x i, unfolded vector-minus-component] and
xa by(auto elim!: allE[where x=i])
   } hence a$i ≤ x$i by(rule ccontr)auto
   moreover
   { assume xb:b$i < x$i
    with as obtain y where y:y∈{a..b} y ≠ x dist y x < x$i − b$i by(erule-tac
x=x$i − b$i in allE)auto
     hence False unfolding mem-interval and dist-def
       using component-le-norm[of y−x i, unfolded vector-minus-component] and
xb by(auto elim!: allE[where x=i])
   } hence x$i ≤ b$i by(rule ccontr)auto
   ultimately
   have a $ i ≤ x $ i ∧ x $ i ≤ b $ i by auto }
 thus ?thesis unfolding closed-limpt islimpt-approachable mem-interval by auto
```

qed

lemma *interior-closed-interval*: **fixes** *a* :: *real^'n*::*finite* **shows**
interior {*a* .. *b*} = {*a*<..<*b*} (**is** *?L* = *?R*)
proof(*rule subset-antisym*)
show *?R* ⊆ *?L* **using** *interior-maximal*[*OF interval-open-subset-closed open-interval*]
by *auto*
next
{ fix *x* **assume** ∃ *T*. *open T* ∧ *x* ∈ *T* ∧ *T* ⊆ {*a*..*b*}
then obtain *s* **where** *s*:*open s x* ∈ *s s* ⊆ {*a*..*b*} **by** *auto*
then obtain *e* **where** *e*>*0* **and** *e*:∀ *x'*. *dist x' x* < *e* ⟶ *x'* ∈ {*a*..*b*} **unfolding** *open-def* **and** *subset-eq* **by** *auto*
{ fix *i*
have *dist* (*x* − (*e* / *2*) ∗*s basis i*) *x* < *e*
dist (*x* + (*e* / *2*) ∗*s basis i*) *x* < *e*
unfolding *dist-def* **apply** *auto*
unfolding *norm-minus-cancel* **and** *norm-mul* **using** *norm-basis*[*of i*] **and** ‹*e*>*0*› **by** *auto*
hence *a* $ *i* ≤ (*x* − (*e* / *2*) ∗*s basis i*) $ *i*
(*x* + (*e* / *2*) ∗*s basis i*) $ *i* ≤ *b* $ *i*
using *e*[*THEN spec*[**where** *x*=*x* − (*e*/*2*) ∗*s basis i*]]
and *e*[*THEN spec*[**where** *x*=*x* + (*e*/*2*) ∗*s basis i*]]
unfolding *mem-interval* **by** (*auto elim*!: *allE*[**where** *x*=*i*])
hence *a* $ *i* < *x* $ *i* **and** *x* $ *i* < *b* $ *i*
unfolding *vector-minus-component* **and** *vector-add-component*
unfolding *vector-smult-component* **and** *basis-component* **using** ‹*e*>*0*› **by** *auto* **}**
hence *x* ∈ {*a*<..<*b*} **unfolding** *mem-interval* **by** *auto* **}**
thus *?L* ⊆ *?R* **unfolding** *interior-def* **and** *subset-eq* **by** *auto*
qed

lemma *bounded-closed-interval*: **fixes** *a* :: *real^'n*::*finite* **shows**
bounded {*a* .. *b*}
proof−
let *?b* = ∑ *i*∈*UNIV*. |*a*$*i*| + |*b*$*i*|
{ fix *x*::*real^'n* **assume** *x*:∀ *i*. *a* $ *i* ≤ *x* $ *i* ∧ *x* $ *i* ≤ *b* $ *i*
{ fix *i*
have |*x*$*i*| ≤ |*a*$*i*| + |*b*$*i*| **using** *x*[*THEN spec*[**where** *x*=*i*]] **by** *auto* **}**
hence (∑ *i*∈*UNIV*. |*x* $ *i*|) ≤ *?b* **by**(*rule setsum-mono*)
hence *norm x* ≤ *?b* **using** *norm-le-l1*[*of x*] **by** *auto* **}**
thus *?thesis* **unfolding** *interval* **and** *bounded-def* **by** *auto*
qed

lemma *bounded-interval*: **fixes** *a* :: *real^'n*::*finite* **shows**
bounded {*a* .. *b*} ∧ *bounded* {*a*<..<*b*}
using *bounded-closed-interval*[*of a b*]
using *interval-open-subset-closed*[*of a b*]
using *bounded-subset*[*of* {*a*..*b*} {*a*<..<*b*}]
by *simp*

lemma *not-interval-univ*: **fixes** $a :: real\hat{}'n::finite$ **shows**
$(\{a\ ..\ b\} \neq UNIV) \wedge (\{a<..<b\} \neq UNIV)$
using *bounded-interval*[*of a b*]
by *auto*

lemma *compact-interval*: **fixes** $a :: real\hat{}'n::finite$ **shows**
compact $\{a\ ..\ b\}$
using *bounded-closed-imp-compact* **using** *bounded-interval*[*of a b*] **using** *closed-interval*[*of a b*] **by** *auto*

lemma *open-interval-midpoint*: **fixes** $a :: real\hat{}'n::finite$
assumes $\{a<..<b\} \neq \{\}$ **shows** $((1/2) *s\ (a + b)) \in \{a<..<b\}$
proof−
{ fix *i*
have $a\ \$\ i < ((1\ /\ 2) *s\ (a + b))\ \$\ i \wedge ((1\ /\ 2) *s\ (a + b))\ \$\ i < b\ \$\ i$
using *assms*[*unfolded interval-ne-empty*, *THEN spec*[**where** *x*=*i*]]
unfolding *vector-smult-component* **and** *vector-add-component*
by(*auto simp add*: *less-divide-eq-number-of1*) **}**
thus *?thesis* **unfolding** *mem-interval* **by** *auto*
qed

lemma *open-closed-interval-convex*: **fixes** $x :: real\hat{}'n::finite$
assumes $x{:}x \in \{a<..<b\}$ **and** $y{:}y \in \{a\ ..\ b\}$ **and** $e{:}0 < e\ e \leq 1$
shows $(e *s\ x + (1 - e) *s\ y) \in \{a<..<b\}$
proof−
{ fix *i*
have $a\ \$\ i = e * a\$i + (1 - e) * a\$i$ **unfolding** *left-diff-distrib* **by** *simp*
also have $\ldots < e * x\ \$\ i + (1 - e) * y\ \$\ i$ **apply**(*rule add-less-le-mono*)
using *e* **unfolding** *mult-less-cancel-left* **and** *mult-le-cancel-left* **apply** *simp-all*
using *x* **unfolding** *mem-interval* **apply** *simp*
using *y* **unfolding** *mem-interval* **apply** *simp*
done
finally have $a\ \$\ i < (e *s\ x + (1 - e) *s\ y)\ \$\ i$ **by** *auto*
moreover {
have $b\ \$\ i = e * b\$i + (1 - e) * b\$i$ **unfolding** *left-diff-distrib* **by** *simp*
also have $\ldots > e * x\ \$\ i + (1 - e) * y\ \$\ i$ **apply**(*rule add-less-le-mono*)
using *e* **unfolding** *mult-less-cancel-left* **and** *mult-le-cancel-left* **apply** *simp-all*
using *x* **unfolding** *mem-interval* **apply** *simp*
using *y* **unfolding** *mem-interval* **apply** *simp*
done
finally have $(e *s\ x + (1 - e) *s\ y)\ \$\ i < b\ \$\ i$ **by** *auto*
} ultimately have $a\ \$\ i < (e *s\ x + (1 - e) *s\ y)\ \$\ i \wedge (e *s\ x + (1 - e) *s\ y)\ \$\ i < b\ \$\ i$ **by** *auto* **}**
thus *?thesis* **unfolding** *mem-interval* **by** *auto*
qed

lemma *closure-open-interval*: **fixes** $a :: real\hat{}'n::finite$
assumes $\{a<..<b\} \neq \{\}$

shows *closure* $\{a<..<b\} = \{a\ ..\ b\}$
proof−
have *ab*:$a < b$ **using** *assms*[*unfolded interval-ne-empty*] **unfolding** *vector-less-def*
by *auto*
let *?c* = $(1\ /\ 2) *s (a + b)$
{ fix x **assume** *as*:$x \in \{a\ ..\ b\}$
def $f == \lambda n{::}nat.\ x + (inverse\ (real\ n + 1)) *s (?c - x)$
{ fix n **assume** *fn*:$f\ n < b \longrightarrow a < f\ n \longrightarrow f\ n = x$ **and** *xc*:$x \neq ?c$
have $*{:}0 < inverse\ (real\ n + 1)\ inverse\ (real\ n + 1) \leq 1$ **unfolding**
inverse-le-1-iff **by** *auto*
have $inverse\ (real\ n + 1) *s (1\ /\ 2) *s (a + b) + (1 - inverse\ (real\ n +$
$1)) *s x =$
$x + inverse\ (real\ n + 1) *s ((1\ /\ 2) *s (a + b) - x)$ **by** (*auto simp add*:
vector-ssub-ldistrib vector-add-ldistrib field-simps vector-sadd-rdistrib[*THEN sym*])
hence $f\ n < b$ **and** $a < f\ n$ **using** *open-closed-interval-convex*[*OF open-interval-midpoint*[*OF*
assms] *as* $*$] **unfolding** *f-def* **by** *auto*
hence *False* **using** *fn* **unfolding** *f-def* **using** *xc* **by**(*auto simp add*: *vector-mul-lcancel*
vector-ssub-ldistrib) **}**
moreover
{ assume $\neg\ (f\ ---{>}\ x)\ sequentially$
{ fix *e*::*real* **assume** *e*>*0*
hence $\exists N{::}nat.\ inverse\ (real\ (N + 1)) < e$ **using** *real-arch-inv*[*of e*] **apply**
(*auto simp add*: *Suc-pred'*) **apply**(*rule-tac x*=$n - 1$ **in** *exI*) **by** *auto*
then obtain $N{::}nat$ **where** $inverse\ (real\ (N + 1)) < e$ **by** *auto*
hence $\forall\ n{\geq}N.\ inverse\ (real\ n + 1) < e$ **by** (*auto*, *metis Suc-le-mono le-SucE*
less-imp-inverse-less nat-le-real-less order-less-trans real-of-nat-Suc real-of-nat-Suc-gt-zero)
hence $\exists N{::}nat.\ \forall\ n{\geq}N.\ inverse\ (real\ n + 1) < e$ **by** *auto* **}**
hence $((vec1 \circ (\lambda n.\ inverse\ (real\ n + 1)))\ ---{>}\ vec1\ 0)\ sequentially$
unfolding *Lim-sequentially* **by**(*auto simp add*: *dist-vec1*)
hence $(f\ ---{>}\ x)\ sequentially$ **unfolding** *f-def*
using *Lim-add*[*OF Lim-const*, *of* $\lambda n{::}nat.\ (inverse\ (real\ n + 1)) *s ((1\ /$
$2) *s (a + b) - x)\ 0\ sequentially\ x$]
using *Lim-vmul*[*of* $\lambda n{::}nat.\ inverse\ (real\ n + 1)\ 0\ sequentially\ ((1\ /\ 2) *s$
$(a + b) - x)$] **by** *auto* **}**
ultimately have $x \in closure\ \{a<..<b\}$
using *as* **and** *open-interval-midpoint*[*OF assms*] **unfolding** *closure-def* **un-**
folding *islimpt-sequential* **by**(*cases x*=*?c*)*auto* **}**
thus *?thesis* **using** *closure-minimal*[*OF interval-open-subset-closed closed-interval*,
of a b] **by** *blast*
qed

lemma *bounded-subset-open-interval-symmetric*: **fixes** *s*::($real\hat{}'n{::}finite$) *set*
assumes *bounded s* **shows** $\exists\ a.\ s \subseteq \{-a<..<a\}$
proof−
obtain *b* **where** *b*>*0* **and** *b*:$\forall\ x{\in}s.\ norm\ x \leq b$ **using** *assms*[*unfolded bounded-pos*]
by *auto*
def $a \equiv (\chi\ i.\ b{+}1){::}real\hat{}'n$
{ fix x **assume** $x{\in}s$
fix i

have $(-a)\$i < x\$i$ **and** $x\$i < a\$i$ **using** *b*[*THEN bspec*[**where** *x*=*x*], *OF* ‹*x*∈*s*›] **and** *component-le-norm*[*of x i*]
unfolding *vector-uminus-component* **and** *a-def* **and** *Cart-lambda-beta* **by** *auto*
}
thus *?thesis* **by**(*auto intro*: *exI*[**where** *x*=*a*] *simp add*: *vector-less-def*)
qed

lemma *bounded-subset-open-interval*:
bounded s ==> (∃ *a b*. *s* ⊆ {*a*<..<*b*})
by(*metis bounded-subset-open-interval-symmetric*)

lemma *bounded-subset-closed-interval-symmetric*:
assumes *bounded s* **shows** ∃ *a*. *s* ⊆ {−*a* .. *a*}
proof−
obtain *a* **where** *s* ⊆ {− *a*<..<*a*} **using** *bounded-subset-open-interval-symmetric*[*OF assms*] **by** *auto*
thus *?thesis* **using** *interval-open-subset-closed*[*of* −*a a*] **by** *auto*
qed

lemma *bounded-subset-closed-interval*:
bounded s ==> (∃ *a b*. *s* ⊆ {*a* .. *b*})
using *bounded-subset-closed-interval-symmetric*[*of s*] **by** *auto*

lemma *frontier-closed-interval*:
frontier {*a* .. *b*} = {*a* .. *b*} − {*a*<..<*b*}
unfolding *frontier-def* **unfolding** *interior-closed-interval* **and** *closure-closed*[*OF closed-interval*] **..**

lemma *frontier-open-interval*:
frontier {*a*<..<*b*} = (*if* {*a*<..<*b*} = {} *then* {} *else* {*a* .. *b*} − {*a*<..<*b*})
proof(*cases* {*a*<..<*b*} = {})
case *True* **thus** *?thesis* **using** *frontier-empty* **by** *auto*
next
case *False* **thus** *?thesis* **unfolding** *frontier-def* **and** *closure-open-interval*[*OF False*] **and** *interior-open*[*OF open-interval*] **by** *auto*
qed

lemma *inter-interval-mixed-eq-empty*: **fixes** *a* :: *real^'n*::*finite*
assumes {*c*<..<*d*} ≠ {} **shows** {*a*<..<*b*} ∩ {*c* .. *d*} = {} ⟷ {*a*<..<*b*} ∩ {*c*<..<*d*} = {}
unfolding *closure-open-interval*[*OF assms*, *THEN sym*] **unfolding** *open-inter-closure-eq-empty*[*OF open-interval*] **..**

lemma *all-1*: (∀ *x*::*1*. *P x*) ⟷ *P 1*
by (*metis num1-eq-iff*)

lemma *ex-1*: $(\exists\, x{::}1.\ P\ x) \longleftrightarrow P\ 1$
by *auto* (*metis num1-eq-iff*)

lemma *interval-cases-1*: **fixes** $x :: real\hat{}1$ **shows**
$x \in \{a\ ..\ b\} ==> x \in \{a<..<b\} \vee (x = a) \vee (x = b)$
by(*simp add*: *Cart-eq vector-less-def vector-less-eq-def all-1*, *auto*)

lemma *in-interval-1*: **fixes** $x :: real\hat{}1$ **shows**
$(x \in \{a\ ..\ b\} \longleftrightarrow dest\text{-}vec1\ a \leq dest\text{-}vec1\ x \wedge dest\text{-}vec1\ x \leq dest\text{-}vec1\ b) \wedge$
$(x \in \{a<..<b\} \longleftrightarrow dest\text{-}vec1\ a < dest\text{-}vec1\ x \wedge dest\text{-}vec1\ x < dest\text{-}vec1\ b)$
by(*simp add*: *Cart-eq vector-less-def vector-less-eq-def all-1 dest-vec1-def*)

lemma *interval-eq-empty-1*: **fixes** $a :: real\hat{}1$ **shows**
$\{a\ ..\ b\} = \{\} \longleftrightarrow dest\text{-}vec1\ b < dest\text{-}vec1\ a$
$\{a<..<b\} = \{\} \longleftrightarrow dest\text{-}vec1\ b \leq dest\text{-}vec1\ a$
unfolding *interval-eq-empty* **and** *ex-1* **and** *dest-vec1-def* **by** *auto*

lemma *subset-interval-1*: **fixes** $a :: real\hat{}1$ **shows**
$(\{a\ ..\ b\} \subseteq \{c\ ..\ d\} \longleftrightarrow dest\text{-}vec1\ b < dest\text{-}vec1\ a \vee$
$dest\text{-}vec1\ c \leq dest\text{-}vec1\ a \wedge dest\text{-}vec1\ a \leq dest\text{-}vec1\ b \wedge dest\text{-}vec1\ b \leq dest\text{-}vec1\ d)$
$(\{a\ ..\ b\} \subseteq \{c<..<d\} \longleftrightarrow dest\text{-}vec1\ b < dest\text{-}vec1\ a \vee$
$dest\text{-}vec1\ c < dest\text{-}vec1\ a \wedge dest\text{-}vec1\ a \leq dest\text{-}vec1\ b \wedge dest\text{-}vec1\ b < dest\text{-}vec1\ d)$
$(\{a<..<b\} \subseteq \{c\ ..\ d\} \longleftrightarrow dest\text{-}vec1\ b \leq dest\text{-}vec1\ a \vee$
$dest\text{-}vec1\ c \leq dest\text{-}vec1\ a \wedge dest\text{-}vec1\ a < dest\text{-}vec1\ b \wedge dest\text{-}vec1\ b \leq dest\text{-}vec1\ d)$
$(\{a<..<b\} \subseteq \{c<..<d\} \longleftrightarrow dest\text{-}vec1\ b \leq dest\text{-}vec1\ a \vee$
$dest\text{-}vec1\ c \leq dest\text{-}vec1\ a \wedge dest\text{-}vec1\ a < dest\text{-}vec1\ b \wedge dest\text{-}vec1\ b \leq dest\text{-}vec1\ d)$
unfolding *subset-interval*[*of a b c d*] **unfolding** *all-1* **and** *dest-vec1-def* **by** *auto*

lemma *eq-interval-1*: **fixes** $a :: real\hat{}1$ **shows**
$\{a\ ..\ b\} = \{c\ ..\ d\} \longleftrightarrow$
$dest\text{-}vec1\ b < dest\text{-}vec1\ a \wedge dest\text{-}vec1\ d < dest\text{-}vec1\ c \vee$
$dest\text{-}vec1\ a = dest\text{-}vec1\ c \wedge dest\text{-}vec1\ b = dest\text{-}vec1\ d$
using *set-eq-subset*[*of* $\{a\ ..\ b\}$ $\{c\ ..\ d\}$]
using *subset-interval-1*(*1*)[*of a b c d*]
using *subset-interval-1*(*1*)[*of c d a b*]
by *auto*

lemma *disjoint-interval-1*: **fixes** $a :: real\hat{}1$ **shows**
$\{a\ ..\ b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow dest\text{-}vec1\ b < dest\text{-}vec1\ a \vee dest\text{-}vec1\ d < dest\text{-}vec1\ c \vee dest\text{-}vec1\ b < dest\text{-}vec1\ c \vee dest\text{-}vec1\ d < dest\text{-}vec1\ a$
$\{a\ ..\ b\} \cap \{c<..<d\} = \{\} \longleftrightarrow dest\text{-}vec1\ b < dest\text{-}vec1\ a \vee dest\text{-}vec1\ d \leq dest\text{-}vec1\ c \vee dest\text{-}vec1\ b \leq dest\text{-}vec1\ c \vee dest\text{-}vec1\ d \leq dest\text{-}vec1\ a$
$\{a<..<b\} \cap \{c\ ..\ d\} = \{\} \longleftrightarrow dest\text{-}vec1\ b \leq dest\text{-}vec1\ a \vee dest\text{-}vec1\ d < dest\text{-}vec1\ c \vee dest\text{-}vec1\ b \leq dest\text{-}vec1\ c \vee dest\text{-}vec1\ d \leq dest\text{-}vec1\ a$

$\{a<..<b\} \cap \{c<..<d\} = \{\} \longleftrightarrow$ *dest-vec1* $b \leq$ *dest-vec1* $a \vee$ *dest-vec1* $d \leq$ *dest-vec1* $c \vee$ *dest-vec1* $b \leq$ *dest-vec1* $c \vee$ *dest-vec1* $d \leq$ *dest-vec1* a
unfolding *disjoint-interval* **and** *dest-vec1-def ex-1* **by** *auto*

lemma *open-closed-interval-1*: **fixes** a :: *real^1* **shows**
$\{a<..<b\} = \{a\ ..\ b\} - \{a,\ b\}$
unfolding *expand-set-eq* **apply** *simp* **unfolding** *vector-less-def* **and** *vector-less-eq-def* **and** *all-1* **and** *dest-vec1-eq*[*THEN sym*] **and** *dest-vec1-def* **by** *auto*

lemma *closed-open-interval-1*: *dest-vec1* $(a$::*real^1*$) \leq$ *dest-vec1* b ==> $\{a\ ..\ b\}$ $= \{a<..<b\} \cup \{a,b\}$
unfolding *expand-set-eq* **apply** *simp* **unfolding** *vector-less-def* **and** *vector-less-eq-def* **and** *all-1* **and** *dest-vec1-eq*[*THEN sym*] **and** *dest-vec1-def* **by** *auto*

lemma *closed-interval-left*: **fixes** b::*real^'n*::*finite*
shows *closed* $\{x$::*real^'n*$.\ \forall i.\ x\$i \leq b\$i\}$
proof−
{ fix i
fix x::*real^'n* **assume** x:$\forall e>0.\ \exists x'\in\{x.\ \forall i.\ x\ \$\ i \leq b\ \$\ i\}.\ x' \neq x \wedge$ *dist* $x'\ x < e$
{ assume $x\$i > b\$i$
then obtain y **where** $y\ \$\ i \leq b\ \$\ i\ \ y \neq x\ \ dist\ y\ x < x\$i - b\$i$ **using** x[*THEN spec*[**where** x=$x\$i - b\$i$]] **by** *auto*
hence *False* **using** *component-le-norm*[*of* $y - x\ i$] **unfolding** *dist-def* **and** *vector-minus-component* **by** *auto* **}**
hence $x\$i \leq b\$i$ **by**(*rule ccontr*)*auto* **}**
thus *?thesis* **unfolding** *closed-limpt* **unfolding** *islimpt-approachable* **by** *blast*
qed

lemma *closed-interval-right*: **fixes** a::*real^'n*::*finite*
shows *closed* $\{x$::*real^'n*$.\ \forall i.\ a\$i \leq x\$i\}$
proof−
{ fix i
fix x::*real^'n* **assume** x:$\forall e>0.\ \exists x'\in\{x.\ \forall i.\ a\ \$\ i \leq x\ \$\ i\}.\ x' \neq x \wedge$ *dist* $x'\ x < e$
{ assume $a\$i > x\$i$
then obtain y **where** $a\ \$\ i \leq y\ \$\ i\ \ y \neq x\ \ dist\ y\ x < a\$i - x\$i$ **using** x[*THEN spec*[**where** x=$a\$i - x\$i$]] **by** *auto*
hence *False* **using** *component-le-norm*[*of* $y - x\ i$] **unfolding** *dist-def* **and** *vector-minus-component* **by** *auto* **}**
hence $a\$i \leq x\$i$ **by**(*rule ccontr*)*auto* **}**
thus *?thesis* **unfolding** *closed-limpt* **unfolding** *islimpt-approachable* **by** *blast*
qed

64.36 Intervals in general, including infinite and mixtures of open and closed.

definition *is-interval s* $\longleftrightarrow$ $(\forall\, a{\in}s.\ \forall\, b{\in}s.\ \forall\, x.\ a \le x \wedge x \le b \longrightarrow x \in s)$

lemma *is-interval-interval*: **fixes** *a*::*real^'n*::*finite* **shows**
is-interval $\{a{<}..{<}b\}$ *is-interval* $\{a\ ..\ b\}$
unfolding *is-interval-def* **apply**(*auto simp add*: *vector-less-def vector-less-eq-def*)
apply(*erule-tac x*=*i* **in** *allE*)+ **apply** *simp*
apply(*erule-tac x*=*i* **in** *allE*)+ **apply** *simp*
apply(*erule-tac x*=*i* **in** *allE*)+ **apply** *simp*
apply(*erule-tac x*=*i* **in** *allE*)+ **apply** *simp*
done

lemma *is-interval-empty*:
is-interval {}
unfolding *is-interval-def*
by *simp*

lemma *is-interval-univ*:
is-interval UNIV
unfolding *is-interval-def*
by *simp*

64.37 Closure of halfspaces and hyperplanes.

lemma *Lim-vec1-dot*: **fixes** *f* :: *real^'m* $\Rightarrow$ *real^'n*::*finite*
assumes (*f* ---> *l*) *net* **shows** ((*vec1 o* (λ*y*. *a* $\cdot$ (*f y*))) ---> *vec1*(*a* $\cdot$ *l*)) *net*
proof(*cases a* = *vec 0*)
case *True* **thus** *?thesis* **using** *dot-lzero* **and** *Lim-const*[*of 0 net*] **unfolding** *vec1-vec* **and** *o-def* **by** *auto*
next
case *False*
{ fix *e*::*real*
assume *0* < *e* $\forall\, e{>}0.\ \exists\, y.\ (\exists\, x.\ netord\ net\ x\ y) \wedge (\forall\, x.\ netord\ net\ x\ y \longrightarrow dist\ l\ (f\ x) < e)$
then obtain *x y* **where** *x*:*netord net x y* **and** *y*:$\forall\, x.$ *netord net x y* $\longrightarrow$ *dist l* (*f x*) < *e* / *norm a* **apply**(*erule-tac x*=*e* / *norm a* **in** *allE*) **apply** *auto* **using** *False* **using** *norm-ge-zero*[*of a*] **apply** *auto*
using *divide-pos-pos*[*of e norm a*] **by** *auto*
{ fix *z* **assume** *netord net z y* **hence** *dist l* (*f z*) < *e* / *norm a* **using** *y* **by** *blast*
hence *norm a* * *norm* (*l* − *f z*) < *e* **unfolding** *dist-def* **and** *pos-less-divide-eq*[*OF False*[*unfolded vec-0 zero-less-norm-iff*[*of a*, *THEN sym*]]] **and** *real-mult-commute* **by** *auto*
hence |*a* $\cdot$ *l* − *a* $\cdot$ *f z*| < *e* **using** *order-le-less-trans*[*OF norm-cauchy-schwarz-abs*[*of a l* − *f z*], *of e*] **unfolding** *dot-rsub*[*symmetric*] **by** *auto* **}**
hence $\exists\, y.\ (\exists\, x.\ netord\ net\ x\ y) \wedge (\forall\, x.\ netord\ net\ x\ y \longrightarrow |a \cdot l - a \cdot f\ x| < e)$ **using** *x* **by** *auto* **}**

thus *?thesis* **using** *assms* **unfolding** *Lim* **apply** (*auto simp add*: *dist-sym*)
unfolding *dist-vec1* **by** *auto*
qed

lemma *continuous-at-vec1-dot*:
continuous (*at x*) (*vec1 o* (λ*y*. *a* · *y*))
proof−
have ((λ*x*. *x*) −−−> *x*) (*at x*) **unfolding** *Lim-at* **by** *auto*
thus *?thesis* **unfolding** *continuous-at* **and** *o-def* **using** *Lim-vec1-dot*[*of* λ*x*. *x x at x a*] **by** *auto*
qed

lemma *continuous-on-vec1-dot*:
continuous-on s (*vec1 o* (λ*y*. *a* · *y*))
using *continuous-at-imp-continuous-on*[*of s vec1 o* (λ*y*. *a* · *y*)]
using *continuous-at-vec1-dot*
by *auto*

lemma *closed-halfspace-le*: **fixes** *a*::*real*^′*n*::*finite*
shows *closed* {*x*. *a* · *x* ≤ *b*}
proof−
have ∗:{*x* ∈ *UNIV*. (*vec1* ∘ *op* · *a*) *x* ∈ *vec1* ‘ {*r*. ∃*x*. *a* · *x* = *r* ∧ *r* ≤ *b*}} = {*x*. *a* · *x* ≤ *b*} **by** *auto*
let *?T* = {*x*::*real*^*1*. (∀*i*. *x*$*i* ≤ (*vec1 b*)$*i*)}
have *closed ?T* **using** *closed-interval-left*[*of vec1 b*] **by** *simp*
moreover have *vec1* ‘ {*r*. ∃*x*. *a* · *x* = *r* ∧ *r* ≤ *b*} = *range* (*vec1* ∘ *op* · *a*) ∩ *?T* **unfolding** *all-1*
unfolding *image-def* **by** *auto*
ultimately have ∃*T*. *closed T* ∧ *vec1* ‘ {*r*. ∃*x*. *a* · *x* = *r* ∧ *r* ≤ *b*} = *range* (*vec1* ∘ *op* · *a*) ∩ *T* **by** *auto*
hence *closedin euclidean* {*x* ∈ *UNIV*. (*vec1* ∘ *op* · *a*) *x* ∈ *vec1* ‘ {*r*. ∃*x*. *a* · *x* = *r* ∧ *r* ≤ *b*}}
using *continuous-on-vec1-dot*[*of UNIV a*, *unfolded continuous-on-closed subtopology-UNIV*]
unfolding *closedin-closed*
by (*auto elim*!: *allE*[**where** *x*=*vec1* ‘ {*r*. (∃*x*. *a* · *x* = *r* ∧ *r* ≤ *b*)}])
thus *?thesis* **unfolding** *closed-closedin*[*THEN sym*] **and** ∗ **by** *auto*
qed

lemma *closed-halfspace-ge*: *closed* {*x*. *a* · *x* ≥ *b*}
using *closed-halfspace-le*[*of* −*a* −*b*] **unfolding** *dot-lneg* **by** *auto*

lemma *closed-hyperplane*: *closed* {*x*. *a* · *x* = *b*}
proof−
have {*x*. *a* · *x* = *b*} = {*x*. *a* · *x* ≥ *b*} ∩ {*x*. *a* · *x* ≤ *b*} **by** *auto*
thus *?thesis* **using** *closed-halfspace-le*[*of a b*] **and** *closed-halfspace-ge*[*of b a*]
using *closed-Int* **by** *auto*
qed

lemma *closed-halfspace-component-le*:

shows *closed* {*x*::*real^'n*::*finite*. *x*$*i* ≤ *a*}
using *closed-halfspace-le*[*of* (*basis i*)::*real^'n a*] **unfolding** *dot-basis*[*OF assms*]
by *auto*

lemma *closed-halfspace-component-ge*:
shows *closed* {*x*::*real^'n*::*finite*. *x*$*i* ≥ *a*}
using *closed-halfspace-ge*[*of a* (*basis i*)::*real^'n*] **unfolding** *dot-basis*[*OF assms*]
by *auto*

Openness of halfspaces.

lemma *open-halfspace-lt*: *open* {*x*. *a* · *x* < *b*}
proof−
have *UNIV* − {*x*. *b* ≤ *a* · *x*} = {*x*. *a* · *x* < *b*} **by** *auto*
thus *?thesis* **using** *closed-halfspace-ge*[*unfolded closed-def*, *of b a*] **by** *auto*
qed

lemma *open-halfspace-gt*: *open* {*x*. *a* · *x* > *b*}
proof−
have *UNIV* − {*x*. *b* ≥ *a* · *x*} = {*x*. *a* · *x* > *b*} **by** *auto*
thus *?thesis* **using** *closed-halfspace-le*[*unfolded closed-def*, *of a b*] **by** *auto*
qed

lemma *open-halfspace-component-lt*:
shows *open* {*x*::*real^'n*::*finite*. *x*$*i* < *a*}
using *open-halfspace-lt*[*of* (*basis i*)::*real^'n a*] **unfolding** *dot-basis*[*OF assms*]
by *auto*

lemma *open-halfspace-component-gt*:
shows *open* {*x*::*real^'n*::*finite*. *x*$*i* > *a*}
using *open-halfspace-gt*[*of a* (*basis i*)::*real^'n*] **unfolding** *dot-basis*[*OF assms*]
by *auto*

This gives a simple derivation of limit component bounds.

lemma *Lim-component-le*: **fixes** *f* :: *'a* ⇒ *real^'n*::*finite*
assumes (*f* −−−> *l*) *net* ¬ (*trivial-limit net*) *eventually* (λ*x*. *f*(*x*)$*i* ≤ *b*) *net*
shows *l*$*i* ≤ *b*
proof−
{ fix *x* **have** *x* ∈ {*x*::*real^'n*. *basis i* · *x* ≤ *b*} ⟷ *x*$*i* ≤ *b* **unfolding** *dot-basis*
by *auto* **} note** ∗ = *this*
show *?thesis* **using** *Lim-in-closed-set*[*of* {*x*. *basis i* · *x* ≤ *b*} *f net l*] **unfolding**
∗
using *closed-halfspace-le*[*of* (*basis i*)::*real^'n b*] **and** *assms*(*1*,*2*,*3*) **by** *auto*
qed

lemma *Lim-component-ge*: **fixes** *f* :: *'a* ⇒ *real^'n*::*finite*
assumes (*f* −−−> *l*) *net* ¬ (*trivial-limit net*) *eventually* (λ*x*. *b* ≤ (*f x*)$*i*) *net*
shows *b* ≤ *l*$*i*
proof−
{ fix *x* **have** *x* ∈ {*x*::*real^'n*. *basis i* · *x* ≥ *b*} ⟷ *x*$*i* ≥ *b* **unfolding** *dot-basis*
by *auto* **} note** ∗ = *this*

show *?thesis* **using** *Lim-in-closed-set*[*of* {*x*. *basis i* · *x* ≥ *b*} *f net l*] **unfolding** *
using *closed-halfspace-ge*[*of b* (*basis i*)::*real^'n*] **and** *assms*(*1*,*2*,*3*) **by** *auto*
qed

lemma *Lim-component-eq*: **fixes** *f* :: *'a* ⇒ *real^'n*::*finite*
assumes *net*:(*f* ---> *l*) *net* ~(*trivial-limit net*) **and** *ev*:*eventually* (λ*x*. *f*(*x*)$*i* = *b*) *net*
shows *l*$*i* = *b*
using *ev*[*unfolded order-eq-iff eventually-and*] **using** *Lim-component-ge*[*OF net*, *of b i*] **and** *Lim-component-le*[*OF net*, *of i b*] **by** *auto*

lemma *Lim-drop-le*: **fixes** *f* :: *'a* ⇒ *real^1* **shows**
(*f* ---> *l*) *net* ⟹ ~(*trivial-limit net*) ⟹ *eventually* (λ*x*. *dest-vec1* (*f x*) ≤ *b*) *net* ==> *dest-vec1 l* ≤ *b*
using *Lim-component-le*[*of f l net 1 b*] **unfolding** *dest-vec1-def* **by** *auto*

lemma *Lim-drop-ge*: **fixes** *f* :: *'a* ⇒ *real^1* **shows**
(*f* ---> *l*) *net* ⟹ ~(*trivial-limit net*) ⟹ *eventually* (λ*x*. *b* ≤ *dest-vec1* (*f x*)) *net* ==> *b* ≤ *dest-vec1 l*
using *Lim-component-ge*[*of f l net b 1*] **unfolding** *dest-vec1-def* **by** *auto*

Limits relative to a union.

lemma *Lim-within-union*:
(*f* ---> *l*) (*at x within* (*s* ∪ *t*)) ⟷
(*f* ---> *l*) (*at x within s*) ∧ (*f* ---> *l*) (*at x within t*)
unfolding *Lim-within* **apply** *auto* **apply** *blast* **apply** *blast*
apply(*erule-tac x*=*e* **in** *allE*)+ **apply** *auto*
apply(*rule-tac x*=*min d da* **in** *exI*) **by** *auto*

lemma *continuous-on-union*:
assumes *closed s closed t continuous-on s f continuous-on t f*
shows *continuous-on* (*s* ∪ *t*) *f*
using *assms* **unfolding** *continuous-on* **unfolding** *Lim-within-union*
unfolding *Lim* **unfolding** *trivial-limit-within* **unfolding** *closed-limpt* **by** *auto*

lemma *continuous-on-cases*: **fixes** *g* :: *real^'m*::*finite* ⇒ *real ^'n*::*finite*
assumes *closed s closed t continuous-on s f continuous-on t g*
∀*x*. (*x*∈*s* ∧ ¬ *P x*) ∨ (*x* ∈ *t* ∧ *P x*) ⟶ *f x* = *g x*
shows *continuous-on* (*s* ∪ *t*) (λ*x*. *if P x then f x else g x*)
proof−
let *?h* = (λ*x*. *if P x then f x else g x*)
have ∀*x*∈*s*. *f x* = (*if P x then f x else g x*) **using** *assms*(*5*) **by** *auto*
hence *continuous-on s ?h* **using** *continuous-on-eq*[*of s f ?h*] **using** *assms*(*3*) **by** *auto*
moreover
have ∀*x*∈*t*. *g x* = (*if P x then f x else g x*) **using** *assms*(*5*) **by** *auto*
hence *continuous-on t ?h* **using** *continuous-on-eq*[*of t g ?h*] **using** *assms*(*4*) **by** *auto*

ultimately show *?thesis* **using** *continuous-on-union*[*OF assms*(*1*,*2*), *of ?h*] **by** *auto*
qed

Some more convenient intermediate-value theorem formulations.

lemma *connected-ivt-hyperplane*: **fixes** *y* :: *real^'n*::*finite*
assumes *connected s x* ∈ *s y* ∈ *s a* · *x* ≤ *b b* ≤ *a* · *y*
shows ∃ *z* ∈ *s*. *a* · *z* = *b*
proof(*rule ccontr*)
assume *as*:¬ (∃ *z*∈*s*. *a* · *z* = *b*)
let *?A* = {*x*::*real^'n*. *a* · *x* < *b*}
let *?B* = {*x*::*real^'n*. *a* · *x* > *b*}
have *open ?A open ?B* **using** *open-halfspace-lt* **and** *open-halfspace-gt* **by** *auto*
moreover have *?A* ∩ *?B* = {} **by** *auto*
moreover have *s* ⊆ *?A* ∪ *?B* **using** *as* **by** *auto*
ultimately show *False* **using** *assms*(*1*)[*unfolded connected-def not-ex*, *THEN spec*[**where** *x*=*?A*], *THEN spec*[**where** *x*=*?B*]] **and** *assms*(*2*−*5*) **by** *auto*
qed

lemma *connected-ivt-component*: **fixes** *x*::*real^'n*::*finite* **shows**
connected s ⟹ *x* ∈ *s* ⟹ *y* ∈ *s* ⟹ *x*$*k* ≤ *a* ⟹ *a* ≤ *y*$*k* ⟹ (∃ *z*∈*s*. *z*$*k* = *a*)
using *connected-ivt-hyperplane*[*of s x y* (*basis k*)::*real^'n a*] **by** (*auto simp add*: *dot-basis*)

Also more convenient formulations of monotone convergence.

lemma *bounded-increasing-convergent*: **fixes** *s*::*nat* ⇒ *real^1*
assumes *bounded* {*s n*| *n*::*nat*. *True*} ∀ *n*. *dest-vec1*(*s n*) ≤ *dest-vec1*(*s*(*Suc n*))
shows ∃ *l*. (*s* −−−> *l*) *sequentially*
proof−
obtain *a* **where** *a*:∀ *n*. |*dest-vec1* (*s n*)| ≤ *a* **using** *assms*(*1*)[*unfolded bounded-def abs-dest-vec1*] **by** *auto*
{ **fix** *m*::*nat*
have ⋀ *n*. *n*≥*m* ⟶ *dest-vec1* (*s m*) ≤ *dest-vec1* (*s n*)
apply(*induct-tac n*) **apply** *simp* **using** *assms*(*2*) **apply**(*erule-tac x*=*na* **in** *allE*) **by**(*auto simp add*: *not-less-eq-eq*) }
hence ∀ *m n*. *m* ≤ *n* ⟶ *dest-vec1* (*s m*) ≤ *dest-vec1* (*s n*) **by** *auto*
then obtain *l* **where** ∀ *e*>*0*. ∃ *N*. ∀ *n*≥*N*. |*dest-vec1* (*s n*) − *l*| < *e* **using** *convergent-bounded-monotone*[*OF a*] **by** *auto*
thus *?thesis* **unfolding** *Lim-sequentially* **apply**(*rule-tac x*=*vec1 l* **in** *exI*)
unfolding *dist-def* **unfolding** *abs-dest-vec1* **and** *dest-vec1-sub* **by** *auto*
qed

64.38 Basic homeomorphism definitions.

definition *homeomorphism s t f g* ≡
(∀ *x*∈*s*. (*g*(*f x*) = *x*)) ∧ (*f* ‘ *s* = *t*) ∧ *continuous-on s f* ∧
(∀ *y*∈*t*. (*f*(*g y*) = *y*)) ∧ (*g* ‘ *t* = *s*) ∧ *continuous-on t g*

definition *homeomorphic* :: $((real\hat{}'a::finite)\ set) \Rightarrow ((real\hat{}'b::finite)\ set) \Rightarrow bool$
(**infixr** *homeomorphic 60*) **where**
homeomorphic-def: *s homeomorphic t* $\equiv$ $(\exists f\ g.$ *homeomorphism s t f g*$)$

lemma *homeomorphic-refl*: *s homeomorphic s*
unfolding *homeomorphic-def*
unfolding *homeomorphism-def*
using *continuous-on-id*
apply(*rule-tac x* = $(\lambda x::real\hat{}'a.x)$ **in** *exI*)
apply(*rule-tac x* = $(\lambda x::real\hat{}'b.x)$ **in** *exI*)
by *blast*

lemma *homeomorphic-sym*:
s homeomorphic t $\longleftrightarrow$ *t homeomorphic s*
unfolding *homeomorphic-def*
unfolding *homeomorphism-def*
by *blast*

lemma *homeomorphic-trans*:
assumes *s homeomorphic t t homeomorphic u* **shows** *s homeomorphic u*
proof−
obtain *f1 g1* **where** *fg1*:$\forall x{\in}s.$ *g1* (*f1 x*) = *x* *f1* ' *s* = *t continuous-on s f1* $\forall y{\in}t.$ *f1* (*g1 y*) = *y g1* ' *t* = *s continuous-on t g1*
using *assms*(*1*) **unfolding** *homeomorphic-def homeomorphism-def* **by** *auto*
obtain *f2 g2* **where** *fg2*:$\forall x{\in}t.$ *g2* (*f2 x*) = *x* *f2* ' *t* = *u continuous-on t f2* $\forall y{\in}u.$ *f2* (*g2 y*) = *y g2* ' *u* = *t continuous-on u g2*
using *assms*(*2*) **unfolding** *homeomorphic-def homeomorphism-def* **by** *auto*

{ fix *x* **assume** *x*$\in$*s* **hence** (*g1* $\circ$ *g2*) ((*f2* $\circ$ *f1*) *x*) = *x* **using** *fg1*(*1*)[*THEN bspec*[**where** *x*=*x*]] **and** *fg2*(*1*)[*THEN bspec*[**where** *x*=*f1 x*]] **and** *fg1*(*2*) **by** *auto* **}**
moreover have (*f2* $\circ$ *f1*) ' *s* = *u* **using** *fg1*(*2*) *fg2*(*2*) **by** *auto*
moreover have *continuous-on s* (*f2* $\circ$ *f1*) **using** *continuous-on-compose*[*OF fg1*(*3*)] **and** *fg2*(*3*) **unfolding** *fg1*(*2*) **by** *auto*
moreover { fix *y* **assume** *y*$\in$*u* **hence** (*f2* $\circ$ *f1*) ((*g1* $\circ$ *g2*) *y*) = *y* **using** *fg2*(*4*)[*THEN bspec*[**where** *x*=*y*]] **and** *fg1*(*4*)[*THEN bspec*[**where** *x*=*g2 y*]] **and** *fg2*(*5*) **by** *auto* **}**
moreover have (*g1* $\circ$ *g2*) ' *u* = *s* **using** *fg1*(*5*) *fg2*(*5*) **by** *auto*
moreover have *continuous-on u* (*g1* $\circ$ *g2*) **using** *continuous-on-compose*[*OF fg2*(*6*)] **and** *fg1*(*6*) **unfolding** *fg2*(*5*) **by** *auto*
ultimately show *?thesis* **unfolding** *homeomorphic-def homeomorphism-def* **apply**(*rule-tac x*=*f2* $\circ$ *f1* **in** *exI*) **apply**(*rule-tac x*=*g1* $\circ$ *g2* **in** *exI*) **by** *auto*
qed

lemma *homeomorphic-minimal*:
s homeomorphic t $\longleftrightarrow$
$(\exists f\ g.\ (\forall x{\in}s.\ f(x) \in t \wedge (g(f(x)) = x)) \wedge$
$(\forall y{\in}t.\ g(y) \in s \wedge (f(g(y)) = y)) \wedge$
continuous-on s f $\wedge$ *continuous-on t g*$)$

unfolding *homeomorphic-def homeomorphism-def*
apply *auto* **apply** (*rule-tac x=f* **in** *exI*) **apply** (*rule-tac x=g* **in** *exI*)
apply *auto* **apply** (*rule-tac x=f* **in** *exI*) **apply** (*rule-tac x=g* **in** *exI*) **apply** *auto*
unfolding *image-iff*
apply(*erule-tac x=g x* **in** *ballE*) **apply**(*erule-tac x=x* **in** *ballE*)
apply *auto* **apply**(*rule-tac x=g x* **in** *bexI*) **apply** *auto*
apply(*erule-tac x=f x* **in** *ballE*) **apply**(*erule-tac x=x* **in** *ballE*)
apply *auto* **apply**(*rule-tac x=f x* **in** *bexI*) **by** *auto*

64.39 Relatively weak hypotheses if a set is compact.

definition *inv-on f s* = $(\lambda x.\ SOME\ y.\ y{\in}s \wedge f\ y = x)$

lemma assumes *inj-on f s x∈s*
shows *inv-on f s* (*f x*) = *x*
using *assms* **unfolding** *inj-on-def inv-on-def* **by** *auto*

lemma *homeomorphism-compact*:
assumes *compact s continuous-on s f f ‘ s = t inj-on f s*
shows $\exists g.$ *homeomorphism s t f g*
proof−
def $g \equiv \lambda x.\ SOME\ y.\ y{\in}s \wedge f\ y = x$
have *g*:$\forall x{\in}s.\ g\ (f\ x) = x$ **using** *assms*(*3*) *assms*(*4*)[*unfolded inj-on-def*] **unfolding** *g-def* **by** *auto*
{ fix *y* **assume** *y∈t*
then obtain *x* **where** *x*:*f x = y x∈s* **using** *assms*(*3*) **by** *auto*
hence *g* (*f x*) = *x* **using** *g* **by** *auto*
hence *f* (*g y*) = *y* **unfolding** *x*(*1*)[*THEN sym*] **by** *auto* **}**
hence *g′*:$\forall x{\in}t.$ *f* (*g x*) = *x* **by** *auto*
moreover
{ fix *x*
have *x∈s* $\Longrightarrow$ *x* ∈ *g ‘ t* **using** *g*[*THEN bspec*[**where** *x=x*]] **unfolding** *image-iff* **using** *assms*(*3*) **by**(*auto intro*!: *bexI*[**where** *x=f x*])
moreover
{ assume *x∈g ‘ t*
then obtain *y* **where** *y*:*y∈t g y = x* **by** *auto*
then obtain *x′* **where** *x′*:*x′∈s f x′ = y* **using** *assms*(*3*) **by** *auto*
hence *x* ∈ *s* **unfolding** *g-def* **using** *someI2*[*of* $\lambda b.\ b{\in}s \wedge f\ b = y$ *x′* $\lambda x.$ *x∈s*] **unfolding** *y*(*2*)[*THEN sym*] **and** *g-def* **by** *auto* **}**
ultimately have *x∈s* $\longleftrightarrow$ *x* ∈ *g ‘ t* **by** *auto* **}**
hence *g ‘ t = s* **by** *auto*
ultimately
show *?thesis* **unfolding** *homeomorphism-def homeomorphic-def*
apply(*rule-tac x=g* **in** *exI*) **using** *g* **and** *assms*(*3*) **and** *continuous-on-inverse*[*OF assms*(*2*,*1*), *of g*, *unfolded assms*(*3*)] **and** *assms*(*2*) **by** *auto*
qed

lemma *homeomorphic-compact*:
compact s $\Longrightarrow$ *continuous-on s f* $\Longrightarrow$ (*f ‘ s = t*) $\Longrightarrow$ *inj-on f s*

$\Longrightarrow$ *s homeomorphic t*
unfolding *homeomorphic-def* **by**(*metis homeomorphism-compact*)

Preservation of topological properties.

lemma *homeomorphic-compactness*:
s homeomorphic t ==> (compact s ⟷ compact t)
unfolding *homeomorphic-def homeomorphism-def*
by (*metis compact-continuous-image*)

Results on translation, scaling etc.

lemma *homeomorphic-scaling*:
assumes *c ≠ 0* **shows** *s homeomorphic ((λx. c *s x) ' s)*
unfolding *homeomorphic-minimal*
apply(*rule-tac x=λx. c *s x* **in** *exI*)
apply(*rule-tac x=λx. (1 / c) *s x* **in** *exI*)
apply *auto* **unfolding** *vector-smult-assoc* **using** *assms* **apply** *auto*
using *continuous-on-cmul*[*OF continuous-on-id*] **by** *auto*

lemma *homeomorphic-translation*:
s homeomorphic ((λx. a + x) ' s)
unfolding *homeomorphic-minimal*
apply(*rule-tac x=λx. a + x* **in** *exI*)
apply(*rule-tac x=λx. −a + x* **in** *exI*)
using *continuous-on-add*[*OF continuous-on-const continuous-on-id*] **by** *auto*

lemma *homeomorphic-affinity*:
assumes *c ≠ 0* **shows** *s homeomorphic ((λx. a + c *s x) ' s)*
proof−
have *∗:op + a ' op *s c ' s = (λx. a + c *s x) ' s* **by** *auto*
show *?thesis*
using *homeomorphic-trans*
using *homeomorphic-scaling*[*OF assms*, *of s*]
using *homeomorphic-translation*[*of (λx. c *s x) ' s a*] **unfolding** *∗* **by** *auto*
qed

lemma *homeomorphic-balls*: **fixes** *a b ::real^'a::finite*
assumes *0 < d 0 < e*
shows (*ball a d*) *homeomorphic* (*ball b e*) (**is** *?th*)
(*cball a d*) *homeomorphic* (*cball b e*) (**is** *?cth*)
proof−
have *∗:|e / d| > 0 |d / e| >0* **using** *assms* **using** *divide-pos-pos* **by** *auto*
show *?th* **unfolding** *homeomorphic-minimal*
apply(*rule-tac x=λx. b + (e/d) *s (x − a)* **in** *exI*)
apply(*rule-tac x=λx. a + (d/e) *s (x − b)* **in** *exI*)
apply (*auto simp add*: *dist-sym*) **unfolding** *dist-def* **and** *vector-smult-assoc*
using *assms* **apply** *auto*
unfolding *norm-minus-cancel* **and** *norm-mul*
using *continuous-on-add*[*OF continuous-on-const continuous-on-cmul*[*OF continuous-on-sub*[*OF continuous-on-id continuous-on-const*]]]

apply (*auto simp add*: *dist-sym*)
using *pos-less-divide-eq*[*OF* ∗(*1*), *THEN sym*] **unfolding** *real-mult-commute*[*of* - |*e* / *d*|]
using *pos-less-divide-eq*[*OF* ∗(*2*), *THEN sym*] **unfolding** *real-mult-commute*[*of* - |*d* / *e*|]
by (*auto simp add*: *dist-sym*)
next
have ∗:|*e* / *d*| > *0* |*d* / *e*| >*0* **using** *assms* **using** *divide-pos-pos* **by** *auto*
show *?cth* **unfolding** *homeomorphic-minimal*
apply(*rule-tac x*=λ*x*. *b* + (*e*/*d*) ∗*s* (*x* − *a*) **in** *exI*)
apply(*rule-tac x*=λ*x*. *a* + (*d*/*e*) ∗*s* (*x* − *b*) **in** *exI*)
apply (*auto simp add*: *dist-sym*) **unfolding** *dist-def* **and** *vector-smult-assoc* **using** *assms* **apply** *auto*
unfolding *norm-minus-cancel* **and** *norm-mul*
using *continuous-on-add*[*OF continuous-on-const continuous-on-cmul*[*OF continuous-on-sub*[*OF continuous-on-id continuous-on-const*]]]
apply *auto*
using *pos-le-divide-eq*[*OF* ∗(*1*), *THEN sym*] **unfolding** *real-mult-commute*[*of* - |*e* / *d*|]
using *pos-le-divide-eq*[*OF* ∗(*2*), *THEN sym*] **unfolding** *real-mult-commute*[*of* - |*d* / *e*|]
by *auto*
qed

"Isometry" (up to constant bounds) of injective linear map etc.

lemma *cauchy-isometric*:
assumes *e*:*0* < *e* **and** *s*:*subspace s* **and** *f*:*linear f* **and** *normf*:∀ *x*∈*s*. *norm*(*f x*) ≥ *e* ∗ *norm*(*x*) **and** *xs*:∀ *n*::*nat*. *x n* ∈ *s* **and** *cf*:*cauchy*(*f o x*)
shows *cauchy x*
proof−
{ fix *d*::*real* **assume** *d*>*0*
then obtain *N* **where** *N*:∀ *n*≥*N*. *norm* (*f* (*x n*) − *f* (*x N*)) < *e* ∗ *d*
using *cf*[*unfolded cauchy o-def dist-def*, *THEN spec*[**where** *x*=*e*∗*d*]] **and** *e* **and** *mult-pos-pos*[*of e d*] **by** *auto*
{ fix *n* **assume** *n*≥*N*
hence *norm* (*f* (*x n* − *x N*)) < *e* ∗ *d* **using** *N*[*THEN spec*[**where** *x*=*n*]] **unfolding** *linear-sub*[*OF f*, *THEN sym*] **by** *auto*
moreover have *e* ∗ *norm* (*x n* − *x N*) ≤ *norm* (*f* (*x n* − *x N*))
using *subspace-sub*[*OF s*, *of x n x N*] **using** *xs*[*THEN spec*[**where** *x*=*N*]] **and** *xs*[*THEN spec*[**where** *x*=*n*]]
using *normf*[*THEN bspec*[**where** *x*=*x n* − *x N*]] **by** *auto*
ultimately have *norm* (*x n* − *x N*) < *d* **using** ‹*e*>*0*›
using *mult-left-less-imp-less*[*of e norm* (*x n* − *x N*) *d*] **by** *auto* **}**
hence ∃ *N*. ∀ *n*≥*N*. *norm* (*x n* − *x N*) < *d* **by** *auto* **}**
thus *?thesis* **unfolding** *cauchy* **and** *dist-def* **by** *auto*
qed

lemma *complete-isometric-image*:
assumes *0* < *e* **and** *s*:*subspace s* **and** *f*:*linear f* **and** *normf*:∀ *x*∈*s*. *norm*(*f x*)

$\geq e * norm(x)$ **and** *cs*:*complete s*
shows *complete*(f ‘ s)
proof−
{ fix g **assume** *as*:$\forall\, n$::*nat*. $g\ n \in f$ ‘ s **and** *cfg*:*cauchy g*
then obtain x **where** $\forall\, n.\ x\ n \in s \wedge g\ n = f\ (x\ n)$ **unfolding** *image-iff* **and** *Bex-def*
using *choice*[*of* $\lambda\ n\ xa.\ xa \in s \wedge g\ n = f\ xa$] **by** *auto*
hence *x*:$\forall\, n.\ x\ n \in s\ \ \forall\, n.\ g\ n = f\ (x\ n)$ **by** *auto*
hence $f \circ x = g$ **unfolding** *expand-fun-eq* **by** *auto*
then obtain l **where** $l \in s$ **and** *l*:(x −−−> l) *sequentially*
using *cs*[*unfolded complete-def*, *THEN spec*[**where** x=x]]
using *cauchy-isometric*[*OF* ‹0<*e*› *s f normf*] **and** *cfg* **and** $x(1)$ **by** *auto*
hence $\exists\, l \in f$ ‘ s. (g −−−> l) *sequentially*
using *linear-continuous-at*[*OF f*, *unfolded continuous-at-sequentially*, *THEN spec*[**where** x=x], *of l*]
unfolding ‹$f \circ x = g$› **by** *auto* **}**
thus *?thesis* **unfolding** *complete-def* **by** *auto*
qed

lemma *dist-0-norm*:*dist 0 x* = *norm x* **unfolding** *dist-def* **by**(*auto simp add*: *norm-minus-cancel*)

lemma *injective-imp-isometric*: **fixes** *f*::*real*^'*m*::*finite* ⇒ *real*^'*n*::*finite*
assumes *s*:*closed s* *subspace s* **and** *f*:*linear f* $\forall\, x \in s.\ (f\ x = 0) \longrightarrow (x = 0)$
shows $\exists\, e > 0.\ \forall\, x \in s.\ norm\ (f\ x) \geq e * norm(x)$
proof(*cases s* ⊆ {*0*::*real*^'*m*})
case *True*
{ fix x **assume** $x \in s$
hence $x = 0$ **using** *True* **by** *auto*
hence *norm x* ≤ *norm* ($f\ x$) **by** *auto* **}**
thus *?thesis* **by**(*auto intro*!: *exI*[**where** *x*=*1*])
next
case *False*
then obtain a **where** *a*:$a \neq 0$ $a \in s$ **by** *auto*
from *False* **have** $s \neq \{\}$ **by** *auto*
let *?S* = {$f\ x$| x. ($x \in s \wedge norm\ x = norm\ a$)}
let *?S'* = {*x*::*real*^'*m*. $x \in s \wedge norm\ x = norm\ a$}
let *?S''* = {*x*::*real*^'*m*. *norm x* = *norm a*}

have *?S''* = *frontier*(*cball 0* (*norm a*)) **unfolding** *frontier-cball* **and** *dist-def* **by** (*auto simp add*: *norm-minus-cancel*)
hence *compact ?S''* **using** *compact-frontier*[*OF compact-cball*, *of 0 norm a*] **by** *auto*
moreover have *?S'* = *s* ∩ *?S''* **by** *auto*
ultimately have *compact ?S'* **using** *closed-inter-compact*[*of s ?S''*] **using** *s*(*1*) **by** *auto*
moreover have ∗:*f* ‘ *?S'* = *?S* **by** *auto*
ultimately have *compact ?S* **using** *compact-continuous-image*[*OF linear-continuous-on*[*OF f*(*1*)], *of ?S'*] **by** *auto*

hence *closed ?S* **using** *compact-imp-closed* **by** *auto*
moreover have *?S* $\neq$ {} **using** *a* **by** *auto*
ultimately obtain *b'* **where** *b'*$\in$*?S* $\forall$ *y*$\in$*?S*. *norm b'* $\leq$ *norm y* **using** *distance-attains-inf*[*of ?S 0*] **unfolding** *dist-0-norm* **by** *auto*
then obtain *b* **where** *b*$\in$*s* **and** *ba*:*norm b* = *norm a* **and** *b*:$\forall$ *x*$\in${*x* $\in$ *s*. *norm x* = *norm a*}. *norm* (*f b*) $\leq$ *norm* (*f x*) **unfolding** $*$[*THEN sym*] **unfolding** *image-iff* **by** *auto*

let *?e* = *norm* (*f b*) / *norm b*
have *norm b* > *0* **using** *ba* **and** *a* **and** *norm-ge-zero* **by** *auto*
moreover have *norm* (*f b*) > *0* **using** *f*(*2*)[*THEN bspec*[**where** *x*=*b*], *OF* ‹*b*$\in$*s*›] **using** ‹*norm b* >*0*› **unfolding** *zero-less-norm-iff* **by** *auto*
ultimately have *0* < *norm* (*f b*) / *norm b* **by**(*simp only*: *divide-pos-pos*)
moreover
{ fix *x* **assume** *x*$\in$*s*
hence *norm* (*f b*) / *norm b* $*$ *norm x* $\leq$ *norm* (*f x*)
proof(*cases x*=*0*)
case *True* **thus** *norm* (*f b*) / *norm b* $*$ *norm x* $\leq$ *norm* (*f x*) **by** *auto*
next
case *False*
hence $*$:*0* < *norm a* / *norm x* **using** ‹*a*$\neq$*0*› **unfolding** *zero-less-norm-iff*[*THEN sym*] **by**(*simp only*: *divide-pos-pos*)
have $\forall$ *c*. $\forall$ *x*$\in$*s*. *c* $*s$ *x* $\in$ *s* **using** *s*[*unfolded subspace-def*] **by** *auto*
hence (*norm a* / *norm x*) $*s$ *x* $\in$ {*x* $\in$ *s*. *norm x* = *norm a*} **using** ‹*x*$\in$*s*› **and** ‹*x*$\neq$*0*› **by** *auto*
thus *norm* (*f b*) / *norm b* $*$ *norm x* $\leq$ *norm* (*f x*) **using** *b*[*THEN bspec*[**where** *x*=(*norm a* / *norm x*) $*s$ *x*]]
unfolding *linear-cmul*[*OF f*(*1*)] **and** *norm-mul* **and** *ba* **using** ‹*x*$\neq$*0*› ‹*a*$\neq$*0*›
by (*auto simp add*: *real-mult-commute pos-le-divide-eq pos-divide-le-eq*)
qed }
ultimately
show *?thesis* **by** *auto*
qed

lemma *closed-injective-image-subspace*:
assumes *subspace s linear f* $\forall$ *x*$\in$*s*. *f x* = *0* $--$> *x* = *0 closed s*
shows *closed*(*f* ‘ *s*)
proof−
obtain *e* **where** *e*>*0* **and** *e*:$\forall$ *x*$\in$*s*. *e* $*$ *norm x* $\leq$ *norm* (*f x*) **using** *injective-imp-isometric*[*OF assms*(*4*,*1*,*2*,*3*)] **by** *auto*
show *?thesis* **using** *complete-isometric-image*[*OF* ‹*e*>*0*› *assms*(*1*,*2*) *e*] **and** *assms*(*4*)
unfolding *complete-eq-closed*[*THEN sym*] **by** *auto*
qed

64.40 Some properties of a canonical subspace.

lemma *subspace-substandard*:
subspace {*x*::*real*^'*n*. ($\forall$ *i*. *P i* $\longrightarrow$ *x*$*i* = *0*)}

unfolding *subspace-def* **by**(*auto simp add*: *vector-add-component vector-smult-component elim*!: *ballE*)

lemma *closed-substandard*:
closed {*x*::*real*^'*n*::*finite*. ∀ *i*. *P i* −−> *x*$*i* = *0*} (**is** *closed ?A*)
proof−
let *?D* = {*i*. *P i*}
let *?Bs* = {{*x*::*real*^'*n*. *basis i* · *x* = *0*}| *i*. *i* ∈ *?D*}
{ fix *x*
{ assume *x*∈*?A*
hence *x*:∀ *i*∈*?D*. *x* $ *i* = *0* **by** *auto*
hence *x*∈ ⋂ *?Bs* **by**(*auto simp add*: *dot-basis x*) **}**
moreover
{ assume *x*:*x*∈⋂ *?Bs*
{ fix *i* **assume** *i*:*i* ∈ *?D*
then obtain *B* **where** *BB*:*B* ∈ *?Bs* **and** *B*:*B* = {*x*::*real*^'*n*. *basis i* · *x* = *0*} **by** *auto*
hence *x* $ *i* = *0* **unfolding** *B* **using** *x* **unfolding** *dot-basis* **by** *auto* **}**
hence *x*∈*?A* **by** *auto* **}**
ultimately have *x*∈*?A* ⟷ *x*∈ ⋂ *?Bs* **by** *auto* **}**
hence *?A* = ⋂ *?Bs* **by** *auto*
thus *?thesis* **by**(*auto simp add*: *closed-Inter closed-hyperplane*)
qed

lemma *dim-substandard*:
shows *dim* {*x*::*real*^'*n*::*finite*. ∀ *i*. *i* ∉ *d* ⟶ *x*$*i* = *0*} = *card d* (**is** *dim ?A* = -)
proof−
let *?D* = *UNIV*::'*n set*
let *?B* = (*basis*::'*n*⇒*real*^'*n*) ‘ *d*

let *?bas* = *basis*::'*n* ⇒ *real*^'*n*

have *?B* ⊆ *?A* **by** *auto*

moreover
{ fix *x*::*real*^'*n* **assume** *x*∈*?A*
with *finite*[*of d*]
have *x*∈ *span ?B*
proof(*induct d arbitrary*: *x*)
case *empty* **hence** *x*=*0* **unfolding** *Cart-eq* **by** *auto*
thus *?case* **using** *subspace-0*[*OF subspace-span*[*of* {}]] **by** *auto*
next
case (*insert k F*)
hence ∗:∀ *i*. *i* ∉ *insert k F* ⟶ *x* $ *i* = *0* **by** *auto*
have ∗∗:*F* ⊆ *insert k F* **by** *auto*
def *y* ≡ *x* − *x*$*k* ∗*s basis k*
have *y*:*x* = *y* + (*x*$*k*) ∗*s basis k* **unfolding** *y-def* **by** *auto*
{ fix *i* **assume** *i'*:*i* ∉ *F*
hence *y* $ *i* = *0* **unfolding** *y-def* **unfolding** *vector-minus-component*

```
      and vector-smult-component and basis-component
      using *[THEN spec[where x=i]] by auto }
    hence y ∈ span (basis ‘ (insert k F)) using insert(3)
     using span-mono[of ?bas ‘ F ?bas ‘ (insert k F)]
     using image-mono[OF **, of basis] by auto
    moreover
    have basis k ∈ span (?bas ‘ (insert k F)) by(rule span-superset, auto)
    hence x$k *s basis k ∈ span (?bas ‘ (insert k F)) using span-mul by auto
    ultimately
    have y + x$k *s basis k ∈ span (?bas ‘ (insert k F))
     using span-add by auto
    thus ?case using y by auto
   qed
  }
  hence ?A ⊆ span ?B by auto

  moreover
  { fix x assume x ∈ ?B
   hence x∈{(basis i)::real^'n |i. i ∈ ?D} using assms by auto  }
 hence independent ?B using independent-mono[OF independent-stdbasis, of ?B]
and assms by auto

  moreover
  have d ⊆ ?D unfolding subset-eq using assms by auto
  hence *:inj-on (basis::'n⇒real^'n) d using subset-inj-on[OF basis-inj, of d] by
auto
  have ?B hassize (card d) unfolding hassize-def and card-image[OF *] by auto

  ultimately show ?thesis using dim-unique[of basis ‘ d ?A] by auto
qed
```

Hence closure and completeness of all subspaces.

```
lemma closed-subspace-lemma: n ≤ card (UNIV::'n::finite set) ⟹ ∃ A::'n set.
card A = n
apply (induct n)
apply (rule-tac x={} in exI, simp)
apply clarsimp
apply (subgoal-tac ∃x. x ∉ A)
apply (erule exE)
apply (rule-tac x=insert x A in exI, simp)
apply (subgoal-tac A ≠ UNIV, auto)
done

lemma closed-subspace: fixes s::(real^'n::finite) set
  assumes subspace s shows closed s
proof−
  have dim s ≤ card (UNIV :: 'n set) using dim-subset-univ by auto
  then obtain d::'n set where t: card d = dim s
   using closed-subspace-lemma by auto
```

let *?t* = {*x*::*real*^'*n*. ∀ *i*. *i* ∉ *d* ⟶ *x*$*i* = *0*}
obtain *f* **where** *f*:*linear f f* ' *?t* = *s inj-on f ?t*
using *subspace-isomorphism*[*OF subspace-substandard*[*of* λ*i*. *i* ∉ *d*] *assms*]
using *dim-substandard*[*of d*] **and** *t* **by** *auto*
have ∀ *x*∈*?t*. *f x* = *0* ⟶ *x* = *0* **using** *linear-0*[*OF f*(*1*)] **using** *f*(*3*)[*unfolded inj-on-def*]
by(*erule-tac x=0* **in** *ballE*) *auto*
moreover have *closed ?t* **using** *closed-substandard* .
moreover have *subspace ?t* **using** *subspace-substandard* .
ultimately show *?thesis* **using** *closed-injective-image-subspace*[*of ?t f*]
unfolding *f*(*2*) **using** *f*(*1*) **by** *auto*
qed

lemma *complete-subspace*:
subspace s ==> *complete s*
using *complete-eq-closed closed-subspace*
by *auto*

lemma *dim-closure*:
dim(*closure s*) = *dim s* (**is** *?dc* = *?d*)
proof−
have *?dc* ≤ *?d* **using** *closure-minimal*[*OF span-inc*, *of s*]
using *closed-subspace*[*OF subspace-span*, *of s*]
using *dim-subset*[*of closure s span s*] **unfolding** *dim-span* **by** *auto*
thus *?thesis* **using** *dim-subset*[*OF closure-subset*, *of s*] **by** *auto*
qed

Affine transformations of intervals.

lemma *affinity-inverses*:
assumes *m0*: *m* ≠ (*0*::'*a*::*field*)
shows (λ*x*. *m* ∗*s x* + *c*) *o* (λ*x*. *inverse*(*m*) ∗*s x* + (−(*inverse*(*m*) ∗*s c*))) = *id*
(λ*x*. *inverse*(*m*) ∗*s x* + (−(*inverse*(*m*) ∗*s c*))) *o* (λ*x*. *m* ∗*s x* + *c*) = *id*
using *m0*
apply (*auto simp add*: *expand-fun-eq vector-add-ldistrib vector-smult-assoc*)
by (*simp add*: *vector-smult-lneg*[*symmetric*] *vector-smult-assoc vector-sneg-minus1*[*symmetric*])

lemma *real-affinity-le*:
0 < (*m*::'*a*::*ordered-field*) ==> (*m* ∗ *x* + *c* ≤ *y* ⟷ *x* ≤ *inverse*(*m*) ∗ *y* + −(*c* / *m*))
by (*simp add*: *field-simps inverse-eq-divide*)

lemma *real-le-affinity*:
0 < (*m*::'*a*::*ordered-field*) ==> (*y* ≤ *m* ∗ *x* + *c* ⟷ *inverse*(*m*) ∗ *y* + −(*c* / *m*) ≤ *x*)
by (*simp add*: *field-simps inverse-eq-divide*)

lemma *real-affinity-lt*:
0 < (*m*::'*a*::*ordered-field*) ==> (*m* ∗ *x* + *c* < *y* ⟷ *x* < *inverse*(*m*) ∗ *y* + −(*c* / *m*))

by (*simp add*: *field-simps inverse-eq-divide*)

lemma *real-lt-affinity*:
$0 < (m::'a::ordered\text{-}field) ==> (y < m * x + c \longleftrightarrow inverse(m) * y + -(c\ /\ m) < x)$
by (*simp add*: *field-simps inverse-eq-divide*)

lemma *real-affinity-eq*:
$(m::'a::ordered\text{-}field) \neq 0 ==> (m * x + c = y \longleftrightarrow x = inverse(m) * y + -(c\ /\ m))$
by (*simp add*: *field-simps inverse-eq-divide*)

lemma *real-eq-affinity*:
$(m::'a::ordered\text{-}field) \neq 0 ==> (y = m * x + c \longleftrightarrow inverse(m) * y + -(c\ /\ m) = x)$
by (*simp add*: *field-simps inverse-eq-divide*)

lemma *vector-affinity-eq*:
assumes *m0*: $(m::'a::field) \neq 0$
shows $m *s x + c = y \longleftrightarrow x = inverse\ m *s y + -(inverse\ m *s c)$
proof
assume *h*: $m *s x + c = y$
hence $m *s x = y - c$ **by** (*simp add*: *ring-simps*)
hence $inverse\ m *s (m *s x) = inverse\ m *s (y - c)$ **by** *simp*
then show $x = inverse\ m *s y + - (inverse\ m *s c)$
using *m0* **by** (*simp add*: *vector-smult-assoc vector-ssub-ldistrib*)
next
assume *h*: $x = inverse\ m *s y + - (inverse\ m *s c)$
show $m *s x + c = y$ **unfolding** *h diff-minus*[*symmetric*]
using *m0* **by** (*simp add*: *vector-smult-assoc vector-ssub-ldistrib*)
qed

lemma *vector-eq-affinity*:
$(m::'a::field) \neq 0 ==> (y = m *s x + c \longleftrightarrow inverse(m) *s y + -(inverse(m) *s c) = x)$
using *vector-affinity-eq*[**where** *m*=*m* **and** *x*=*x* **and** *y*=*y* **and** *c*=*c*]
by *metis*

lemma *image-affinity-interval*: **fixes** *m*::*real*
fixes *a b c* :: *real^'n*::*finite*
shows $(\lambda x.\ m *s x + c)\ `\ \{a\ ..\ b\} =$
(if $\{a\ ..\ b\} = \{\}$ *then* $\{\}$
else (if $0 \leq m$ *then* $\{m *s a + c\ ..\ m *s b + c\}$
else $\{m *s b + c\ ..\ m *s a + c\}$*))*
proof(*cases m=0*)
{ fix *x* **assume** $x \leq c\ c \leq x$
hence *x=c* **unfolding** *vector-less-eq-def* **and** *Cart-eq* **by** (*auto intro*: *order-antisym*)
}
moreover case *True*

moreover have $c \in \{m *s a + c..m *s b + c\}$ **unfolding** *True* **by**(*auto simp add*: *vector-less-eq-def*)
ultimately show *?thesis* **by** *auto*
next
case *False*
{ fix y **assume** $a \leq y\ y \leq b\ m > 0$
hence $m *s a + c \leq m *s y + c\ \ m *s y + c \leq m *s b + c$
unfolding *vector-less-eq-def* **by**(*auto simp add*: *vector-smult-component vector-add-component*)
} moreover
{ fix y **assume** $a \leq y\ y \leq b\ m < 0$
hence $m *s b + c \leq m *s y + c\ \ m *s y + c \leq m *s a + c$
unfolding *vector-less-eq-def* **by**(*auto simp add*: *vector-smult-component vector-add-component mult-left-mono-neg elim*!:*ballE*)
} moreover
{ fix y **assume** $m > 0\ \ m *s a + c \leq y\ \ y \leq m *s b + c$
hence $y \in (\lambda x.\ m *s x + c)\ `\ \{a..b\}$
unfolding *image-iff Bex-def mem-interval vector-less-eq-def*
apply(*auto simp add*: *vector-smult-component vector-add-component vector-minus-component vector-smult-assoc pth-3*[*symmetric*]
intro!: *exI*[**where** x=(*1 / m*) *s* (*y − c*)])
by(*auto simp add*: *pos-le-divide-eq pos-divide-le-eq real-mult-commute diff-le-iff*)
} moreover
{ fix y **assume** $m *s b + c \leq y\ y \leq m *s a + c\ m < 0$
hence $y \in (\lambda x.\ m *s x + c)\ `\ \{a..b\}$
unfolding *image-iff Bex-def mem-interval vector-less-eq-def*
apply(*auto simp add*: *vector-smult-component vector-add-component vector-minus-component vector-smult-assoc pth-3*[*symmetric*]
intro!: *exI*[**where** x=(*1 / m*) *s* (*y − c*)])
by(*auto simp add*: *neg-le-divide-eq neg-divide-le-eq real-mult-commute diff-le-iff*)
}
ultimately show *?thesis* **using** *False* **by** *auto*
qed

64.41 Banach fixed point theorem (not really topological...)

lemma *banach-fix*:
assumes *s*:*complete s*\ $s \neq \{\}$ **and** *c*:$0 \leq c\ c < 1$ **and** f:$(f\ `\ s) \subseteq s$ **and**
lipschitz:$\forall x{\in}s.\ \forall y{\in}s.\ dist\ (f\ x)\ (f\ y) \leq c * dist\ x\ y$
shows $\exists!\ x{\in}s.\ (f\ x = x)$
proof−
have $1 - c > 0$ **using** c **by** *auto*

from *s*(*2*) **obtain** *z0* **where** $z0 \in s$ **by** *auto*
def $z \equiv \lambda\ n{::}nat.\ fun\text{-}pow\ n\ f\ z0$
{ fix $n{::}nat$
have $z\ n \in s$ **unfolding** *z-def*
proof(*induct n*) **case** *0* **thus** *?case* **using** ‹*z0* ∈*s*› **by** *auto*
next case *Suc* **thus** *?case* **using** f **by** *auto* **qed }**

note *z-in-s* = *this*

def $d \equiv dist\ (z\ 0)\ (z\ 1)$

have *fzn*:$\bigwedge n.\ f\ (z\ n) = z\ (Suc\ n)$ **unfolding** *z-def* **by** *auto*
{ fix *n*::*nat*
have $dist\ (z\ n)\ (z\ (Suc\ n)) \leq (c\ \hat{}\ n) * d$
proof(*induct n*)
case *0* **thus** *?case* **unfolding** *d-def* **by** *auto*
next
case (*Suc m*)
hence $c * dist\ (z\ m)\ (z\ (Suc\ m)) \leq c\ \hat{}\ Suc\ m * d$
using ‹$0 \leq c$› **using** *mult-mono1-class.mult-mono1*[*of dist* (*z m*) (*z* (*Suc m*)) *c* ^ *m* * *d c*] **by** *auto*
thus *?case* **using** *lipschitz*[*THEN bspec*[**where** *x*=*z m*], *OF z-in-s*, *THEN bspec*[**where** *x*=*z* (*Suc m*)], *OF z-in-s*]
unfolding *fzn* **and** *mult-le-cancel-left* **by** *auto*
qed
} note *cf-z* = *this*

{ fix *n m*::*nat*
have $(1 - c) * dist\ (z\ m)\ (z\ (m+n)) \leq (c\ \hat{}\ m) * d * (1 - c\ \hat{}\ n)$
proof(*induct n*)
case *0* **show** *?case* **by** *auto*
next
case (*Suc k*)
have $(1 - c) * dist\ (z\ m)\ (z\ (m + Suc\ k)) \leq (1 - c) * (dist\ (z\ m)\ (z\ (m + k)) + dist\ (z\ (m + k))\ (z\ (Suc\ (m + k))))$
using *dist-triangle* **and** *c* **by**(*auto simp add*: *dist-triangle*)
also have $\ldots \leq (1 - c) * (dist\ (z\ m)\ (z\ (m + k)) + c\ \hat{}\ (m + k) * d)$
using *cf-z*[*of m* + *k*] **and** *c* **by** *auto*
also have $\ldots \leq c\ \hat{}\ m * d * (1 - c\ \hat{}\ k) + (1 - c) * c\ \hat{}\ (m + k) * d$
using *Suc* **by** (*auto simp add*: *ring-simps*)
also have $\ldots = (c\ \hat{}\ m) * (d * (1 - c\ \hat{}\ k) + (1 - c) * c\ \hat{}\ k * d)$
unfolding *power-add* **by** (*auto simp add*: *ring-simps*)
also have $\ldots \leq (c\ \hat{}\ m) * d * (1 - c\ \hat{}\ Suc\ k)$
using *c* **by** (*auto simp add*: *ring-simps dist-pos-le*)
finally show *?case* **by** *auto*
qed
} note *cf-z2* = *this*
{ fix *e*::*real* **assume** *e*>*0*
hence $\exists N.\ \forall m\ n.\ N \leq m \wedge N \leq n \longrightarrow dist\ (z\ m)\ (z\ n) < e$
proof(*cases d* = *0*)
case *True*
hence $\bigwedge n.\ z\ n = z0$ **using** *cf-z2*[*of 0*] **and** *c* **unfolding** *z-def* **by** (*auto simp add*: *pos-prod-le*[*OF* ‹*1* − *c* > *0*›] *dist-le-0*)
thus *?thesis* **using** ‹*e*>*0*› **by** *auto*
next
case *False* **hence** *d*>*0* **unfolding** *d-def* **using** *dist-pos-le*[*of z 0 z 1*]

```
      by (metis False d-def real-less-def)
     hence 0 < e * (1 - c) / d using ‹e>0› and ‹1-c>0›
      using divide-pos-pos[of e * (1 - c) d] and mult-pos-pos[of e 1 - c] by auto
    then obtain N where N:c ^ N < e * (1 - c) / d using real-arch-pow-inv[of
e * (1 - c) / d c] and c by auto
     { fix m n::nat assume m>n and as:m≥N n≥N
       have *:c ^ n ≤ c ^ N using ‹n≥N› and c using power-decreasing[OF
‹n≥N›, of c] by auto
      have 1 - c ^ (m - n) > 0 using c and power-strict-mono[of c 1 m - n]
using ‹m>n› by auto
      hence **:d * (1 - c ^ (m - n)) / (1 - c) > 0
       using real-mult-order[OF ‹d>0›, of 1 - c ^ (m - n)]
       using divide-pos-pos[of d * (1 - c ^ (m - n)) 1 - c]
       using ‹0 < 1 - c› by auto

      have dist (z m) (z n) ≤ c ^ n * d * (1 - c ^ (m - n)) / (1 - c)
          using cf-z2[of n m - n] and ‹m>n› unfolding pos-le-divide-eq[OF
‹1-c>0›]
       by (auto simp add: real-mult-commute dist-sym)
      also have ... ≤ c ^ N * d * (1 - c ^ (m - n)) / (1 - c)
       using mult-right-mono[OF * order-less-imp-le[OF **]]
       unfolding real-mult-assoc by auto
      also have ... < (e * (1 - c) / d) * d * (1 - c ^ (m - n)) / (1 - c)
       using mult-strict-right-mono[OF N **] unfolding real-mult-assoc by auto
      also have ... = e * (1 - c ^ (m - n)) using c and ‹d>0› and ‹1 - c >
0› by auto
      also have ... ≤ e using c and ‹1 - c ^ (m - n) > 0› and ‹e>0› using
mult-right-le-one-le[of e 1 - c ^ (m - n)] by auto
      finally have  dist (z m) (z n) < e by auto
     } note * = this
     { fix m n::nat assume as:N≤m N≤n
      hence dist (z n) (z m) < e
      proof(cases n = m)
       case True thus ?thesis using ‹e>0› by auto
      next
        case False thus ?thesis using as and *[of n m] *[of m n] unfolding
nat-neq-iff by (auto simp add: dist-sym)
      qed }
     thus ?thesis by auto
   qed
  }
  hence cauchy z unfolding cauchy-def by auto
  then obtain x where x∈s and x:(z ---> x) sequentially using s(1)[unfolded
compact-def complete-def, THEN spec[where x=z]] and z-in-s by auto

  def e ≡ dist (f x) x
  have e = 0 proof(rule ccontr)
   assume e ≠ 0 hence e>0 unfolding e-def using dist-pos-le[of f x x]
    by (metis dist-eq-0 dist-nz dist-sym e-def)
```

then obtain N **where** $N{:}\forall\, n{\geq}N.\ dist\ (z\ n)\ x < e\ /\ 2$
using $x[unfolded\ Lim\text{-}sequentially,\ THEN\ spec[\textbf{where}\ x{=}e/2]]$ **by** *auto*
hence $N'{:}dist\ (z\ N)\ x < e\ /\ 2$ **by** *auto*

have $*{:}c * dist\ (z\ N)\ x \leq dist\ (z\ N)\ x$ **unfolding** *mult-le-cancel-right2*
using *dist-pos-le*[*of z N x*] **and** *c*
by (*metis dist-eq-0 dist-nz dist-sym order-less-asym real-less-def*)
have $dist\ (f\ (z\ N))\ (f\ x) \leq c * dist\ (z\ N)\ x$ **using** *lipschitz*[*THEN bspec*[**where** $x{=}z\ N$], *THEN bspec*[**where** $x{=}x$]]
using *z-in-s*[*of N*] ‹$x{\in}s$› **using** *c* **by** *auto*
also have $\ldots < e\ /\ 2$ **using** N' **and** *c* **using** $*$ **by** *auto*
finally show *False* **unfolding** *fzn*
using N[*THEN spec*[**where** $x{=}Suc\ N$]] **and** *dist-triangle-half-r*[*of z* (*Suc N*) *f x e x*]
unfolding *e-def* **by** *auto*
qed
hence $f\ x = x$ **unfolding** *e-def* **and** *dist-eq-0* **by** *auto*
moreover
{ fix y **assume** $f\ y = y\ y{\in}s$
hence $dist\ x\ y \leq c * dist\ x\ y$ **using** *lipschitz*[*THEN bspec*[**where** $x{=}x$], *THEN bspec*[**where** $x{=}y$]]
using ‹$x{\in}s$› **and** ‹$f\ x = x$› **by** *auto*
hence $dist\ x\ y = 0$ **unfolding** *mult-le-cancel-right1*
using *c* **and** *dist-pos-le*[*of x y*] **by** *auto*
hence $y = x$ **unfolding** *dist-eq-0* **by** *auto*
}
ultimately show *?thesis* **unfolding** *Bex1-def* **using** ‹$x{\in}s$› **by** *blast+*
qed

64.42 Edelstein fixed point theorem.

lemma *edelstein-fix*:
assumes $s{:}compact\ s\ s \neq \{\}$ **and** $gs{:}(g\ `\ s) \subseteq s$
and $dist{:}\forall\, x{\in}s.\ \forall\, y{\in}s.\ x \neq y \longrightarrow dist\ (g\ x)\ (g\ y) < dist\ x\ y$
shows $\exists!\ x{::}real\hat{}'a{::}finite{\in}s.\ g\ x = x$
proof(*cases* $\exists\, x{\in}s.\ g\ x \neq x$)
obtain x **where** $x{\in}s$ **using** $s(2)$ **by** *auto*
case *False* **hence** $g{:}\forall\, x{\in}s.\ g\ x = x$ **by** *auto*
{ fix y **assume** $y{\in}s$
hence $x = y$ **using** ‹$x{\in}s$› **and** *dist*[*THEN bspec*[**where** $x{=}x$], *THEN bspec*[**where** $x{=}y$]]
unfolding g[*THEN bspec*[**where** $x{=}x$], *OF* ‹$x{\in}s$›]
unfolding g[*THEN bspec*[**where** $x{=}y$], *OF* ‹$y{\in}s$›] **by** *auto* **}**
thus *?thesis* **unfolding** *Bex1-def* **using** ‹$x{\in}s$› **and** g **by** *blast+*
next
case *True*
then obtain x **where** [*simp*]:$x{\in}s$ **and** $g\ x \neq x$ **by** *auto*
{ fix $x\ y$ **assume** $x \in s\ y \in s$
hence $dist\ (g\ x)\ (g\ y) \leq dist\ x\ y$

using *dist*[*THEN bspec*[**where** *x*=*x*], *THEN bspec*[**where** *x*=*y*]] **by** *auto* **}**
note *dist'* = *this*
def *y* ≡ *g x*
have [*simp*]:*y*∈*s* **unfolding** *y-def* **using** *gs*[*unfolded image-subset-iff*] **and** ‹*x*∈*s*› **by** *blast*
def *f* ≡ λ *n*. *fun-pow n g*
have [*simp*]:⋀*n z*. *g* (*f n z*) = *f* (*Suc n*) *z* **unfolding** *f-def* **by** *auto*
have [*simp*]:⋀*z*. *f 0 z* = *z* **unfolding** *f-def* **by** *auto*
{ fix *n*::*nat* **and** *z* **assume** *z*∈*s*
have *f n z* ∈ *s* **unfolding** *f-def*
proof(*induct n*)
case *0* **thus** *?case* **using** ‹*z*∈*s*› **by** *simp*
next
case (*Suc n*) **thus** *?case* **using** *gs*[*unfolded image-subset-iff*] **by** *auto*
qed **}** **note** *fs* = *this*
{ fix *m n* ::*nat* **assume** *m*≤*n*
fix *w z* **assume** *w*∈*s z*∈*s*
have *dist* (*f n w*) (*f n z*) ≤ *dist* (*f m w*) (*f m z*) **using** ‹*m*≤*n*›
proof(*induct n*)
case *0* **thus** *?case* **by** *auto*
next
case (*Suc n*)
thus *?case* **proof**(*cases m*≤*n*)
case *True* **thus** *?thesis* **using** *Suc*(*1*)
using *dist'*[*OF fs fs*, *OF* ‹*w*∈*s*› ‹*z*∈*s*›, *of n n*] **by** *auto*
next
case *False* **hence** *mn*:*m* = *Suc n* **using** *Suc*(*2*) **by** *simp*
show *?thesis* **unfolding** *mn* **by** *auto*
qed
qed **}** **note** *distf* = *this*

def *h* ≡ λ*n*. *pastecart* (*f n x*) (*f n y*)
let *?s2* = {*pastecart x y* |*x y*. *x* ∈ *s* ∧ *y* ∈ *s*}
obtain *l r* **where** *l*∈*?s2* **and** *r*:∀ *m n*. *m* < *n* ⟶ *r m* < *r n* **and** *lr*:((*h* ∘ *r*) −−−> *l*) *sequentially*
using *compact-pastecart*[*OF s*(*1*) *s*(*1*), *unfolded compact-def*, *THEN spec*[**where** *x*=*h*]] **unfolding** *h-def*
using *fs*[*OF* ‹*x*∈*s*›] **and** *fs*[*OF* ‹*y*∈*s*›] **by** *blast*
def *a* ≡ *fstcart l* **def** *b* ≡ *sndcart l*
have *lab*:*l* = *pastecart a b* **unfolding** *a-def b-def* **and** *pastecart-fst-snd* **by** *simp*
have [*simp*]:*a*∈*s b*∈*s* **unfolding** *a-def b-def* **using** ‹*l*∈*?s2*› **by** *auto*

have *continuous-on UNIV fstcart* **and** *continuous-on UNIV sndcart*
using *linear-continuous-on* **using** *linear-fstcart* **and** *linear-sndcart* **by** *auto*
hence *lima*:((*fstcart* ∘ (*h* ∘ *r*)) −−−> *a*) *sequentially* **and** *limb*:((*sndcart* ∘ (*h* ∘ *r*)) −−−> *b*) *sequentially*
unfolding *atomize-conj* **unfolding** *continuous-on-sequentially*
apply(*erule-tac x*=*h* ∘ *r* **in** *allE*) **apply**(*erule-tac x*=*h* ∘ *r* **in** *allE*) **using** *lr*
unfolding *o-def* **and** *h-def a-def b-def* **by** *auto*

{ fix *n*::*nat*
have ∗:⋀*fx fy x y*. *dist fx fy* ≤ *dist x y* ⟹ ¬ (*dist* (*fx* − *fy*) (*a* − *b*) < *dist a b* − *dist x y*) **unfolding** *dist-def* **by** *norm*
{ fix *x y* ::*real*^'*a*
have *dist* (−*x*) (−*y*) = *dist x y* **unfolding** *dist-def*
using *norm-minus-cancel*[*of x* − *y*] **by** (*auto simp add*: *uminus-add-conv-diff*)
} **note** ∗∗ = *this*

{ assume *as*:*dist a b* > *dist* (*f n x*) (*f n y*)
then obtain *Na Nb* **where** ∀ *m*≥*Na*. *dist* (*f* (*r m*) *x*) *a* < (*dist a b* − *dist* (*f n x*) (*f n y*)) / *2*
and ∀ *m*≥*Nb*. *dist* (*f* (*r m*) *y*) *b* < (*dist a b* − *dist* (*f n x*) (*f n y*)) / *2*
using *lima limb* **unfolding** *h-def Lim-sequentially* **by** (*fastsimp simp del*: *less-divide-eq-number-of1*)
hence *dist* (*f* (*r* (*Na* + *Nb* + *n*)) *x* − *f* (*r* (*Na* + *Nb* + *n*)) *y*) (*a* − *b*) < *dist a b* − *dist* (*f n x*) (*f n y*)
apply(*erule-tac x*=*Na*+*Nb*+*n* **in** *allE*)
apply(*erule-tac x*=*Na*+*Nb*+*n* **in** *allE*) **apply** *simp*
using *dist-triangle-add-half*[*of a f* (*r* (*Na* + *Nb* + *n*)) *x dist a b* − *dist* (*f n x*) (*f n y*)
−*b* − *f* (*r* (*Na* + *Nb* + *n*)) *y*]
unfolding ∗∗ **unfolding** *group-simps*(*12*) **by** (*auto simp add*: *dist-sym*)
moreover
have *dist* (*f* (*r* (*Na* + *Nb* + *n*)) *x* − *f* (*r* (*Na* + *Nb* + *n*)) *y*) (*a* − *b*) ≥ *dist a b* − *dist* (*f n x*) (*f n y*)
using *distf*[*of n r* (*Na*+*Nb*+*n*), *OF* - ‹*x*∈*s*› ‹*y*∈*s*›]
using *monotone-bigger*[*OF r*, *of Na*+*Nb*+*n*]
using ∗[*of f* (*r* (*Na* + *Nb* + *n*)) *x f* (*r* (*Na* + *Nb* + *n*)) *y f n x f n y*] **by** *auto*
ultimately have *False* **by** *simp*
}
hence *dist a b* ≤ *dist* (*f n x*) (*f n y*) **by**(*rule ccontr*)*auto* **}**
note *ab-fn* = *this*

have [*simp*]:*a* = *b* **proof**(*rule ccontr*)
def *e* ≡ *dist a b* − *dist* (*g a*) (*g b*)
assume *a*≠*b* **hence** *e* > *0* **unfolding** *e-def* **using** *dist* **by** *fastsimp*
hence ∃ *n*. *dist* (*f n x*) *a* < *e*/*2* ∧ *dist* (*f n y*) *b* < *e*/*2*
using *lima limb* **unfolding** *Lim-sequentially*
apply (*auto elim*!: *allE*[**where** *x*=*e*/*2*]) **apply**(*rule-tac x*=*r* (*max N Na*) **in** *exI*) **unfolding** *h-def* **by** *fastsimp*
then obtain *n* **where** *n*:*dist* (*f n x*) *a* < *e*/*2* ∧ *dist* (*f n y*) *b* < *e*/*2* **by** *auto*
have *dist* (*f* (*Suc n*) *x*) (*g a*) ≤ *dist* (*f n x*) *a*
using *dist*[*THEN bspec*[**where** *x*=*f n x*], *THEN bspec*[**where** *x*=*a*]] **and** *fs* **by** *auto*
moreover have *dist* (*f* (*Suc n*) *y*) (*g b*) ≤ *dist* (*f n y*) *b*
using *dist*[*THEN bspec*[**where** *x*=*f n y*], *THEN bspec*[**where** *x*=*b*]] **and** *fs* **by** *auto*

ultimately have *dist* (*f* (*Suc n*) *x*) (*g a*) + *dist* (*f* (*Suc n*) *y*) (*g b*) < *e* **using** *n* **by** *auto*
thus *False* **unfolding** *e-def* **using** *ab-fn*[*of Suc n*] **by** *norm*
qed

have [*simp*]:⋀*n*. *f* (*Suc n*) *x* = *f n y* **unfolding** *f-def y-def* **by**(*induct-tac n*)*auto*
{ fix *x y* **assume** *x*∈*s y*∈*s* **moreover**
fix *e*::*real* **assume** *e*>*0* **ultimately**
have *dist y x* < *e* ⟶ *dist* (*g y*) (*g x*) < *e* **using** *dist* **by** *fastsimp* **}**
hence *continuous-on s g* **unfolding** *continuous-on-def* **by** *auto*

hence ((*sndcart* ∘ *h* ∘ *r*) −−−> *g a*) *sequentially* **unfolding** *continuous-on-sequentially*
apply (*rule allE*[**where** *x*=λ*n*. (*fstcart* ∘ *h* ∘ *r*) *n*]) **apply** (*erule ballE*[**where** *x*=*a*])
using *lima* **unfolding** *h-def o-def* **using** *fs*[*OF* ‹*x*∈*s*›] **by** (*auto simp add*: *y-def*)
hence *g a* = *a* **using** *Lim-unique*[*OF trivial-limit-sequentially limb*, *of g a*]
unfolding ‹*a*=*b*› **and** *o-assoc* **by** *auto*
moreover
{ fix *x* **assume** *x*∈*s g x* = *x x*≠*a*
hence *False* **using** *dist*[*THEN bspec*[**where** *x*=*a*], *THEN bspec*[**where** *x*=*x*]]
using ‹*g a* = *a*› **and** ‹*a*∈*s*› **by** *auto* **}**
ultimately show ∃!*x*∈*s*. *g x* = *x* **unfolding** *Bex1-def* **using** ‹*a*∈*s*› **by** *blast*
qed

end

65 Univ-Poly: Univariate Polynomials

theory *Univ-Poly*
imports *Main*
begin

Application of polynomial as a function.

primrec (**in** *semiring-0*) *poly* :: *'a list* => *'a* => *'a* **where**
poly-Nil: *poly* [] *x* = *0*
| *poly-Cons*: *poly* (*h*#*t*) *x* = *h* + *x* ∗ *poly t x*

65.1 Arithmetic Operations on Polynomials

addition

primrec (**in** *semiring-0*) *padd* :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixl** +++ *65*)
where
padd-Nil: [] +++ *l2* = *l2*
| *padd-Cons*: (*h*#*t*) +++ *l2* = (*if l2* = [] *then h*#*t*
else (*h* + *hd l2*)#(*t* +++ *tl l2*))

Multiplication by a constant

primrec (**in** *semiring-0*) *cmult* :: *'a* ⇒ *'a list* ⇒ *'a list* (**infixl** %∗ *70*) **where**
cmult-Nil: *c* %∗ [] = []
| *cmult-Cons*: *c* %∗ (*h*#*t*) = (*c* ∗ *h*)#(*c* %∗ *t*)

Multiplication by a polynomial

primrec (**in** *semiring-0*) *pmult* :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixl** ∗∗∗ *70*)
where
pmult-Nil: [] ∗∗∗ *l2* = []
| *pmult-Cons*: (*h*#*t*) ∗∗∗ *l2* = (*if t* = [] *then h* %∗ *l2*
else (*h* %∗ *l2*) +++ ((*0*) # (*t* ∗∗∗ *l2*)))

Repeated multiplication by a polynomial

primrec (**in** *semiring-0*) *mulexp* :: *nat* ⇒ *'a list* ⇒ *'a list* ⇒ *'a list* **where**
mulexp-zero: *mulexp 0 p q* = *q*
| *mulexp-Suc*: *mulexp* (*Suc n*) *p q* = *p* ∗∗∗ *mulexp n p q*

Exponential

primrec (**in** *semiring-1*) *pexp* :: *'a list* ⇒ *nat* ⇒ *'a list* (**infixl** %^ *80*) **where**
pexp-0: *p* %^ *0* = [*1*]
| *pexp-Suc*: *p* %^ (*Suc n*) = *p* ∗∗∗ (*p* %^ *n*)

Quotient related value of dividing a polynomial by x + a

primrec (**in** *field*) *pquot* :: *'a list* ⇒ *'a* ⇒ *'a list* **where**
pquot-Nil: *pquot* [] *a*= []
| *pquot-Cons*: *pquot* (*h*#*t*) *a* = (*if t* = [] *then* [*h*]
else (*inverse*(*a*) ∗ (*h* − *hd*(*pquot t a*)))#(*pquot t a*))

normalization of polynomials (remove extra 0 coeff)

primrec (**in** *semiring-0*) *pnormalize* :: *'a list* ⇒ *'a list* **where**
pnormalize-Nil: *pnormalize* [] = []
| *pnormalize-Cons*: *pnormalize* (*h*#*p*) = (*if* ((*pnormalize p*) = [])
then (*if* (*h* = *0*) *then* [] *else* [*h*])
else (*h*#(*pnormalize p*)))

definition (**in** *semiring-0*) *pnormal p* = ((*pnormalize p* = *p*) ∧ *p* ≠ [])
definition (**in** *semiring-0*) *nonconstant p* = (*pnormal p* ∧ (∀ *x*. *p* ≠ [*x*]))

Other definitions

definition (**in** *ring-1*)
poly-minus :: *'a list* => *'a list* (−− - [*80*] *80*) **where**
−− *p* = (− *1*) %∗ *p*

definition (**in** *semiring-0*)
divides :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** *divides 70*) **where**
[*code del*]: *p1 divides p2* = (∃ *q*. *poly p2* = *poly*(*p1* ∗∗∗ *q*))

— order of a polynomial
definition (**in** *ring-1*) *order* :: *'a* => *'a list* => *nat* **where**
order a p = (*SOME n*. ([−*a*, *1*] %^ *n*) *divides p* &

$$\sim (([-a,\ 1]\ \%\hat{}\ (Suc\ n))\ divides\ p))$$

— 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 *recpower-semiring* = *semiring* + *recpower*
class *recpower-semiring-1* = *semiring-1* + *recpower*
class *recpower-semiring-0* = *semiring-0* + *recpower*
class *recpower-ring* = *ring* + *recpower*

class *recpower-ring-1* = *ring-1* + *recpower*
subclass (**in** *recpower-ring-1*) *recpower-ring* **..**
class *recpower-comm-semiring-1* = *recpower* + *comm-semiring-1*
class *recpower-comm-ring-1* = *recpower* + *comm-ring-1*
subclass (**in** *recpower-comm-ring-1*) *recpower-comm-semiring-1* **..**
class *recpower-idom* = *recpower* + *idom*
subclass (**in** *recpower-idom*) *recpower-comm-ring-1* **..**
class *idom-char-0* = *idom* + *ring-char-0*
class *recpower-idom-char-0* = *recpower* + *idom-char-0*
subclass (**in** *recpower-idom-char-0*) *recpower-idom* **..**

lemma (**in** *recpower-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

65.2 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*: $\exists\, q\ r.\ h\#t = [r] +\!\!+\!\!+ [-a,\ 1] \ast\!\ast\!\ast\ q$
by (*cut-tac* $t = t$ **and** $a = a$ **in** *lemma-poly-linear-rem*, *auto*)

lemma (**in** *comm-ring-1*) *poly-linear-divides*: $(poly\ p\ a = 0) = ((p = []) \mid (\exists\, q.\ p = [-a,\ 1] \ast\!\ast\!\ast\ q))$
proof−
{assume *p*: $p = []$ **hence** *?thesis* **by** *simp***}**
moreover
{fix *x xs* **assume** *p*: $p = x\#xs$
{fix *q* **assume** $p = [-a,\ 1] \ast\!\ast\!\ast\ q$ **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*: $x\#xs = [r] +\!\!+\!\!+ [-\ a,\ 1] \ast\!\ast\!\ast\ q$ **by** *blast*
have $r = 0$ **using** *p0* **by** (*simp only*: *p qr poly-mult poly-add*) *simp*
hence $\exists\, q.\ p = [-\ a,\ 1] \ast\!\ast\!\ast\ q$ **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*]: $\forall\, h\ k\ a.\ length\ (k\ \%\ast\ p +\!\!+\!\!+\ (h\ \#\ (a\ \%\ast\ p))) = Suc\ (length\ p)$
by (*induct p*, *auto*)

lemma (**in** *semiring-0*) *lemma-poly-length-mult2*[*simp*]: $\forall\, h\ k.\ length\ (k\ \%\ast\ p +\!\!+\!\!+\ (h\ \#\ p)) = Suc\ (length\ p)$
by (*induct p*, *auto*)

lemma (**in** *ring-1*) *poly-length-mult*[*simp*]: $length([-a,1] \ast\!\ast\!\ast\ q) = Suc\ (length\ q)$
by *auto*

65.3 Polynomial length

lemma (**in** *semiring-0*) *poly-cmult-length*[*simp*]: $length\ (a\ \%\ast\ p) = length\ p$
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*]: $length([a,b] \ast\!\ast\!\ast\ p) = Suc$

(*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* (λ*n*. *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** *recpower-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** *recpower-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** *recpower-comm-semiring-1*) *poly-divides-exp*: $m \leq n$ ==> (*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** *recpower-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** *recpower-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** *recpower-semiring-1*)
zero-power-iff: *0 ^ n = (if n = 0 then 1 else 0)*
by (*induct n*, *simp-all add*: *power-Suc*)

lemma (**in** *recpower-idom-char-0*) *poly-order-exists*:
assumes *lp*: *length p = d* **and** *p0*: *poly p ≠ poly []*
shows ∃ *n*. ([−*a*, *1*] % ^ *n*) *divides p* & ~(([−*a*, *1*] % ^ (*Suc n*)) *divides p*)
proof−
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** *recpower-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** *recpower-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** *recpower-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** *recpower-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** *recpower-idom-char-0*) *order-unique-lemma*: (*poly p ≠ poly* [] & ([−*a*, *1*] % ^ *n*) *divides p* &
~(([−*a*, *1*] % ^ (*Suc n*)) *divides p*))
==> (*n = order a p*)
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** *recpower-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** *recpower-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** *recpower-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*::{*recpower-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** *recpower-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** *recpower-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** *recpower-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*
apply (*cases pnormalize p* = [], *auto*)
by (*cases c* = *0*, *auto*)

lemma (**in** *semiring-0*) *pnormal-last-nonzero*: *pnormal p* ==> *last p* ≠ *0*
proof(*induct p*)
case *Nil* **thus** *?case* **by** (*simp add*: *pnormal-def*)
next
case (*Cons a as*) **thus** *?case*
apply (*simp add*: *pnormal-def*)
apply (*cases pnormalize as* = [], *simp-all*)
apply (*cases as* = [], *simp-all*)
apply (*cases a*=*0*, *simp-all*)
apply (*cases a*=*0*, *simp-all*)
done
qed

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*
apply (*induct p*, *auto*)
apply (*case-tac p* = [], *auto*)
apply (*simp add*: *pnormal-def*)
by (*rule pnormal-cons*, *auto*)

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*
apply (*induct p*, *auto*)
apply (*case-tac p*, *auto*)+
done

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** *recpower-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*: $abs(x) \leq k \Longrightarrow abs(poly\ p\ (x::'a::\{ordered\text{-}idom\})) \leq poly\ (map\ abs\ p)\ k$
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

66 While-Combinator: A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

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

function (*tailrec*) *while* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where
while-unfold[*simp del*]: *while b c s* = (*if b s then while b c* (*c s*) *else s*)
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*: $!!s.\ P\ s ==> \neg\ b\ s ==> Q\ s$
and *wf*: $wf\ \{(t, s).\ P\ s \wedge b\ s \wedge t = c\ s\}$
shows $P\ s \Longrightarrow Q\ (while\ b\ c\ s)$
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* (λ*N*::*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

67 Word: Binary Words

theory *Word*
imports ~~/*src*/*HOL*/*Main*
begin

67.1 Auxilary Lemmas

lemma *max-le* [*intro*!]: [| *x* ≤ *z*; *y* ≤ *z* |] ==> *max x y* ≤ *z*
by (*simp add*: *max-def*)

lemma *max-mono*:
fixes *x* :: *'a*::*linorder*
assumes *mf*: *mono f*
shows *max* (*f x*) (*f y*) ≤ *f* (*max x y*)
proof −
from *mf* **and** *le-maxI1* [*of x y*]
have *fx*: *f x* ≤ *f* (*max x y*) **by** (*rule monoD*)
from *mf* **and** *le-maxI2* [*of y x*]
have *fy*: *f y* ≤ *f* (*max x y*) **by** (*rule monoD*)
from *fx* **and** *fy*
show *max* (*f x*) (*f y*) ≤ *f* (*max x y*) **by** *auto*
qed

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

lemma *int-nat-two-exp*: *2* ^ *k* = *int* (*2* ^ *k*)
by (*simp add*: *zpower-int* [*symmetric*])

67.2 Bits

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

primrec
  bitval :: *bit* => *nat*
where
  bitval **0** = *0*
| *bitval* **1** = *1*

consts
  bitnot :: *bit* => *bit*
  bitand :: *bit* => *bit* => *bit* (**infixr** *bitand 35*)
  bitor :: *bit* => *bit* => *bit* (**infixr** *bitor 30*)
  bitxor :: *bit* => *bit* => *bit* (**infixr** *bitxor 30*)

notation (*xsymbols*)
  bitnot ($\neg_b$ - [*40*] *40*) **and**
  bitand (**infixr** $\wedge_b$ *35*) **and**
  bitor (**infixr** $\vee_b$ *30*) **and**
  bitxor (**infixr** $\oplus_b$ *30*)

notation (*HTML* **output**)
  bitnot ($\neg_b$ - [*40*] *40*) **and**
  bitand (**infixr** $\wedge_b$ *35*) **and**
  bitor (**infixr** $\vee_b$ *30*) **and**
  bitxor (**infixr** $\oplus_b$ *30*)

primrec
  bitnot-zero: (*bitnot* **0**) = **1**
  bitnot-one : (*bitnot* **1**) = **0**

primrec
  bitand-zero: (**0** *bitand y*) = **0**
  bitand-one: (**1** *bitand y*) = *y*

primrec
  bitor-zero: (**0** *bitor y*) = *y*
  bitor-one: (**1** *bitor y*) = **1**

primrec
  bitxor-zero: (**0** *bitxor y*) = *y*
  bitxor-one: (**1** *bitxor y*) = (*bitnot y*)

lemma *bitnot-bitnot* [*simp*]: (*bitnot* (*bitnot b*)) = *b*
  by (*cases b*) *simp-all*

lemma *bitand-cancel* [*simp*]: (*b bitand b*) = *b*

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

lemma *bitor-cancel* [*simp*]: (*b bitor b*) = *b*
by (*cases b*) *simp-all*

lemma *bitxor-cancel* [*simp*]: (*b bitxor b*) = **0**
by (*cases b*) *simp-all*

67.3 Bit Vectors

First, a couple of theorems expressing case analysis and induction principles for bit vectors.

lemma *bit-list-cases*:
assumes *empty*: *w* = [] ==> *P w*
and *zero*: !!*bs*. *w* = **0** # *bs* ==> *P w*
and *one*: !!*bs*. *w* = **1** # *bs* ==> *P w*
shows *P w*
proof (*cases w*)
assume *w* = []
thus *?thesis* **by** (*rule empty*)
next
fix *b bs*
assume [*simp*]: *w* = *b* # *bs*
show *P w*
proof (*cases b*)
assume *b* = **0**
hence *w* = **0** # *bs* **by** *simp*
thus *?thesis* **by** (*rule zero*)
next
assume *b* = **1**
hence *w* = **1** # *bs* **by** *simp*
thus *?thesis* **by** (*rule one*)
qed
qed

lemma *bit-list-induct*:
assumes *empty*: *P* []
and *zero*: !!*bs*. *P bs* ==> *P* (**0**#*bs*)
and *one*: !!*bs*. *P bs* ==> *P* (**1**#*bs*)
shows *P w*
proof (*induct w*, *simp-all add*: *empty*)
fix *b bs*
assume *P bs*
then show *P* (*b*#*bs*)
by (*cases b*) (*auto intro*!: *zero one*)
qed

definition
bv-msb :: *bit list* => *bit* **where**

bv-msb w = (if w = [] then **0** *else hd w)*

definition
bv-extend :: [nat,bit,bit list]=>bit list **where**
bv-extend i b w = (replicate (i − length w) b) @ w

definition
bv-not :: bit list => bit list **where**
bv-not w = map bitnot w

lemma *bv-length-extend* [*simp*]: *length w ≤ i ==> length (bv-extend i b w) = i*
by (*simp add*: *bv-extend-def*)

lemma *bv-not-Nil* [*simp*]: *bv-not [] = []*
by (*simp add*: *bv-not-def*)

lemma *bv-not-Cons* [*simp*]: *bv-not (b#bs) = (bitnot b) # bv-not bs*
by (*simp add*: *bv-not-def*)

lemma *bv-not-bv-not* [*simp*]: *bv-not (bv-not w) = w*
by (*rule bit-list-induct* [*of* - *w*]) *simp-all*

lemma *bv-msb-Nil* [*simp*]: *bv-msb [] =* **0**
by (*simp add*: *bv-msb-def*)

lemma *bv-msb-Cons* [*simp*]: *bv-msb (b#bs) = b*
by (*simp add*: *bv-msb-def*)

lemma *bv-msb-bv-not* [*simp*]: *0 < length w ==> bv-msb (bv-not w) = (bitnot (bv-msb w))*
by (*cases w*) *simp-all*

lemma *bv-msb-one-length* [*simp*,*intro*]: *bv-msb w =* **1** *==> 0 < length w*
by (*cases w*) *simp-all*

lemma *length-bv-not* [*simp*]: *length (bv-not w) = length w*
by (*induct w*) *simp-all*

definition
bv-to-nat :: bit list => nat **where**
*bv-to-nat = foldl (%bn b. 2 * bn + bitval b) 0*

lemma *bv-to-nat-Nil* [*simp*]: *bv-to-nat [] = 0*
by (*simp add*: *bv-to-nat-def*)

lemma *bv-to-nat-helper* [*simp*]: *bv-to-nat (b # bs) = bitval b * 2 ^ length bs + bv-to-nat bs*
proof −
let *?bv-to-nat' = foldl (λbn b. 2 * bn + bitval b)*

```
  have helper: ⋀base. ?bv-to-nat' base bs = base * 2 ^ length bs + ?bv-to-nat' 0
bs
  proof (induct bs)
    case Nil
    show ?case by simp
  next
    case (Cons x xs base)
    show ?case
      apply (simp only: foldl.simps)
      apply (subst Cons [of 2 * base + bitval x])
      apply simp
      apply (subst Cons [of bitval x])
      apply (simp add: add-mult-distrib)
      done
  qed
  show ?thesis by (simp add: bv-to-nat-def) (rule helper)
qed

lemma bv-to-nat0 [simp]: bv-to-nat (0#bs) = bv-to-nat bs
  by simp

lemma bv-to-nat1 [simp]: bv-to-nat (1#bs) = 2 ^ length bs + bv-to-nat bs
  by simp

lemma bv-to-nat-upper-range: bv-to-nat w < 2 ^ length w
proof (induct w, simp-all)
  fix b bs
  assume bv-to-nat bs < 2 ^ length bs
  show bitval b * 2 ^ length bs + bv-to-nat bs < 2 * 2 ^ length bs
  proof (cases b, simp-all)
    have bv-to-nat bs < 2 ^ length bs by fact
    also have ... < 2 * 2 ^ length bs by auto
    finally show bv-to-nat bs < 2 * 2 ^ length bs by simp
  next
    have bv-to-nat bs < 2 ^ length bs by fact
    hence 2 ^ length bs + bv-to-nat bs < 2 ^ length bs + 2 ^ length bs by arith
    also have ... = 2 * (2 ^ length bs) by simp
    finally show bv-to-nat bs < 2 ^ length bs by simp
  qed
qed

lemma bv-extend-longer [simp]:
  assumes wn: n ≤ length w
  shows       bv-extend n b w = w
  by (simp add: bv-extend-def wn)

lemma bv-extend-shorter [simp]:
  assumes wn: length w < n
  shows       bv-extend n b w = bv-extend n b (b#w)
```

proof −
from *wn*
have *s*: *n* − *Suc* (*length w*) + *1* = *n* − *length w*
by *arith*
have *bv-extend n b w* = *replicate* (*n* − *length w*) *b* @ *w*
by (*simp add*: *bv-extend-def*)
also have ... = *replicate* (*n* − *Suc* (*length w*) + *1*) *b* @ *w*
by (*subst s*) *rule*
also have ... = (*replicate* (*n* − *Suc* (*length w*)) *b* @ *replicate 1 b*) @ *w*
by (*subst replicate-add*) *rule*
also have ... = *replicate* (*n* − *Suc* (*length w*)) *b* @ *b* # *w*
by *simp*
also have ... = *bv-extend n b* (*b*#*w*)
by (*simp add*: *bv-extend-def*)
finally show *bv-extend n b w* = *bv-extend n b* (*b*#*w*) **.**
qed

consts
rem-initial :: *bit* => *bit list* => *bit list*
primrec
rem-initial b [] = []
rem-initial b (*x*#*xs*) = (*if b* = *x then rem-initial b xs else x*#*xs*)

lemma *rem-initial-length*: *length* (*rem-initial b w*) ≤ *length w*
by (*rule bit-list-induct* [*of* - *w*],*simp-all* (*no-asm*),*safe*,*simp-all*)

lemma *rem-initial-equal*:
assumes *p*: *length* (*rem-initial b w*) = *length w*
shows *rem-initial b w* = *w*
proof −
have *length* (*rem-initial b w*) = *length w* −−> *rem-initial b w* = *w*
proof (*induct w*, *simp-all*, *clarify*)
fix *xs*
assume *length* (*rem-initial b xs*) = *length xs* −−> *rem-initial b xs* = *xs*
assume *f*: *length* (*rem-initial b xs*) = *Suc* (*length xs*)
with *rem-initial-length* [*of b xs*]
show *rem-initial b xs* = *b*#*xs*
by *auto*
qed
from *this* **and** *p* **show** *?thesis* **..**
qed

lemma *bv-extend-rem-initial*: *bv-extend* (*length w*) *b* (*rem-initial b w*) = *w*
proof (*induct w*, *simp-all*, *safe*)
fix *xs*
assume *ind*: *bv-extend* (*length xs*) *b* (*rem-initial b xs*) = *xs*
from *rem-initial-length* [*of b xs*]
have [*simp*]: *Suc* (*length xs*) − *length* (*rem-initial b xs*) =
1 + (*length xs* − *length* (*rem-initial b xs*))

```
    by arith
  have bv-extend (Suc (length xs)) b (rem-initial b xs) =
      replicate (Suc (length xs) − length (rem-initial b xs)) b @ (rem-initial b xs)
    by (simp add: bv-extend-def)
  also have ... =
      replicate (1 + (length xs − length (rem-initial b xs))) b @ rem-initial b xs
    by simp
  also have ... =
     (replicate 1 b @ replicate (length xs − length (rem-initial b xs)) b) @ rem-initial
b xs
    by (subst replicate-add) (rule refl)
  also have ... = b # bv-extend (length xs) b (rem-initial b xs)
    by (auto simp add: bv-extend-def [symmetric])
  also have ... = b # xs
    by (simp add: ind)
  finally show bv-extend (Suc (length xs)) b (rem-initial b xs) = b # xs .
qed

lemma rem-initial-append1:
  assumes rem-initial b xs ~= []
  shows   rem-initial b (xs @ ys) = rem-initial b xs @ ys
  using assms by (induct xs) auto

lemma rem-initial-append2:
  assumes rem-initial b xs = []
  shows   rem-initial b (xs @ ys) = rem-initial b ys
  using assms by (induct xs) auto

definition
  norm-unsigned :: bit list => bit list where
  norm-unsigned = rem-initial 0

lemma norm-unsigned-Nil [simp]: norm-unsigned [] = []
  by (simp add: norm-unsigned-def)

lemma norm-unsigned-Cons0 [simp]: norm-unsigned (0#bs) = norm-unsigned bs
  by (simp add: norm-unsigned-def)

lemma norm-unsigned-Cons1 [simp]: norm-unsigned (1#bs) = 1#bs
  by (simp add: norm-unsigned-def)

lemma norm-unsigned-idem [simp]: norm-unsigned (norm-unsigned w) = norm-unsigned
w
  by (rule bit-list-induct [of - w],simp-all)

consts
  nat-to-bv-helper :: nat => bit list => bit list
recdef nat-to-bv-helper measure (λn. n)
  nat-to-bv-helper n = (%bs. (if n = 0 then bs
```

else nat-to-bv-helper (*n div 2*) ((*if n mod 2 = 0 then* **0** *else* **1**)#*bs*)))

definition
nat-to-bv :: *nat* => *bit list* **where**
nat-to-bv n = *nat-to-bv-helper n* []

lemma *nat-to-bv0* [*simp*]: *nat-to-bv 0* = []
by (*simp add*: *nat-to-bv-def*)

lemmas [*simp del*] = *nat-to-bv-helper.simps*

lemma *n-div-2-cases*:
assumes *zero*: (*n*::*nat*) = *0* ==> *R*
and *div* : [| *n div 2* < *n* ; *0* < *n* |] ==> *R*
shows *R*
proof (*cases n* = *0*)
assume *n* = *0*
thus *R* **by** (*rule zero*)
next
assume *n* ~= *0*
hence *0* < *n* **by** *simp*
hence *n div 2* < *n* **by** *arith*
from *this* **and** ‹*0* < *n*› **show** *R* **by** (*rule div*)
qed

lemma *int-wf-ge-induct*:
assumes *ind* : !!*i*::*int*. (!!*j*. [| *k* ≤ *j* ; *j* < *i* |] ==> *P j*) ==> *P i*
shows *P i*
proof (*rule wf-induct-rule* [*OF wf-int-ge-less-than*])
fix *x*
assume *ih*: (⋀*y*::*int*. (*y*, *x*) ∈ *int-ge-less-than k* ⟹ *P y*)
thus *P x*
by (*rule ind*) (*simp add*: *int-ge-less-than-def*)
qed

lemma *unfold-nat-to-bv-helper*:
nat-to-bv-helper b l = *nat-to-bv-helper b* [] @ *l*
proof −
have ∀ *l*. *nat-to-bv-helper b l* = *nat-to-bv-helper b* [] @ *l*
proof (*induct b rule*: *less-induct*)
fix *n*
assume *ind*: !!*j*. *j* < *n* ⟹ ∀ *l*. *nat-to-bv-helper j l* = *nat-to-bv-helper j* [] @ *l*
show ∀ *l*. *nat-to-bv-helper n l* = *nat-to-bv-helper n* [] @ *l*
proof
fix *l*
show *nat-to-bv-helper n l* = *nat-to-bv-helper n* [] @ *l*
proof (*cases n* < *0*)
assume *n* < *0*

```
      thus ?thesis
        by (simp add: nat-to-bv-helper.simps)
    next
      assume ~n < 0
      show ?thesis
      proof (rule n-div-2-cases [of n])
        assume [simp]: n = 0
        show ?thesis
          apply (simp only: nat-to-bv-helper.simps [of n])
          apply simp
          done
      next
        assume n2n: n div 2 < n
        assume [simp]: 0 < n
        hence n20: 0 ≤ n div 2
          by arith
        from ind [of n div 2] and n2n n20
        have ind': ∀ l. nat-to-bv-helper (n div 2) l = nat-to-bv-helper (n div 2) []
@ l
          by blast
        show ?thesis
          apply (simp only: nat-to-bv-helper.simps [of n])
          apply (cases n=0)
          apply simp
          apply (simp only: if-False)
          apply simp
          apply (subst spec [OF ind',of 0#l])
          apply (subst spec [OF ind',of 1#l])
          apply (subst spec [OF ind',of [1]])
          apply (subst spec [OF ind',of [0]])
          apply simp
          done
      qed
    qed
  qed
  thus ?thesis ..
qed

lemma nat-to-bv-non0 [simp]: n≠0 ==> nat-to-bv n = nat-to-bv (n div 2) @ [if
n mod 2 = 0 then 0 else 1]
proof −
  assume [simp]: n≠0
  show ?thesis
    apply (subst nat-to-bv-def [of n])
    apply (simp only: nat-to-bv-helper.simps [of n])
    apply (subst unfold-nat-to-bv-helper)
    using prems
    apply (simp)
```

```
    apply (subst nat-to-bv-def [of n div 2])
    apply auto
    done
qed

lemma bv-to-nat-dist-append:
  bv-to-nat (l1 @ l2) = bv-to-nat l1 * 2 ^ length l2 + bv-to-nat l2
proof −
  have ∀ l2. bv-to-nat (l1 @ l2) = bv-to-nat l1 * 2 ^ length l2 + bv-to-nat l2
  proof (induct l1, simp-all)
    fix x xs
   assume ind: ∀ l2. bv-to-nat (xs @ l2) = bv-to-nat xs * 2 ^ length l2 + bv-to-nat
l2
    show ∀ l2::bit list. bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^
length l2 = (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
    proof
      fix l2
      show bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2 =
(bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
      proof −
        have (2::nat) ^ (length xs + length l2) = 2 ^ length xs * 2 ^ length l2
          by (induct length xs,simp-all)
       hence bitval x * 2 ^ (length xs + length l2) + bv-to-nat xs * 2 ^ length l2 =
          bitval x * 2 ^ length xs * 2 ^ length l2 + bv-to-nat xs * 2 ^ length l2
          by simp
        also have ... = (bitval x * 2 ^ length xs + bv-to-nat xs) * 2 ^ length l2
          by (simp add: ring-distribs)
        finally show ?thesis by simp
      qed
    qed
  qed
  thus ?thesis ..
qed

lemma bv-nat-bv [simp]: bv-to-nat (nat-to-bv n) = n
proof (induct n rule: less-induct)
  fix n
  assume ind: !!j. j < n ⟹ bv-to-nat (nat-to-bv j) = j
  show bv-to-nat (nat-to-bv n) = n
  proof (rule n-div-2-cases [of n])
    assume n = 0 then show ?thesis by simp
  next
    assume nn: n div 2 < n
    assume n0: 0 < n
    from ind and nn
    have ind': bv-to-nat (nat-to-bv (n div 2)) = n div 2 by blast
    from n0 have n0': n ≠ 0 by simp
    show ?thesis
      apply (subst nat-to-bv-def)
```

```
    apply (simp only: nat-to-bv-helper.simps [of n])
    apply (simp only: n0' if-False)
    apply (subst unfold-nat-to-bv-helper)
    apply (subst bv-to-nat-dist-append)
    apply (fold nat-to-bv-def)
    apply (simp add: ind' split del: split-if)
    apply (cases n mod 2 = 0)
    proof (simp-all)
      assume n mod 2 = 0
      with mod-div-equality [of n 2]
      show n div 2 * 2 = n by simp
    next
      assume n mod 2 = Suc 0
      with mod-div-equality [of n 2]
      show Suc (n div 2 * 2) = n by arith
    qed
  qed
qed

lemma bv-to-nat-type [simp]: bv-to-nat (norm-unsigned w) = bv-to-nat w
  by (rule bit-list-induct) simp-all

lemma length-norm-unsigned-le [simp]: length (norm-unsigned w) ≤ length w
  by (rule bit-list-induct) simp-all

lemma bv-to-nat-rew-msb: bv-msb w = 1 ==> bv-to-nat w = 2 ^ (length w - 1)
+ bv-to-nat (tl w)
  by (rule bit-list-cases [of w]) simp-all

lemma norm-unsigned-result: norm-unsigned xs = [] ∨ bv-msb (norm-unsigned xs)
= 1
proof (rule length-induct [of - xs])
  fix xs :: bit list
  assume ind: ∀ys. length ys < length xs --> norm-unsigned ys = [] ∨ bv-msb
(norm-unsigned ys) = 1
  show norm-unsigned xs = [] ∨ bv-msb (norm-unsigned xs) = 1
  proof (rule bit-list-cases [of xs],simp-all)
    fix bs
    assume [simp]: xs = 0#bs
    from ind
   have length bs < length xs --> norm-unsigned bs = [] ∨ bv-msb (norm-unsigned
bs) = 1 ..
    thus norm-unsigned bs = [] ∨ bv-msb (norm-unsigned bs) = 1 by simp
  qed
qed

lemma norm-empty-bv-to-nat-zero:
  assumes nw: norm-unsigned w = []
  shows       bv-to-nat w = 0
```

proof −
have *bv-to-nat w = bv-to-nat (norm-unsigned w)* **by** *simp*
also have *... = bv-to-nat []* **by** *(subst nw) (rule refl)*
also have *... = 0* **by** *simp*
finally show *?thesis* **.**
qed

lemma *bv-to-nat-lower-limit*:
assumes *w0*: *0 < bv-to-nat w*
shows *2 ^ (length (norm-unsigned w) − 1) ≤ bv-to-nat w*
proof −
from *w0* **and** *norm-unsigned-result [of w]*
have *msbw*: *bv-msb (norm-unsigned w) = 1*
by *(auto simp add*: *norm-empty-bv-to-nat-zero)*
have *2 ^ (length (norm-unsigned w) − 1) ≤ bv-to-nat (norm-unsigned w)*
by *(subst bv-to-nat-rew-msb [OF msbw],simp)*
thus *?thesis* **by** *simp*
qed

lemmas *[simp del] = nat-to-bv-non0*

lemma *norm-unsigned-length [intro!]*: *length (norm-unsigned w) ≤ length w*
by *(subst norm-unsigned-def ,rule rem-initial-length)*

lemma *norm-unsigned-equal*:
length (norm-unsigned w) = length w ==> norm-unsigned w = w
by *(simp add*: *norm-unsigned-def ,rule rem-initial-equal)*

lemma *bv-extend-norm-unsigned*: *bv-extend (length w)* **0** *(norm-unsigned w) = w*
by *(simp add*: *norm-unsigned-def ,rule bv-extend-rem-initial)*

lemma *norm-unsigned-append1 [simp]*:
norm-unsigned xs ≠ [] ==> norm-unsigned (xs @ ys) = norm-unsigned xs @ ys
by *(simp add*: *norm-unsigned-def ,rule rem-initial-append1)*

lemma *norm-unsigned-append2 [simp]*:
norm-unsigned xs = [] ==> norm-unsigned (xs @ ys) = norm-unsigned ys
by *(simp add*: *norm-unsigned-def ,rule rem-initial-append2)*

lemma *bv-to-nat-zero-imp-empty*:
bv-to-nat w = 0 ⟹ norm-unsigned w = []
by *(atomize (full), induct w rule*: *bit-list-induct) simp-all*

lemma *bv-to-nat-nzero-imp-nempty*:
bv-to-nat w ≠ 0 ⟹ norm-unsigned w ≠ []
by *(induct w rule*: *bit-list-induct) simp-all*

lemma *nat-helper1*:
assumes *ass*: *nat-to-bv (bv-to-nat w) = norm-unsigned w*

```
  shows       nat-to-bv (2 * bv-to-nat w + bitval x) = norm-unsigned (w @ [x])
proof (cases x)
  assume [simp]: x = 1
  show ?thesis
    apply (simp add: nat-to-bv-non0)
    apply safe
  proof −
    fix q
    assume Suc (2 * bv-to-nat w) = 2 * q
    hence orig: (2 * bv-to-nat w + 1) mod 2 = 2 * q mod 2 (is ?lhs = ?rhs)
      by simp
    have ?lhs = (1 + 2 * bv-to-nat w) mod 2
      by (simp add: add-commute)
    also have ... = 1
      by (subst mod-add-eq) simp
    finally have eq1: ?lhs = 1 .
    have ?rhs  = 0 by simp
    with orig and eq1
   show nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [0] = norm-unsigned (w @ [1])
      by simp
  next
    have nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [1] =
        nat-to-bv ((1 + 2 * bv-to-nat w) div 2) @ [1]
      by (simp add: add-commute)
    also have ... = nat-to-bv (bv-to-nat w) @ [1]
      by (subst div-add1-eq) simp
    also have ... = norm-unsigned w @ [1]
      by (subst ass) (rule refl)
    also have ... = norm-unsigned (w @ [1])
      by (cases norm-unsigned w) simp-all
    finally show nat-to-bv (Suc (2 * bv-to-nat w) div 2) @ [1] = norm-unsigned
(w @ [1]) .
  qed
next
  assume [simp]: x = 0
  show ?thesis
  proof (cases bv-to-nat w = 0)
    assume bv-to-nat w = 0
    thus ?thesis
      by (simp add: bv-to-nat-zero-imp-empty)
  next
    assume bv-to-nat w ≠ 0
    thus ?thesis
      apply simp
      apply (subst nat-to-bv-non0)
      apply simp
      apply auto
      apply (subst ass)
      apply (cases norm-unsigned w)
```

```
    apply (simp-all add: norm-empty-bv-to-nat-zero)
    done
  qed
qed

lemma nat-helper2: nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs
proof −
  have ∀ xs. nat-to-bv (2 ^ length (rev xs) + bv-to-nat (rev xs)) = 1 # (rev xs)
(is ∀ xs. ?P xs)
  proof
    fix xs
    show ?P xs
    proof (rule length-induct [of - xs])
      fix xs :: bit list
      assume ind: ∀ ys. length ys < length xs −−> ?P ys
      show ?P xs
      proof (cases xs)
        assume xs = []
        then show ?thesis by (simp add: nat-to-bv-non0)
      next
        fix y ys
        assume [simp]: xs = y # ys
        show ?thesis
          apply simp
          apply (subst bv-to-nat-dist-append)
          apply simp
        proof −
          have nat-to-bv (2 * 2 ^ length ys + (bv-to-nat (rev ys) * 2 + bitval y)) =
            nat-to-bv (2 * (2 ^ length ys + bv-to-nat (rev ys)) + bitval y)
            by (simp add: add-ac mult-ac)
          also have ... = nat-to-bv (2 * (bv-to-nat (1#rev ys)) + bitval y)
            by simp
          also have ... = norm-unsigned (1#rev ys) @ [y]
          proof −
            from ind
            have nat-to-bv (2 ^ length (rev ys) + bv-to-nat (rev ys)) = 1 # rev ys
              by auto
           hence [simp]: nat-to-bv (2 ^ length ys + bv-to-nat (rev ys)) = 1 # rev ys
              by simp
            show ?thesis
              apply (subst nat-helper1)
              apply simp-all
              done
          qed
          also have ... = (1#rev ys) @ [y]
            by simp
          also have ... = 1 # rev ys @ [y]
            by simp
          finally show nat-to-bv (2 * 2 ^ length ys + (bv-to-nat (rev ys) * 2 +
```

```
bitval y)) =
            1 # rev ys @ [y] .
       qed
     qed
   qed
  qed
  hence nat-to-bv (2 ^ length (rev (rev xs)) + bv-to-nat (rev (rev xs))) =
     1 # rev (rev xs) ..
  thus ?thesis by simp
qed

lemma nat-bv-nat [simp]: nat-to-bv (bv-to-nat w) = norm-unsigned w
proof (rule bit-list-induct [of - w],simp-all)
  fix xs
  assume nat-to-bv (bv-to-nat xs) = norm-unsigned xs
  have bv-to-nat xs = bv-to-nat (norm-unsigned xs) by simp
  have bv-to-nat xs < 2 ^ length xs
    by (rule bv-to-nat-upper-range)
  show nat-to-bv (2 ^ length xs + bv-to-nat xs) = 1 # xs
    by (rule nat-helper2)
qed

lemma bv-to-nat-qinj:
  assumes one: bv-to-nat xs = bv-to-nat ys
  and     len: length xs = length ys
  shows        xs = ys
proof -
  from one
  have nat-to-bv (bv-to-nat xs) = nat-to-bv (bv-to-nat ys)
    by simp
  hence xsys: norm-unsigned xs = norm-unsigned ys
    by simp
  have xs = bv-extend (length xs) 0 (norm-unsigned xs)
    by (simp add: bv-extend-norm-unsigned)
  also have ... = bv-extend (length ys) 0 (norm-unsigned ys)
    by (simp add: xsys len)
  also have ... = ys
    by (simp add: bv-extend-norm-unsigned)
  finally show ?thesis .
qed

lemma norm-unsigned-nat-to-bv [simp]:
  norm-unsigned (nat-to-bv n) = nat-to-bv n
proof -
 have norm-unsigned (nat-to-bv n) = nat-to-bv (bv-to-nat (norm-unsigned (nat-to-bv
n)))
    by (subst nat-bv-nat) simp
  also have ... = nat-to-bv n by simp
  finally show ?thesis .
```

qed

lemma *length-nat-to-bv-upper-limit*:
assumes *nk*: $n \leq 2 \hat{} k - 1$
shows $length\ (nat\text{-}to\text{-}bv\ n) \leq k$
proof (*cases* $n = 0$)
case *True*
thus *?thesis*
by (*simp add*: *nat-to-bv-def nat-to-bv-helper.simps*)
next
case *False*
hence *n0*: $0 < n$ **by** *simp*
show *?thesis*
proof (*rule ccontr*)
assume ~ $length\ (nat\text{-}to\text{-}bv\ n) \leq k$
hence $k < length\ (nat\text{-}to\text{-}bv\ n)$ **by** *simp*
hence $k \leq length\ (nat\text{-}to\text{-}bv\ n) - 1$ **by** *arith*
hence $(2::nat) \hat{} k \leq 2 \hat{} (length\ (nat\text{-}to\text{-}bv\ n) - 1)$ **by** *simp*
also have ... $= 2 \hat{} (length\ (norm\text{-}unsigned\ (nat\text{-}to\text{-}bv\ n)) - 1)$ **by** *simp*
also have ... $\leq bv\text{-}to\text{-}nat\ (nat\text{-}to\text{-}bv\ n)$
by (*rule bv-to-nat-lower-limit*) (*simp add*: *n0*)
also have ... $= n$ **by** *simp*
finally have $2 \hat{} k \leq n$ **.**
with *n0* **have** $2 \hat{} k - 1 < n$ **by** *arith*
with *nk* **show** *False* **by** *simp*
qed
qed

lemma *length-nat-to-bv-lower-limit*:
assumes *nk*: $2 \hat{} k \leq n$
shows $k < length\ (nat\text{-}to\text{-}bv\ n)$
proof (*rule ccontr*)
assume ~ $k < length\ (nat\text{-}to\text{-}bv\ n)$
hence *lnk*: $length\ (nat\text{-}to\text{-}bv\ n) \leq k$ **by** *simp*
have $n = bv\text{-}to\text{-}nat\ (nat\text{-}to\text{-}bv\ n)$ **by** *simp*
also have ... $< 2 \hat{} length\ (nat\text{-}to\text{-}bv\ n)$
by (*rule bv-to-nat-upper-range*)
also from *lnk* **have** ... $\leq 2 \hat{} k$ **by** *simp*
finally have $n < 2 \hat{} k$ **.**
with *nk* **show** *False* **by** *simp*
qed

67.4 Unsigned Arithmetic Operations

definition
bv-add :: [*bit list*, *bit list*] => *bit list* **where**
bv-add w1 w2 = *nat-to-bv* (*bv-to-nat w1* + *bv-to-nat w2*)

lemma *bv-add-type1* [*simp*]: *bv-add* (*norm-unsigned w1*) *w2* = *bv-add w1 w2*

by (*simp add*: *bv-add-def*)

lemma *bv-add-type2* [*simp*]: *bv-add w1* (*norm-unsigned w2*) = *bv-add w1 w2*
by (*simp add*: *bv-add-def*)

lemma *bv-add-returntype* [*simp*]: *norm-unsigned* (*bv-add w1 w2*) = *bv-add w1 w2*
by (*simp add*: *bv-add-def*)

lemma *bv-add-length*: *length* (*bv-add w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
proof (*unfold bv-add-def*,*rule length-nat-to-bv-upper-limit*)
from *bv-to-nat-upper-range* [*of w1*] **and** *bv-to-nat-upper-range* [*of w2*]
have *bv-to-nat w1* + *bv-to-nat w2* ≤ (2 ^ *length w1* − 1) + (2 ^ *length w2* − 1)
by *arith*
also have ... ≤
max (2 ^ *length w1* − 1) (2 ^ *length w2* − 1) + *max* (2 ^ *length w1* − 1) (2 ^ *length w2* − 1)
by (*rule add-mono*,*safe intro*!: *le-maxI1 le-maxI2*)
also have ... = 2 ∗ *max* (2 ^ *length w1* − 1) (2 ^ *length w2* − 1) **by** *simp*
also have ... ≤ 2 ^ *Suc* (*max* (*length w1*) (*length w2*)) − 2
proof (*cases length w1* ≤ *length w2*)
assume *w1w2*: *length w1* ≤ *length w2*
hence (2::*nat*) ^ *length w1* ≤ 2 ^ *length w2* **by** *simp*
hence (2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1 **by** *arith*
with *w1w2* **show** *?thesis*
by (*simp add*: *diff-mult-distrib2 split*: *split-max*)
next
assume [*simp*]: ~ (*length w1* ≤ *length w2*)
have ~ ((2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1)
proof
assume (2::*nat*) ^ *length w1* − 1 ≤ 2 ^ *length w2* − 1
hence ((2::*nat*) ^ *length w1* − 1) + 1 ≤ (2 ^ *length w2* − 1) + 1
by (*rule add-right-mono*)
hence (2::*nat*) ^ *length w1* ≤ 2 ^ *length w2* **by** *simp*
hence *length w1* ≤ *length w2* **by** *simp*
thus *False* **by** *simp*
qed
thus *?thesis*
by (*simp add*: *diff-mult-distrib2 split*: *split-max*)
qed
finally show *bv-to-nat w1* + *bv-to-nat w2* ≤ 2 ^ *Suc* (*max* (*length w1*) (*length w2*)) − 1
by *arith*
qed

definition
bv-mult :: [*bit list*, *bit list*] => *bit list* **where**
bv-mult w1 w2 = *nat-to-bv* (*bv-to-nat w1* ∗ *bv-to-nat w2*)

lemma *bv-mult-type1* [*simp*]: *bv-mult* (*norm-unsigned w1*) *w2* = *bv-mult w1 w2*

by (*simp add*: *bv-mult-def*)

lemma *bv-mult-type2* [*simp*]: *bv-mult w1* (*norm-unsigned w2*) = *bv-mult w1 w2*
by (*simp add*: *bv-mult-def*)

lemma *bv-mult-returntype* [*simp*]: *norm-unsigned* (*bv-mult w1 w2*) = *bv-mult w1 w2*
by (*simp add*: *bv-mult-def*)

lemma *bv-mult-length*: *length* (*bv-mult w1 w2*) ≤ *length w1* + *length w2*
proof (*unfold bv-mult-def*,*rule length-nat-to-bv-upper-limit*)
from *bv-to-nat-upper-range* [*of w1*] **and** *bv-to-nat-upper-range* [*of w2*]
have *h*: *bv-to-nat w1* ≤ *2* ^ *length w1* − *1* ∧ *bv-to-nat w2* ≤ *2* ^ *length w2* − *1*
by *arith*
have *bv-to-nat w1* ∗ *bv-to-nat w2* ≤ (*2* ^ *length w1* − *1*) ∗ (*2* ^ *length w2* − *1*)
apply (*cut-tac h*)
apply (*rule mult-mono*)
apply *auto*
done
also have ... < *2* ^ *length w1* ∗ *2* ^ *length w2*
by (*rule mult-strict-mono*,*auto*)
also have ... = *2* ^ (*length w1* + *length w2*)
by (*simp add*: *power-add*)
finally show *bv-to-nat w1* ∗ *bv-to-nat w2* ≤ *2* ^ (*length w1* + *length w2*) − *1*
by *arith*
qed

67.5 Signed Vectors

consts
norm-signed :: *bit list* => *bit list*
primrec
norm-signed-Nil: *norm-signed* [] = []
norm-signed-Cons: *norm-signed* (*b*#*bs*) =
(*case b of*
0 => *if norm-unsigned bs* = [] *then* [] *else b*#*norm-unsigned bs*
| **1** => *b*#*rem-initial b bs*)

lemma *norm-signed0* [*simp*]: *norm-signed* [**0**] = []
by *simp*

lemma *norm-signed1* [*simp*]: *norm-signed* [**1**] = [**1**]
by *simp*

lemma *norm-signed01* [*simp*]: *norm-signed* (**0**#**1**#*xs*) = **0**#**1**#*xs*
by *simp*

lemma *norm-signed00* [*simp*]: *norm-signed* (**0**#**0**#*xs*) = *norm-signed* (**0**#*xs*)
by *simp*

lemma *norm-signed10* [*simp*]: *norm-signed* (**1**#**0**#*xs*) = **1**#**0**#*xs*
by *simp*

lemma *norm-signed11* [*simp*]: *norm-signed* (**1**#**1**#*xs*) = *norm-signed* (**1**#*xs*)
by *simp*

lemmas [*simp del*] = *norm-signed-Cons*

definition
int-to-bv :: *int* => *bit list* **where**
int-to-bv n = (*if* 0 ≤ *n*
then norm-signed (**0**#*nat-to-bv* (*nat n*))
else norm-signed (*bv-not* (**0**#*nat-to-bv* (*nat* (−*n*− 1)))))

lemma *int-to-bv-ge0* [*simp*]: 0 ≤ *n* ==> *int-to-bv n* = *norm-signed* (**0** # *nat-to-bv* (*nat n*))
by (*simp add*: *int-to-bv-def*)

lemma *int-to-bv-lt0* [*simp*]:
n < 0 ==> *int-to-bv n* = *norm-signed* (*bv-not* (**0**#*nat-to-bv* (*nat* (−*n*− 1))))
by (*simp add*: *int-to-bv-def*)

lemma *norm-signed-idem* [*simp*]: *norm-signed* (*norm-signed w*) = *norm-signed w*
proof (*rule bit-list-induct* [*of* - *w*], *simp-all*)
fix *xs*
assume *eq*: *norm-signed* (*norm-signed xs*) = *norm-signed xs*
show *norm-signed* (*norm-signed* (**0**#*xs*)) = *norm-signed* (**0**#*xs*)
proof (*rule bit-list-cases* [*of xs*],*simp-all*)
fix *ys*
assume *xs* = **0**#*ys*
from *this* [*symmetric*] **and** *eq*
show *norm-signed* (*norm-signed* (**0**#*ys*)) = *norm-signed* (**0**#*ys*)
by *simp*
qed
next
fix *xs*
assume *eq*: *norm-signed* (*norm-signed xs*) = *norm-signed xs*
show *norm-signed* (*norm-signed* (**1**#*xs*)) = *norm-signed* (**1**#*xs*)
proof (*rule bit-list-cases* [*of xs*],*simp-all*)
fix *ys*
assume *xs* = **1**#*ys*
from *this* [*symmetric*] **and** *eq*
show *norm-signed* (*norm-signed* (**1**#*ys*)) = *norm-signed* (**1**#*ys*)
by *simp*
qed
qed

definition

bv-to-int :: *bit list* => *int* **where**
bv-to-int w =
(*case bv-msb w of* **0** => *int* (*bv-to-nat w*)
| **1** => − *int* (*bv-to-nat* (*bv-not w*) + *1*))

lemma *bv-to-int-Nil* [*simp*]: *bv-to-int* [] = *0*
by (*simp add*: *bv-to-int-def*)

lemma *bv-to-int-Cons0* [*simp*]: *bv-to-int* (**0**#*bs*) = *int* (*bv-to-nat bs*)
by (*simp add*: *bv-to-int-def*)

lemma *bv-to-int-Cons1* [*simp*]: *bv-to-int* (**1**#*bs*) = − *int* (*bv-to-nat* (*bv-not bs*) + *1*)
by (*simp add*: *bv-to-int-def*)

lemma *bv-to-int-type* [*simp*]: *bv-to-int* (*norm-signed w*) = *bv-to-int w*
proof (*rule bit-list-induct* [*of* - *w*], *simp-all*)
fix *xs*
assume *ind*: *bv-to-int* (*norm-signed xs*) = *bv-to-int xs*
show *bv-to-int* (*norm-signed* (**0**#*xs*)) = *int* (*bv-to-nat xs*)
proof (*rule bit-list-cases* [*of xs*], *simp-all*)
fix *ys*
assume [*simp*]: *xs* = **0**#*ys*
from *ind*
show *bv-to-int* (*norm-signed* (**0**#*ys*)) = *int* (*bv-to-nat ys*)
by *simp*
qed
next
fix *xs*
assume *ind*: *bv-to-int* (*norm-signed xs*) = *bv-to-int xs*
show *bv-to-int* (*norm-signed* (**1**#*xs*)) = *−1* − *int* (*bv-to-nat* (*bv-not xs*))
proof (*rule bit-list-cases* [*of xs*], *simp-all*)
fix *ys*
assume [*simp*]: *xs* = **1**#*ys*
from *ind*
show *bv-to-int* (*norm-signed* (**1**#*ys*)) = *−1* − *int* (*bv-to-nat* (*bv-not ys*))
by *simp*
qed
qed

lemma *bv-to-int-upper-range*: *bv-to-int w* < *2* ^ (*length w* − *1*)
proof (*rule bit-list-cases* [*of w*],*simp-all*)
fix *bs*
from *bv-to-nat-upper-range*
show *int* (*bv-to-nat bs*) < *2* ^ *length bs*
by (*simp add*: *int-nat-two-exp*)
next
fix *bs*
have *−1* − *int* (*bv-to-nat* (*bv-not bs*)) ≤ *0* **by** *simp*

```
  also have ... < 2 ^ length bs by (induct bs) simp-all
  finally show -1 - int (bv-to-nat (bv-not bs)) < 2 ^ length bs .
qed

lemma bv-to-int-lower-range: - (2 ^ (length w - 1)) ≤ bv-to-int w
proof (rule bit-list-cases [of w],simp-all)
  fix bs :: bit list
  have - (2 ^ length bs) ≤ (0::int) by (induct bs) simp-all
  also have ... ≤ int (bv-to-nat bs) by simp
  finally show - (2 ^ length bs) ≤ int (bv-to-nat bs) .
next
  fix bs
  from bv-to-nat-upper-range [of bv-not bs]
  show - (2 ^ length bs) ≤ -1 - int (bv-to-nat (bv-not bs))
    by (simp add: int-nat-two-exp)
qed

lemma int-bv-int [simp]: int-to-bv (bv-to-int w) = norm-signed w
proof (rule bit-list-cases [of w],simp)
  fix xs
  assume [simp]: w = 0#xs
  show ?thesis
    apply simp
    apply (subst norm-signed-Cons [of 0 xs])
    apply simp
    using norm-unsigned-result [of xs]
    apply safe
    apply (rule bit-list-cases [of norm-unsigned xs])
    apply simp-all
    done
next
  fix xs
  assume [simp]: w = 1#xs
  show ?thesis
    apply (simp del: int-to-bv-lt0)
    apply (rule bit-list-induct [of - xs])
    apply simp
    apply (subst int-to-bv-lt0)
    apply (subgoal-tac - int (bv-to-nat (bv-not (0 # bs))) + -1 < 0 + 0)
    apply simp
    apply (rule add-le-less-mono)
    apply simp
    apply simp
    apply (simp del: bv-to-nat1 bv-to-nat-helper)
    apply simp
    done
qed

lemma bv-int-bv [simp]: bv-to-int (int-to-bv i) = i
```

by (*cases 0 ≤ i*) *simp-all*

lemma *bv-msb-norm* [*simp*]: *bv-msb* (*norm-signed w*) = *bv-msb w*
by (*rule bit-list-cases* [*of w*]) (*simp-all add*: *norm-signed-Cons*)

lemma *norm-signed-length*: *length* (*norm-signed w*) ≤ *length w*
apply (*cases w*, *simp-all*)
apply (*subst norm-signed-Cons*)
apply (*case-tac a*, *simp-all*)
apply (*rule rem-initial-length*)
done

lemma *norm-signed-equal*: *length* (*norm-signed w*) = *length w* ==> *norm-signed w* = *w*
proof (*rule bit-list-cases* [*of w*], *simp-all*)
fix *xs*
assume *length* (*norm-signed* (**0**#*xs*)) = *Suc* (*length xs*)
thus *norm-signed* (**0**#*xs*) = **0**#*xs*
apply (*simp add*: *norm-signed-Cons*)
apply *safe*
apply *simp-all*
apply (*rule norm-unsigned-equal*)
apply *assumption*
done
next
fix *xs*
assume *length* (*norm-signed* (**1**#*xs*)) = *Suc* (*length xs*)
thus *norm-signed* (**1**#*xs*) = **1**#*xs*
apply (*simp add*: *norm-signed-Cons*)
apply (*rule rem-initial-equal*)
apply *assumption*
done
qed

lemma *bv-extend-norm-signed*: *bv-msb w* = *b* ==> *bv-extend* (*length w*) *b* (*norm-signed w*) = *w*
proof (*rule bit-list-cases* [*of w*],*simp-all*)
fix *xs*
show *bv-extend* (*Suc* (*length xs*)) **0** (*norm-signed* (**0**#*xs*)) = **0**#*xs*
proof (*simp add*: *norm-signed-list-def*,*auto*)
assume *norm-unsigned xs* = []
hence *xx*: *rem-initial* **0** *xs* = []
by (*simp add*: *norm-unsigned-def*)
have *bv-extend* (*Suc* (*length xs*)) **0** (**0**#*rem-initial* **0** *xs*) = **0**#*xs*
apply (*simp add*: *bv-extend-def replicate-app-Cons-same*)
apply (*fold bv-extend-def*)
apply (*rule bv-extend-rem-initial*)
done
thus *bv-extend* (*Suc* (*length xs*)) **0** [**0**] = **0**#*xs*

```
    by (simp add: xx)
  next
    show bv-extend (Suc (length xs)) 0 (0#norm-unsigned xs) = 0#xs
      apply (simp add: norm-unsigned-def)
      apply (simp add: bv-extend-def replicate-app-Cons-same)
      apply (fold bv-extend-def)
      apply (rule bv-extend-rem-initial)
      done
  qed
next
  fix xs
  show bv-extend (Suc (length xs)) 1 (norm-signed (1#xs)) = 1#xs
    apply (simp add: norm-signed-Cons)
    apply (simp add: bv-extend-def replicate-app-Cons-same)
    apply (fold bv-extend-def)
    apply (rule bv-extend-rem-initial)
    done
qed

lemma bv-to-int-qinj:
  assumes one: bv-to-int xs = bv-to-int ys
  and     len: length xs = length ys
  shows        xs = ys
proof -
  from one
  have int-to-bv (bv-to-int xs) = int-to-bv (bv-to-int ys) by simp
  hence xsys: norm-signed xs = norm-signed ys by simp
  hence xsys': bv-msb xs = bv-msb ys
  proof -
    have bv-msb xs = bv-msb (norm-signed xs) by simp
    also have ... = bv-msb (norm-signed ys) by (simp add: xsys)
    also have ... = bv-msb ys by simp
    finally show ?thesis .
  qed
  have xs = bv-extend (length xs) (bv-msb xs) (norm-signed xs)
    by (simp add: bv-extend-norm-signed)
  also have ... = bv-extend (length ys) (bv-msb ys) (norm-signed ys)
    by (simp add: xsys xsys' len)
  also have ... = ys
    by (simp add: bv-extend-norm-signed)
  finally show ?thesis .
qed

lemma int-to-bv-returntype [simp]: norm-signed (int-to-bv w) = int-to-bv w
  by (simp add: int-to-bv-def)

lemma bv-to-int-msb0: 0 ≤ bv-to-int w1 ==> bv-msb w1 = 0
  by (rule bit-list-cases,simp-all)
```

lemma *bv-to-int-msb1*: *bv-to-int w1 < 0 ==> bv-msb w1 =* **1**
by (*rule bit-list-cases*,*simp-all*)

lemma *bv-to-int-lower-limit-gt0*:
assumes *w0*: *0 < bv-to-int w*
shows *2 ^ (length (norm-signed w) − 2) ≤ bv-to-int w*
proof −
from *w0*
have *0 ≤ bv-to-int w* **by** *simp*
hence [*simp*]: *bv-msb w =* **0** **by** (*rule bv-to-int-msb0*)
have *2 ^ (length (norm-signed w) − 2) ≤ bv-to-int (norm-signed w)*
proof (*rule bit-list-cases* [*of w*])
assume *w* = []
with *w0* **show** *?thesis* **by** *simp*
next
fix *w′*
assume *weq*: *w =* **0** *# w′*
thus *?thesis*
proof (*simp add*: *norm-signed-Cons*,*safe*)
assume *norm-unsigned w′* = []
with *weq* **and** *w0* **show** *False*
by (*simp add*: *norm-empty-bv-to-nat-zero*)
next
assume *w′0*: *norm-unsigned w′* ≠ []
have *0 < bv-to-nat w′*
proof (*rule ccontr*)
assume ~ (*0 < bv-to-nat w′*)
hence *bv-to-nat w′ = 0*
by *arith*
hence *norm-unsigned w′* = []
by (*simp add*: *bv-to-nat-zero-imp-empty*)
with *w′0*
show *False* **by** *simp*
qed
with *bv-to-nat-lower-limit* [*of w′*]
show *2 ^ (length (norm-unsigned w′) − Suc 0) ≤ int (bv-to-nat w′)*
by (*simp add*: *int-nat-two-exp*)
qed
next
fix *w′*
assume *w =* **1** *# w′*
from *w0* **have** *bv-msb w =* **0** **by** *simp*
with *prems* **show** *?thesis* **by** *simp*
qed
also have *... = bv-to-int w* **by** *simp*
finally show *?thesis* **.**
qed

lemma *norm-signed-result*: *norm-signed w* = [] ∨ *norm-signed w* = [**1**] ∨ *bv-msb*

```
(norm-signed w) ≠ bv-msb (tl (norm-signed w))
  apply (rule bit-list-cases [of w],simp-all)
  apply (case-tac bs,simp-all)
  apply (case-tac a,simp-all)
  apply (simp add: norm-signed-Cons)
  apply safe
  apply simp
proof −
  fix l
  assume msb: 0 = bv-msb (norm-unsigned l)
  assume norm-unsigned l ≠ []
  with norm-unsigned-result [of l]
  have bv-msb (norm-unsigned l) = 1 by simp
  with msb show False by simp
next
  fix xs
  assume p: 1 = bv-msb (tl (norm-signed (1 # xs)))
  have 1 ≠ bv-msb (tl (norm-signed (1 # xs)))
    by (rule bit-list-induct [of - xs],simp-all)
  with p show False by simp
qed

lemma bv-to-int-upper-limit-lem1:
  assumes w0: bv-to-int w < −1
  shows       bv-to-int w < − (2 ^ (length (norm-signed w) − 2))
proof −
  from w0
  have bv-to-int w < 0 by simp
  hence msbw [simp]: bv-msb w = 1
    by (rule bv-to-int-msb1)
  have bv-to-int w = bv-to-int (norm-signed w) by simp
  also from norm-signed-result [of w]
  have ... < − (2 ^ (length (norm-signed w) − 2))
  proof safe
    assume norm-signed w = []
    hence bv-to-int (norm-signed w) = 0 by simp
    with w0 show ?thesis by simp
  next
    assume norm-signed w = [1]
    hence bv-to-int (norm-signed w) = −1 by simp
    with w0 show ?thesis by simp
  next
    assume bv-msb (norm-signed w) ≠ bv-msb (tl (norm-signed w))
    hence msb-tl: 1 ≠ bv-msb (tl (norm-signed w)) by simp
    show bv-to-int (norm-signed w) < − (2 ^ (length (norm-signed w) − 2))
    proof (rule bit-list-cases [of norm-signed w])
      assume norm-signed w = []
      hence bv-to-int (norm-signed w) = 0 by simp
      with w0 show ?thesis by simp
```

```
    next
      fix w'
      assume nw: norm-signed w = 0 # w'
      from msbw have bv-msb (norm-signed w) = 1 by simp
      with nw show ?thesis by simp
    next
      fix w'
      assume weq: norm-signed w = 1 # w'
      show ?thesis
      proof (rule bit-list-cases [of w'])
        assume w'eq: w' = []
        from w0 have bv-to-int (norm-signed w) < -1 by simp
        with w'eq and weq show ?thesis by simp
      next
        fix w''
        assume w'eq: w' = 0 # w''
        show ?thesis
          apply (simp add: weq w'eq)
          apply (subgoal-tac - int (bv-to-nat (bv-not w'')) + -1 < 0 + 0)
          apply (simp add: int-nat-two-exp)
          apply (rule add-le-less-mono)
          apply simp-all
          done
      next
        fix w''
        assume w'eq: w' = 1 # w''
        with weq and msb-tl show ?thesis by simp
      qed
    qed
  qed
  finally show ?thesis .
qed

lemma length-int-to-bv-upper-limit-gt0:
  assumes w0: 0 < i
  and     wk: i ≤ 2 ^ (k - 1) - 1
  shows       length (int-to-bv i) ≤ k
proof (rule ccontr)
  from w0 wk
  have k1: 1 < k
    by (cases k - 1,simp-all)
  assume ~ length (int-to-bv i) ≤ k
  hence k < length (int-to-bv i) by simp
  hence k ≤ length (int-to-bv i) - 1 by arith
  hence a: k - 1 ≤ length (int-to-bv i) - 2 by arith
  hence (2::int) ^ (k - 1) ≤ 2 ^ (length (int-to-bv i) - 2) by simp
  also have ... ≤ i
  proof -
    have 2 ^ (length (norm-signed (int-to-bv i)) - 2) ≤ bv-to-int (int-to-bv i)
```

```
    proof (rule bv-to-int-lower-limit-gt0)
      from w0 show 0 < bv-to-int (int-to-bv i) by simp
    qed
    thus ?thesis by simp
  qed
  finally have 2 ^ (k - 1) ≤ i .
  with wk show False by simp
qed

lemma pos-length-pos:
  assumes i0: 0 < bv-to-int w
  shows       0 < length w
proof -
  from norm-signed-result [of w]
  have 0 < length (norm-signed w)
  proof (auto)
    assume ii: norm-signed w = []
    have bv-to-int (norm-signed w) = 0 by (subst ii) simp
    hence bv-to-int w = 0 by simp
    with i0 show False by simp
  next
    assume ii: norm-signed w = []
    assume jj: bv-msb w ≠ 0
    have 0 = bv-msb (norm-signed w)
      by (subst ii) simp
    also have ... ≠ 0
      by (simp add: jj)
    finally show False by simp
  qed
  also have ... ≤ length w
    by (rule norm-signed-length)
  finally show ?thesis .
qed

lemma neg-length-pos:
  assumes i0: bv-to-int w < -1
  shows       0 < length w
proof -
  from norm-signed-result [of w]
  have 0 < length (norm-signed w)
  proof (auto)
    assume ii: norm-signed w = []
    have bv-to-int (norm-signed w) = 0
      by (subst ii) simp
    hence bv-to-int w = 0 by simp
    with i0 show False by simp
  next
    assume ii: norm-signed w = []
    assume jj: bv-msb w ≠ 0
```

```
    have 0 = bv-msb (norm-signed w) by (subst ii) simp
    also have ... ≠ 0 by (simp add: jj)
    finally show False by simp
  qed
  also have ... ≤ length w
    by (rule norm-signed-length)
  finally show ?thesis .
qed

lemma length-int-to-bv-lower-limit-gt0:
  assumes wk: 2 ^ (k - 1) ≤ i
  shows       k < length (int-to-bv i)
proof (rule ccontr)
  have 0 < (2::int) ^ (k - 1)
    by (rule zero-less-power) simp
  also have ... ≤ i by (rule wk)
  finally have ii0: 0 < i .
  have lii0: 0 < length (int-to-bv i)
    apply (rule pos-length-pos)
    apply (simp,rule ii0)
    done
  assume ~ k < length (int-to-bv i)
  hence length (int-to-bv i) ≤ k by simp
  with lii0
  have a: length (int-to-bv i) - 1 ≤ k - 1
    by arith
  have i < 2 ^ (length (int-to-bv i) - 1)
  proof -
    have i = bv-to-int (int-to-bv i)
      by simp
    also have ... < 2 ^ (length (int-to-bv i) - 1)
      by (rule bv-to-int-upper-range)
    finally show ?thesis .
  qed
  also have (2::int) ^ (length (int-to-bv i) - 1) ≤ 2 ^ (k - 1) using a
    by simp
  finally have i < 2 ^ (k - 1) .
  with wk show False by simp
qed

lemma length-int-to-bv-upper-limit-lem1:
  assumes w1: i < -1
  and     wk: - (2 ^ (k - 1)) ≤ i
  shows       length (int-to-bv i) ≤ k
proof (rule ccontr)
  from w1 wk
  have k1: 1 < k by (cases k - 1) simp-all
  assume ~ length (int-to-bv i) ≤ k
  hence k < length (int-to-bv i) by simp
```

```
  hence k ≤ length (int-to-bv i) − 1 by arith
  hence a: k − 1 ≤ length (int-to-bv i) − 2 by arith
  have i < − (2 ^ (length (int-to-bv i) − 2))
  proof −
    have i = bv-to-int (int-to-bv i)
      by simp
    also have ... < − (2 ^ (length (norm-signed (int-to-bv i)) − 2))
      by (rule bv-to-int-upper-limit-lem1,simp,rule w1)
    finally show ?thesis by simp
  qed
  also have ... ≤ −(2 ^ (k − 1))
  proof −
    have (2::int) ^ (k − 1) ≤ 2 ^ (length (int-to-bv i) − 2) using a by simp
    thus ?thesis by simp
  qed
  finally have i < −(2 ^ (k − 1)) .
  with wk show False by simp
qed

lemma length-int-to-bv-lower-limit-lem1:
  assumes wk: i < −(2 ^ (k − 1))
  shows       k < length (int-to-bv i)
proof (rule ccontr)
  from wk have i ≤ −(2 ^ (k − 1)) − 1 by simp
  also have ... < −1
  proof −
    have 0 < (2::int) ^ (k − 1)
      by (rule zero-less-power) simp
    hence −((2::int) ^ (k − 1)) < 0 by simp
    thus ?thesis by simp
  qed
  finally have i1: i < −1 .
  have lii0: 0 < length (int-to-bv i)
    apply (rule neg-length-pos)
    apply (simp, rule i1)
    done
  assume ~ k < length (int-to-bv i)
  hence length (int-to-bv i) ≤ k
    by simp
  with lii0 have a: length (int-to-bv i) − 1 ≤ k − 1 by arith
  hence (2::int) ^ (length (int-to-bv i) − 1) ≤ 2 ^ (k − 1) by simp
  hence −((2::int) ^ (k − 1)) ≤ − (2 ^ (length (int-to-bv i) − 1)) by simp
  also have ... ≤ i
  proof −
    have − (2 ^ (length (int-to-bv i) − 1)) ≤ bv-to-int (int-to-bv i)
      by (rule bv-to-int-lower-range)
    also have ... = i
      by simp
    finally show ?thesis .
```

qed
finally have $-(2\ \hat{}\ (k - 1)) \leq i$ **.**
with *wk* **show** *False* **by** *simp*
qed

67.6 Signed Arithmetic Operations

67.6.1 Conversion from unsigned to signed

definition
utos :: *bit list* => *bit list* **where**
utos w = *norm-signed* (**0** # *w*)

lemma *utos-type* [*simp*]: *utos* (*norm-unsigned w*) = *utos w*
by (*simp add*: *utos-def norm-signed-Cons*)

lemma *utos-returntype* [*simp*]: *norm-signed* (*utos w*) = *utos w*
by (*simp add*: *utos-def*)

lemma *utos-length*: *length* (*utos w*) ≤ *Suc* (*length w*)
by (*simp add*: *utos-def norm-signed-Cons*)

lemma *bv-to-int-utos*: *bv-to-int* (*utos w*) = *int* (*bv-to-nat w*)
proof (*simp add*: *utos-def norm-signed-Cons*, *safe*)
assume *norm-unsigned w* = []
hence *bv-to-nat* (*norm-unsigned w*) = *0* **by** *simp*
thus *bv-to-nat w* = *0* **by** *simp*
qed

67.6.2 Unary minus

definition
bv-uminus :: *bit list* => *bit list* **where**
bv-uminus w = *int-to-bv* (− *bv-to-int w*)

lemma *bv-uminus-type* [*simp*]: *bv-uminus* (*norm-signed w*) = *bv-uminus w*
by (*simp add*: *bv-uminus-def*)

lemma *bv-uminus-returntype* [*simp*]: *norm-signed* (*bv-uminus w*) = *bv-uminus w*
by (*simp add*: *bv-uminus-def*)

lemma *bv-uminus-length*: *length* (*bv-uminus w*) ≤ *Suc* (*length w*)
proof −
have *1* < −*bv-to-int w* ∨ −*bv-to-int w* = *1* ∨ −*bv-to-int w* = *0* ∨ −*bv-to-int w* = −*1* ∨ −*bv-to-int w* < −*1*
by *arith*
thus *?thesis*
proof *safe*
assume *p*: *1* < − *bv-to-int w*
have *lw*: *0* < *length w*

apply (*rule neg-length-pos*)
using *p*
apply *simp*
done
show *?thesis*
proof (*simp add*: *bv-uminus-def*,*rule length-int-to-bv-upper-limit-gt0*,*simp-all*)
from *prems* **show** *bv-to-int w < 0* **by** *simp*
next
have *−(2^(length w − 1)) ≤ bv-to-int w*
by (*rule bv-to-int-lower-range*)
hence *− bv-to-int w ≤ 2^(length w − 1)* **by** *simp*
also from *lw* **have** *... < 2 ^ length w* **by** *simp*
finally show *− bv-to-int w < 2 ^ length w* **by** *simp*
qed
next
assume *p*: *− bv-to-int w = 1*
hence *lw*: *0 < length w* **by** (*cases w*) *simp-all*
from *p*
show *?thesis*
apply (*simp add*: *bv-uminus-def*)
using *lw*
apply (*simp* (*no-asm*) *add*: *nat-to-bv-non0*)
done
next
assume *− bv-to-int w = 0*
thus *?thesis* **by** (*simp add*: *bv-uminus-def*)
next
assume *p*: *− bv-to-int w = −1*
thus *?thesis* **by** (*simp add*: *bv-uminus-def*)
next
assume *p*: *− bv-to-int w < −1*
show *?thesis*
apply (*simp add*: *bv-uminus-def*)
apply (*rule length-int-to-bv-upper-limit-lem1*)
apply (*rule p*)
apply *simp*
proof *−*
have *bv-to-int w < 2 ^ (length w − 1)*
by (*rule bv-to-int-upper-range*)
also have *... ≤ 2 ^ length w* **by** *simp*
finally show *bv-to-int w ≤ 2 ^ length w* **by** *simp*
qed
qed
qed

lemma *bv-uminus-length-utos*: *length* (*bv-uminus* (*utos w*)) ≤ *Suc* (*length w*)
proof *−*
have *−bv-to-int* (*utos w*) = *0* ∨ *−bv-to-int* (*utos w*) = *−1* ∨ *−bv-to-int* (*utos w*) < *−1*

```
    by (simp add: bv-to-int-utos, arith)
  thus ?thesis
  proof safe
    assume -bv-to-int (utos w) = 0
    thus ?thesis by (simp add: bv-uminus-def)
  next
    assume -bv-to-int (utos w) = -1
    thus ?thesis by (simp add: bv-uminus-def)
  next
    assume p: -bv-to-int (utos w) < -1
    show ?thesis
      apply (simp add: bv-uminus-def)
      apply (rule length-int-to-bv-upper-limit-lem1)
      apply (rule p)
      apply (simp add: bv-to-int-utos)
      using bv-to-nat-upper-range [of w]
      apply (simp add: int-nat-two-exp)
      done
  qed
qed

definition
  bv-sadd :: [bit list, bit list ] => bit list where
  bv-sadd w1 w2 = int-to-bv (bv-to-int w1 + bv-to-int w2)

lemma bv-sadd-type1 [simp]: bv-sadd (norm-signed w1) w2 = bv-sadd w1 w2
  by (simp add: bv-sadd-def)

lemma bv-sadd-type2 [simp]: bv-sadd w1 (norm-signed w2) = bv-sadd w1 w2
  by (simp add: bv-sadd-def)

lemma bv-sadd-returntype [simp]: norm-signed (bv-sadd w1 w2) = bv-sadd w1 w2
  by (simp add: bv-sadd-def)

lemma adder-helper:
  assumes lw: 0 < max (length w1) (length w2)
  shows  ((2::int) ^ (length w1 - 1)) + (2 ^ (length w2 - 1)) ≤ 2 ^ max (length
w1) (length w2)
proof -
  have ((2::int) ^ (length w1 - 1)) + (2 ^ (length w2 - 1)) ≤
      2 ^ (max (length w1) (length w2) - 1) + 2 ^ (max (length w1) (length w2)
- 1)
    apply (cases length w1 ≤ length w2)
    apply (auto simp add: max-def)
    done
  also have ... = 2 ^ max (length w1) (length w2)
  proof -
    from lw
    show ?thesis
```

```
    apply simp
    apply (subst power-Suc [symmetric])
    apply (simp del: power-int.simps)
    done
 qed
 finally show ?thesis .
qed

lemma bv-sadd-length: length (bv-sadd w1 w2) ≤ Suc (max (length w1) (length
w2))
proof −
 let ?Q = bv-to-int w1 + bv-to-int w2

 have helper: ?Q ≠ 0 ==> 0 < max (length w1) (length w2)
 proof −
  assume p: ?Q ≠ 0
  show 0 < max (length w1) (length w2)
  proof (simp add: less-max-iff-disj,rule)
   assume [simp]: w1 = []
   show w2 ≠ []
   proof (rule ccontr,simp)
    assume [simp]: w2 = []
    from p show False by simp
   qed
  qed
 qed

 have 0 < ?Q ∨ ?Q = 0 ∨ ?Q = −1 ∨ ?Q < −1 by arith
 thus ?thesis
 proof safe
  assume ?Q = 0
  thus ?thesis
   by (simp add: bv-sadd-def)
 next
  assume ?Q = −1
  thus ?thesis
   by (simp add: bv-sadd-def)
 next
  assume p: 0 < ?Q
  show ?thesis
   apply (simp add: bv-sadd-def)
   apply (rule length-int-to-bv-upper-limit-gt0)
   apply (rule p)
  proof simp
   from bv-to-int-upper-range [of w2]
   have bv-to-int w2 ≤ 2 ^ (length w2 − 1)
    by simp
   with bv-to-int-upper-range [of w1]
   have bv-to-int w1 + bv-to-int w2 < (2 ^ (length w1 − 1)) + (2 ^ (length w2
```

```
− 1))
      by (rule zadd-zless-mono)
    also have ... ≤ 2 ^ max (length w1) (length w2)
      apply (rule adder-helper)
      apply (rule helper)
      using p
      apply simp
      done
    finally show ?Q < 2 ^ max (length w1) (length w2) .
  qed
 next
  assume p: ?Q < −1
  show ?thesis
    apply (simp add: bv-sadd-def)
    apply (rule length-int-to-bv-upper-limit-lem1,simp-all)
    apply (rule p)
  proof −
    have (2 ^ (length w1 − 1)) + 2 ^ (length w2 − 1) ≤ (2::int) ^ max (length
w1) (length w2)
      apply (rule adder-helper)
      apply (rule helper)
      using p
      apply simp
      done
    hence −((2::int) ^ max (length w1) (length w2)) ≤ − (2 ^ (length w1 − 1))
+ −(2 ^ (length w2 − 1))
      by simp
    also have − (2 ^ (length w1 − 1)) + −(2 ^ (length w2 − 1)) ≤ ?Q
      apply (rule add-mono)
      apply (rule bv-to-int-lower-range [of w1])
      apply (rule bv-to-int-lower-range [of w2])
      done
    finally show − (2^max (length w1) (length w2)) ≤ ?Q .
  qed
 qed
qed

definition
  bv-sub :: [bit list, bit list] => bit list where
  bv-sub w1 w2 = bv-sadd w1 (bv-uminus w2)

lemma bv-sub-type1 [simp]: bv-sub (norm-signed w1) w2 = bv-sub w1 w2
  by (simp add: bv-sub-def)

lemma bv-sub-type2 [simp]: bv-sub w1 (norm-signed w2) = bv-sub w1 w2
  by (simp add: bv-sub-def)

lemma bv-sub-returntype [simp]: norm-signed (bv-sub w1 w2) = bv-sub w1 w2
  by (simp add: bv-sub-def)
```

lemma *bv-sub-length*: *length* (*bv-sub w1 w2*) ≤ *Suc* (*max* (*length w1*) (*length w2*))
proof (*cases bv-to-int w2 = 0*)
  assume *p*: *bv-to-int w2 = 0*
  show *?thesis*
  proof (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def p*)
    have *length* (*norm-signed w1*) ≤ *length w1*
      by (*rule norm-signed-length*)
    also have ... ≤ *max* (*length w1*) (*length w2*)
      by (*rule le-maxI1*)
    also have ... ≤ *Suc* (*max* (*length w1*) (*length w2*))
      by *arith*
    finally show *length* (*norm-signed w1*) ≤ *Suc* (*max* (*length w1*) (*length w2*)) **.**
  qed
next
  assume *bv-to-int w2* ≠ *0*
  hence *0* < *length w2* **by** (*cases w2*,*simp-all*)
  hence *lmw*: *0* < *max* (*length w1*) (*length w2*) **by** *arith*

  let *?Q* = *bv-to-int w1* − *bv-to-int w2*

  have *0* < *?Q* ∨ *?Q* = *0* ∨ *?Q* = −*1* ∨ *?Q* < −*1* **by** *arith*
  thus *?thesis*
  proof *safe*
    assume *?Q* = *0*
    thus *?thesis*
      by (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
  next
    assume *?Q* = −*1*
    thus *?thesis*
      by (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
  next
    assume *p*: *0* < *?Q*
    show *?thesis*
      apply (*simp add*: *bv-sub-def bv-sadd-def bv-uminus-def*)
      apply (*rule length-int-to-bv-upper-limit-gt0*)
      apply (*rule p*)
    proof *simp*
      from *bv-to-int-lower-range* [*of w2*]
      have *v2*: − *bv-to-int w2* ≤ *2* ^ (*length w2* − *1*) **by** *simp*
      have *bv-to-int w1* + − *bv-to-int w2* < (*2* ^ (*length w1* − *1*)) + (*2* ^ (*length w2* − *1*))
        apply (*rule zadd-zless-mono*)
        apply (*rule bv-to-int-upper-range* [*of w1*])
        apply (*rule v2*)
        done
      also have ... ≤ *2* ^ *max* (*length w1*) (*length w2*)
        apply (*rule adder-helper*)
        apply (*rule lmw*)

```
      done
    finally show ?Q < 2 ^ max (length w1) (length w2) by simp
  qed
 next
  assume p: ?Q < -1
  show ?thesis
    apply (simp add: bv-sub-def bv-sadd-def bv-uminus-def)
    apply (rule length-int-to-bv-upper-limit-lem1)
    apply (rule p)
  proof simp
    have (2 ^ (length w1 - 1)) + 2 ^ (length w2 - 1) ≤ (2::int) ^ max (length
w1) (length w2)
      apply (rule adder-helper)
      apply (rule lmw)
      done
    hence -((2::int) ^ max (length w1) (length w2)) ≤ - (2 ^ (length w1 - 1))
+ -(2 ^ (length w2 - 1))
      by simp
    also have - (2 ^ (length w1 - 1)) + -(2 ^ (length w2 - 1)) ≤ bv-to-int w1
+ -bv-to-int w2
      apply (rule add-mono)
      apply (rule bv-to-int-lower-range [of w1])
      using bv-to-int-upper-range [of w2]
      apply simp
      done
    finally show - (2^max (length w1) (length w2)) ≤ ?Q by simp
  qed
 qed
qed

definition
  bv-smult :: [bit list, bit list] => bit list where
  bv-smult w1 w2 = int-to-bv (bv-to-int w1 * bv-to-int w2)

lemma bv-smult-type1 [simp]: bv-smult (norm-signed w1) w2 = bv-smult w1 w2
  by (simp add: bv-smult-def)

lemma bv-smult-type2 [simp]: bv-smult w1 (norm-signed w2) = bv-smult w1 w2
  by (simp add: bv-smult-def)

lemma bv-smult-returntype [simp]: norm-signed (bv-smult w1 w2) = bv-smult w1
w2
  by (simp add: bv-smult-def)

lemma bv-smult-length: length (bv-smult w1 w2) ≤ length w1 + length w2
proof -
  let ?Q = bv-to-int w1 * bv-to-int w2

  have lmw: ?Q ≠ 0 ==> 0 < length w1 ∧ 0 < length w2 by auto
```

have *0 < ?Q ∨ ?Q = 0 ∨ ?Q = −1 ∨ ?Q < −1* **by** *arith*
thus *?thesis*
proof (*safe dest*!: *iffD1* [*OF mult-eq-0-iff*])
assume *bv-to-int w1 = 0*
thus *?thesis* **by** (*simp add*: *bv-smult-def*)
next
assume *bv-to-int w2 = 0*
thus *?thesis* **by** (*simp add*: *bv-smult-def*)
next
assume *p*: *?Q = −1*
show *?thesis*
apply (*simp add*: *bv-smult-def p*)
apply (*cut-tac lmw*)
apply *arith*
using *p*
apply *simp*
done
next
assume *p*: *0 < ?Q*
thus *?thesis*
proof (*simp add*: *zero-less-mult-iff*,*safe*)
assume *bi1*: *0 < bv-to-int w1*
assume *bi2*: *0 < bv-to-int w2*
show *?thesis*
apply (*simp add*: *bv-smult-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)
apply (*rule p*)
proof *simp*
have *?Q < 2 ^ (length w1 − 1) * 2 ^ (length w2 − 1)*
apply (*rule mult-strict-mono*)
apply (*rule bv-to-int-upper-range*)
apply (*rule bv-to-int-upper-range*)
apply (*rule zero-less-power*)
apply *simp*
using *bi2*
apply *simp*
done
also have *... ≤ 2 ^ (length w1 + length w2 − Suc 0)*
apply *simp*
apply (*subst zpower-zadd-distrib* [*symmetric*])
apply *simp*
done
finally show *?Q < 2 ^ (length w1 + length w2 − Suc 0)* **.**
qed
next
assume *bi1*: *bv-to-int w1 < 0*
assume *bi2*: *bv-to-int w2 < 0*
show *?thesis*

```
      apply (simp add: bv-smult-def)
      apply (rule length-int-to-bv-upper-limit-gt0)
      apply (rule p)
     proof simp
      have −bv-to-int w1 ∗ −bv-to-int w2 ≤ 2 ^ (length w1 − 1) ∗ 2 ^(length w2
− 1)
       apply (rule mult-mono)
       using bv-to-int-lower-range [of w1]
       apply simp
       using bv-to-int-lower-range [of w2]
       apply simp
       apply (rule zero-le-power,simp)
       using bi2
       apply simp
       done
      hence ?Q ≤ 2 ^ (length w1 − 1) ∗ 2 ^(length w2 − 1)
       by simp
      also have ... < 2 ^ (length w1 + length w2 − Suc 0)
       apply simp
       apply (subst zpower-zadd-distrib [symmetric])
       apply simp
       apply (cut-tac lmw)
       apply arith
       apply (cut-tac p)
       apply arith
       done
      finally show ?Q < 2 ^ (length w1 + length w2 − Suc 0) .
     qed
    qed
   next
    assume p: ?Q < −1
    show ?thesis
     apply (subst bv-smult-def)
     apply (rule length-int-to-bv-upper-limit-lem1)
     apply (rule p)
    proof simp
     have (2::int) ^ (length w1 − 1) ∗ 2 ^(length w2 − 1) ≤ 2 ^ (length w1 +
length w2 − Suc 0)
      apply simp
      apply (subst zpower-zadd-distrib [symmetric])
      apply simp
      done
     hence −((2::int) ^ (length w1 + length w2 − Suc 0)) ≤ −(2^(length w1 −
1) ∗ 2 ^ (length w2 − 1))
      by simp
     also have ... ≤ ?Q
     proof −
      from p
      have q: bv-to-int w1 ∗ bv-to-int w2 < 0
```

```
        by simp
      thus ?thesis
      proof (simp add: mult-less-0-iff,safe)
        assume bi1: 0 < bv-to-int w1
        assume bi2: bv-to-int w2 < 0
         have -bv-to-int w2 * bv-to-int w1 ≤ ((2::int) ^ (length w2 - 1)) * (2 ^
(length w1 - 1))
          apply (rule mult-mono)
          using bv-to-int-lower-range [of w2]
          apply simp
          using bv-to-int-upper-range [of w1]
          apply simp
          apply (rule zero-le-power,simp)
          using bi1
          apply simp
          done
        hence -?Q ≤ ((2::int) ^ (length w1 - 1)) * (2 ^ (length w2 - 1))
          by (simp add: zmult-ac)
         thus -(((2::int) ^ (length w1 - Suc 0)) * (2 ^ (length w2 - Suc 0))) ≤
?Q
          by simp
      next
        assume bi1: bv-to-int w1 < 0
        assume bi2: 0 < bv-to-int w2
         have -bv-to-int w1 * bv-to-int w2 ≤ ((2::int) ^ (length w1 - 1)) * (2 ^
(length w2 - 1))
          apply (rule mult-mono)
          using bv-to-int-lower-range [of w1]
          apply simp
          using bv-to-int-upper-range [of w2]
          apply simp
          apply (rule zero-le-power,simp)
          using bi2
          apply simp
          done
        hence -?Q ≤ ((2::int) ^ (length w1 - 1)) * (2 ^ (length w2 - 1))
          by (simp add: zmult-ac)
         thus -(((2::int) ^ (length w1 - Suc 0)) * (2 ^ (length w2 - Suc 0))) ≤
?Q
          by simp
      qed
    qed
    finally show -(2 ^ (length w1 + length w2 - Suc 0)) ≤ ?Q .
  qed
 qed
qed

lemma bv-msb-one: bv-msb w = 1 ==> bv-to-nat w ≠ 0
by (cases w) simp-all
```

lemma *bv-smult-length-utos*: *length* (*bv-smult* (*utos w1*) *w2*) ≤ *length w1* + *length w2*
proof −
let *?Q* = *bv-to-int* (*utos w1*) ∗ *bv-to-int w2*

have *lmw*: *?Q* ≠ *0* ==> *0* < *length* (*utos w1*) ∧ *0* < *length w2* **by** *auto*

have *0* < *?Q* ∨ *?Q* = *0* ∨ *?Q* = −*1* ∨ *?Q* < −*1* **by** *arith*
thus *?thesis*
proof (*safe dest*!: *iffD1* [*OF mult-eq-0-iff*])
assume *bv-to-int* (*utos w1*) = *0*
thus *?thesis* **by** (*simp add*: *bv-smult-def*)
next
assume *bv-to-int w2* = *0*
thus *?thesis* **by** (*simp add*: *bv-smult-def*)
next
assume *p*: *0* < *?Q*
thus *?thesis*
proof (*simp add*: *zero-less-mult-iff*,*safe*)
assume *biw2*: *0* < *bv-to-int w2*
show *?thesis*
apply (*simp add*: *bv-smult-def*)
apply (*rule length-int-to-bv-upper-limit-gt0*)
apply (*rule p*)
proof *simp*
have *?Q* < *2* ^ *length w1* ∗ *2* ^ (*length w2* − *1*)
apply (*rule mult-strict-mono*)
apply (*simp add*: *bv-to-int-utos int-nat-two-exp*)
apply (*rule bv-to-nat-upper-range*)
apply (*rule bv-to-int-upper-range*)
apply (*rule zero-less-power*,*simp*)
using *biw2*
apply *simp*
done
also have ... ≤ *2* ^ (*length w1* + *length w2* − *Suc 0*)
apply *simp*
apply (*subst zpower-zadd-distrib* [*symmetric*])
apply *simp*
apply (*cut-tac lmw*)
apply *arith*
using *p*
apply *auto*
done
finally show *?Q* < *2* ^ (*length w1* + *length w2* − *Suc 0*) .
qed
next
assume *bv-to-int* (*utos w1*) < *0*
thus *?thesis* **by** (*simp add*: *bv-to-int-utos*)

```
  qed
 next
  assume p: ?Q = −1
  thus ?thesis
   apply (simp add: bv-smult-def)
   apply (cut-tac lmw)
   apply arith
   apply simp
   done
 next
  assume p: ?Q < −1
  show ?thesis
   apply (subst bv-smult-def)
   apply (rule length-int-to-bv-upper-limit-lem1)
   apply (rule p)
  proof simp
    have (2::int) ^ length w1 * 2 ^(length w2 − 1) ≤ 2 ^ (length w1 + length
w2 − Suc 0)
     apply simp
     apply (subst zpower-zadd-distrib [symmetric])
     apply simp
     apply (cut-tac lmw)
     apply arith
     apply (cut-tac p)
     apply arith
     done
   hence −((2::int) ^ (length w1 + length w2 − Suc 0)) ≤ −(2^ length w1 * 2
^ (length w2 − 1))
     by simp
   also have ... ≤ ?Q
   proof −
     from p
     have q: bv-to-int (utos w1) * bv-to-int w2 < 0
      by simp
     thus ?thesis
     proof (simp add: mult-less-0-iff,safe)
      assume bi1: 0 < bv-to-int (utos w1)
      assume bi2: bv-to-int w2 < 0
      have −bv-to-int w2 * bv-to-int (utos w1) ≤ ((2::int) ^(length w2 − 1)) *
(2 ^ length w1)
       apply (rule mult-mono)
       using bv-to-int-lower-range [of w2]
       apply simp
       apply (simp add: bv-to-int-utos)
       using bv-to-nat-upper-range [of w1]
       apply (simp add: int-nat-two-exp)
       apply (rule zero-le-power,simp)
       using bi1
       apply simp
```

```
        done
      hence −?Q ≤ ((2::int) ^ length w1) * (2 ^ (length w2 − 1))
        by (simp add: zmult-ac)
      thus −(((2::int) ^ length w1) * (2 ^ (length w2 − Suc 0))) ≤ ?Q
        by simp
    next
      assume bi1: bv-to-int (utos w1) < 0
      thus −(((2::int) ^ length w1) * (2 ^ (length w2 − Suc 0))) ≤ ?Q
        by (simp add: bv-to-int-utos)
    qed
  qed
  finally show −(2 ^ (length w1 + length w2 − Suc 0)) ≤ ?Q .
 qed
qed

lemma bv-smult-sym: bv-smult w1 w2 = bv-smult w2 w1
 by (simp add: bv-smult-def zmult-ac)
```

67.7 Structural operations

```
definition
 bv-select :: [bit list,nat] => bit where
 bv-select w i = w ! (length w − 1 − i)

definition
 bv-chop :: [bit list,nat] => bit list * bit list where
 bv-chop w i = (let len = length w in (take (len − i) w,drop (len − i) w))

definition
 bv-slice :: [bit list,nat*nat] => bit list where
 bv-slice w = (λ(b,e). fst (bv-chop (snd (bv-chop w (b+1))) e))

lemma bv-select-rev:
 assumes notnull: n < length w
 shows          bv-select w n = rev w ! n
proof −
 have ∀ n. n < length w −−> bv-select w n = rev w ! n
 proof (rule length-induct [of - w],auto simp add: bv-select-def)
  fix xs :: bit list
  fix n
  assume ind: ∀ ys::bit list. length ys < length xs −−> (∀ n. n < length ys −−>
ys ! (length ys − Suc n) = rev ys ! n)
  assume notx: n < length xs
  show xs ! (length xs − Suc n) = rev xs ! n
  proof (cases xs)
   assume xs = []
   with notx show ?thesis by simp
  next
```

```
    fix y ys
    assume [simp]: xs = y # ys
    show ?thesis
    proof (auto simp add: nth-append)
      assume noty: n < length ys
      from spec [OF ind,of ys]
      have ∀ n. n < length ys −−> ys ! (length ys − Suc n) = rev ys ! n
        by simp
      hence n < length ys −−> ys ! (length ys − Suc n) = rev ys ! n ..
      from this and noty
      have ys ! (length ys − Suc n) = rev ys ! n ..
      thus (y # ys) ! (length ys − n) = rev ys ! n
        by (simp add: nth-Cons' noty linorder-not-less [symmetric])
    next
      assume ~ n < length ys
      hence x: length ys ≤ n by simp
      from notx have n < Suc (length ys) by simp
      hence n ≤ length ys by simp
      with x have length ys = n by simp
      thus y = [y] ! (n − length ys) by simp
    qed
  qed
  then have n < length w −−> bv-select w n = rev w ! n ..
  from this and notnull show ?thesis ..
qed

lemma bv-chop-append: bv-chop (w1 @ w2) (length w2) = (w1,w2)
  by (simp add: bv-chop-def Let-def)

lemma append-bv-chop-id: fst (bv-chop w l) @ snd (bv-chop w l) = w
  by (simp add: bv-chop-def Let-def)

lemma bv-chop-length-fst [simp]: length (fst (bv-chop w i)) = length w − i
  by (simp add: bv-chop-def Let-def)

lemma bv-chop-length-snd [simp]: length (snd (bv-chop w i)) = min i (length w)
  by (simp add: bv-chop-def Let-def)

lemma bv-slice-length [simp]: [| j ≤ i; i < length w |] ==> length (bv-slice w
(i,j)) = i − j + 1
  by (auto simp add: bv-slice-def)

definition
  length-nat :: nat => nat where
  [code del]: length-nat x = (LEAST n. x < 2 ^ n)

lemma length-nat: length (nat-to-bv n) = length-nat n
  apply (simp add: length-nat-def)
```

```
  apply (rule Least-equality [symmetric])
  prefer 2
  apply (rule length-nat-to-bv-upper-limit)
  apply arith
  apply (rule ccontr)
proof −
  assume ~ n < 2 ^ length (nat-to-bv n)
  hence 2 ^ length (nat-to-bv n) ≤ n by simp
  hence length (nat-to-bv n) < length (nat-to-bv n)
    by (rule length-nat-to-bv-lower-limit)
  thus False by simp
qed

lemma length-nat-0 [simp]: length-nat 0 = 0
  by (simp add: length-nat-def Least-equality)

lemma length-nat-non0:
  assumes n0: n ≠ 0
  shows       length-nat n = Suc (length-nat (n div 2))
  apply (simp add: length-nat [symmetric])
  apply (subst nat-to-bv-non0 [of n])
  apply (simp-all add: n0)
  done

definition
  length-int :: int => nat where
  length-int x =
    (if 0 < x then Suc (length-nat (nat x))
    else if x = 0 then 0
    else Suc (length-nat (nat (−x − 1))))

lemma length-int: length (int-to-bv i) = length-int i
proof (cases 0 < i)
  assume i0: 0 < i
  hence length (int-to-bv i) =
      length (norm-signed (0 # norm-unsigned (nat-to-bv (nat i)))) by simp
  also from norm-unsigned-result [of nat-to-bv (nat i)]
  have ... = Suc (length-nat (nat i))
    apply safe
    apply (simp del: norm-unsigned-nat-to-bv)
    apply (drule norm-empty-bv-to-nat-zero)
    using prems
    apply simp
    apply (cases norm-unsigned (nat-to-bv (nat i)))
    apply (drule norm-empty-bv-to-nat-zero [of nat-to-bv (nat i)])
    using prems
    apply simp
    apply simp
    using prems
```

```
    apply (simp add: length-nat [symmetric])
    done
  finally show ?thesis
    using i0
    by (simp add: length-int-def)
next
  assume ~ 0 < i
  hence i0: i ≤ 0 by simp
  show ?thesis
  proof (cases i = 0)
    assume i = 0
    thus ?thesis by (simp add: length-int-def)
  next
    assume i ≠ 0
    with i0 have i0: i < 0 by simp
    hence length (int-to-bv i) =
        length (norm-signed (1 # bv-not (norm-unsigned (nat-to-bv (nat (− i) −
1)))))
      by (simp add: int-to-bv-def nat-diff-distrib)
    also from norm-unsigned-result [of nat-to-bv (nat (− i) − 1)]
    have ... = Suc (length-nat (nat (− i) − 1))
      apply safe
      apply (simp del: norm-unsigned-nat-to-bv)
      apply (drule norm-empty-bv-to-nat-zero [of nat-to-bv (nat (−i) − Suc 0)])
      using prems
      apply simp
      apply (cases − i − 1 = 0)
      apply simp
      apply (simp add: length-nat [symmetric])
      apply (cases norm-unsigned (nat-to-bv (nat (− i) − 1)))
      apply simp
      apply simp
      done
    finally
    show ?thesis
      using i0 by (simp add: length-int-def nat-diff-distrib del: int-to-bv-lt0)
  qed
qed

lemma length-int-0 [simp]: length-int 0 = 0
  by (simp add: length-int-def)

lemma length-int-gt0: 0 < i ==> length-int i = Suc (length-nat (nat i))
  by (simp add: length-int-def)

lemma length-int-lt0: i < 0 ==> length-int i = Suc (length-nat (nat (− i) − 1))
  by (simp add: length-int-def nat-diff-distrib)

lemma bv-chopI: [| w = w1 @ w2 ; i = length w2 |] ==> bv-chop w i = (w1,w2)
```

by (*simp add*: *bv-chop-def Let-def*)

lemma *bv-sliceI*: [| $j \leq i$; $i < length\ w$; $w = w1\ @\ w2\ @\ w3$; $Suc\ i = length\ w2 + j$; $j = length\ w3$ |] ==> *bv-slice w* (*i*,*j*) = *w2*
apply (*simp add*: *bv-slice-def*)
apply (*subst bv-chopI* [*of w1 @ w2 @ w3 w1 w2 @ w3*])
apply *simp*
apply *simp*
apply *simp*
apply (*subst bv-chopI* [*of w2 @ w3 w2 w3*],*simp-all*)
done

lemma *bv-slice-bv-slice*:
assumes *ki*: $k \leq i$
and *ij*: $i \leq j$
and *jl*: $j \leq l$
and *lw*: $l < length\ w$
shows *bv-slice w* (*j*,*i*) = *bv-slice* (*bv-slice w* (*l*,*k*)) (*j*−*k*,*i*−*k*)
proof −
def *w1* == *fst* (*bv-chop w* (*Suc l*))
and *w2* == *fst* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))
and *w3* == *fst* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)
and *w4* == *fst* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)) *k*)
and *w5* == *snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop* (*snd* (*bv-chop w* (*Suc l*))) (*Suc j*))) *i*)) *k*)
note *w-defs* = *this*

have *w-def*: *w* = *w1* @ *w2* @ *w3* @ *w4* @ *w5*
by (*simp add*: *w-defs append-bv-chop-id*)

from *ki ij jl lw*
show *?thesis*
apply (*subst bv-sliceI* [**where** *?j* = *i* **and** *?i* = *j* **and** *?w* = *w* **and** *?w1.0* = *w1* @ *w2* **and** *?w2.0* = *w3* **and** *?w3.0* = *w4* @ *w5*])
apply *simp-all*
apply (*rule w-def*)
apply (*simp add*: *w-defs min-def*)
apply (*simp add*: *w-defs min-def*)
apply (*subst bv-sliceI* [**where** *?j* = *k* **and** *?i* = *l* **and** *?w* = *w* **and** *?w1.0* = *w1* **and** *?w2.0* = *w2* @ *w3* @ *w4* **and** *?w3.0* = *w5*])
apply *simp-all*
apply (*rule w-def*)
apply (*simp add*: *w-defs min-def*)
apply (*simp add*: *w-defs min-def*)
apply (*subst bv-sliceI* [**where** *?j* = *i*−*k* **and** *?i* = *j*−*k* **and** *?w* = *w2* @ *w3* @ *w4* **and** *?w1.0* = *w2* **and** *?w2.0* = *w3* **and** *?w3.0* = *w4*])
apply *simp-all*
apply (*simp-all add*: *w-defs min-def*)

```
    done
qed

lemma bv-to-nat-extend [simp]: bv-to-nat (bv-extend n 0 w) = bv-to-nat w
  apply (simp add: bv-extend-def)
  apply (subst bv-to-nat-dist-append)
  apply simp
  apply (induct n - length w)
   apply simp-all
  done

lemma bv-msb-extend-same [simp]: bv-msb w = b ==> bv-msb (bv-extend n b w)
= b
  apply (simp add: bv-extend-def)
  apply (induct n - length w)
   apply simp-all
  done

lemma bv-to-int-extend [simp]:
  assumes a: bv-msb w = b
  shows      bv-to-int (bv-extend n b w) = bv-to-int w
proof (cases bv-msb w)
  assume [simp]: bv-msb w = 0
  with a have [simp]: b = 0 by simp
  show ?thesis by (simp add: bv-to-int-def)
next
  assume [simp]: bv-msb w = 1
  with a have [simp]: b = 1 by simp
  show ?thesis
    apply (simp add: bv-to-int-def)
    apply (simp add: bv-extend-def)
    apply (induct n - length w,simp-all)
    done
qed

lemma length-nat-mono [simp]: x ≤ y ==> length-nat x ≤ length-nat y
proof (rule ccontr)
  assume xy: x ≤ y
  assume ~ length-nat x ≤ length-nat y
  hence lxly: length-nat y < length-nat x
    by simp
  hence length-nat y < (LEAST n. x < 2 ^ n)
    by (simp add: length-nat-def)
  hence ~ x < 2 ^ length-nat y
    by (rule not-less-Least)
  hence xx: 2 ^ length-nat y ≤ x
    by simp
  have yy: y < 2 ^ length-nat y
    apply (simp add: length-nat-def)
```

```
    apply (rule LeastI)
    apply (subgoal-tac y < 2 ^ y,assumption)
    apply (cases 0 ≤ y)
    apply (induct y,simp-all)
    done
  with xx have y < x by simp
  with xy show False by simp
qed

lemma length-nat-mono-int [simp]: x ≤ y ==> length-nat x ≤ length-nat y
  by (rule length-nat-mono) arith

lemma length-nat-pos [simp,intro!]: 0 < x ==> 0 < length-nat x
  by (simp add: length-nat-non0)

lemma length-int-mono-gt0: [| 0 ≤ x ; x ≤ y |] ==> length-int x ≤ length-int y
  by (cases x = 0) (simp-all add: length-int-gt0 nat-le-eq-zle)

lemma length-int-mono-lt0: [| x ≤ y ; y ≤ 0 |] ==> length-int y ≤ length-int x
  by (cases y = 0) (simp-all add: length-int-lt0)

lemmas [simp] = length-nat-non0

lemma nat-to-bv (number-of Int.Pls) = []
  by simp

consts
  fast-bv-to-nat-helper :: [bit list, int] => int
primrec
  fast-bv-to-nat-Nil: fast-bv-to-nat-helper [] k = k
  fast-bv-to-nat-Cons: fast-bv-to-nat-helper (b#bs) k =
    fast-bv-to-nat-helper bs ((bit-case Int.Bit0 Int.Bit1 b) k)

declare fast-bv-to-nat-helper.simps [code del]

lemma fast-bv-to-nat-Cons0: fast-bv-to-nat-helper (0#bs) bin =
    fast-bv-to-nat-helper bs (Int.Bit0 bin)
  by simp

lemma fast-bv-to-nat-Cons1: fast-bv-to-nat-helper (1#bs) bin =
    fast-bv-to-nat-helper bs (Int.Bit1 bin)
  by simp

lemma fast-bv-to-nat-def:
  bv-to-nat bs == number-of (fast-bv-to-nat-helper bs Int.Pls)
proof (simp add: bv-to-nat-def)
  have ∀ bin. ¬ (neg (number-of bin :: int)) ⟶ (foldl (%bn b. 2 * bn + bitval b)
(number-of bin) bs) = number-of (fast-bv-to-nat-helper bs bin)
    apply (induct bs,simp)
```

```
    apply (case-tac a,simp-all)
    done
  thus foldl (λbn b. 2 * bn + bitval b) 0 bs ≡ number-of (fast-bv-to-nat-helper bs
Int.Pls)
    by (simp del: nat-numeral-0-eq-0 add: nat-numeral-0-eq-0 [symmetric])
qed

declare fast-bv-to-nat-Cons [simp del]
declare fast-bv-to-nat-Cons0 [simp]
declare fast-bv-to-nat-Cons1 [simp]

setup ⟪
(*comments containing lcp are the removal of fast-bv-of-nat*)
let
  fun is-const-bool (Const(True,-)) = true
    | is-const-bool (Const(False,-)) = true
    | is-const-bool - = false
  fun is-const-bit (Const(Word.bit.Zero,-)) = true
    | is-const-bit (Const(Word.bit.One,-)) = true
    | is-const-bit - = false
  fun num-is-usable (Const(@{const-name Int.Pls},-)) = true
    | num-is-usable (Const(@{const-name Int.Min},-)) = false
    | num-is-usable (Const(@{const-name Int.Bit0},-) $ w) =
        num-is-usable w
    | num-is-usable (Const(@{const-name Int.Bit1},-) $ w) =
        num-is-usable w
    | num-is-usable - = false
  fun vec-is-usable (Const(List.list.Nil,-)) = true
    | vec-is-usable (Const(List.list.Cons,-) $ b $ bs) =
        vec-is-usable bs andalso is-const-bit b
    | vec-is-usable - = false
  (*lcp** val fast1-th = PureThy.get-thm thy Word.fast-nat-to-bv-def*)
  val fast2-th = @{thm Word.fast-bv-to-nat-def};
  (*lcp** fun f sg thms (Const(Word.nat-to-bv,-) $ (Const(@{const-name Int.number-of},-)
$ t)) =
    if num-is-usable t
     then SOME (Drule.cterm-instantiate [(cterm-of sg (Var ((w, 0), @{typ int})),
cterm-of sg t)] fast1-th)
      else NONE
    | f - - - = NONE *)
  fun g sg thms (Const(Word.bv-to-nat,-) $ (t as (Const(List.list.Cons,-) $ - $ -)))
=
        if vec-is-usable t then
       SOME (Drule.cterm-instantiate [(cterm-of sg (Var((bs,0),Type(List.list,[Type(Word.bit,[])]))),cterm-of
sg t)] fast2-th)
        else NONE
    | g - - - = NONE
  (*lcp** val simproc1 = Simplifier.simproc thy nat-to-bv [Word.nat-to-bv (number-of
w)] f *)
```

val simproc2 = Simplifier.simproc @{theory} bv-to-nat [Word.bv-to-nat (x # xs)] g
in
Simplifier.map-simpset (fn ss => ss addsimprocs [(∗lcp∗simproc1,∗)simproc2])
end⟩⟩

declare *bv-to-nat1* [*simp del*]
declare *bv-to-nat-helper* [*simp del*]

definition
bv-mapzip :: [*bit* => *bit* => *bit*,*bit list*, *bit list*] => *bit list* **where**
bv-mapzip f w1 w2 =
(*let g* = *bv-extend* (*max* (*length w1*) (*length w2*)) **0**
in map (*split f*) (*zip* (*g w1*) (*g w2*)))

lemma *bv-length-bv-mapzip* [*simp*]:
length (*bv-mapzip f w1 w2*) = *max* (*length w1*) (*length w2*)
by (*simp add*: *bv-mapzip-def Let-def split*: *split-max*)

lemma *bv-mapzip-Nil* [*simp*]: *bv-mapzip f* [] [] = []
by (*simp add*: *bv-mapzip-def Let-def*)

lemma *bv-mapzip-Cons* [*simp*]: *length w1* = *length w2* ==>
bv-mapzip f (*x*#*w1*) (*y*#*w2*) = *f x y* # *bv-mapzip f w1 w2*
by (*simp add*: *bv-mapzip-def Let-def*)

end

68 Order-Relation: Orders as Relations

theory *Order-Relation*
imports *Main*
begin

68.1 Orders on a set

definition *preorder-on A r* ≡ *refl-on A r* ∧ *trans r*

definition *partial-order-on A r* ≡ *preorder-on A r* ∧ *antisym r*

definition *linear-order-on A r* ≡ *partial-order-on A r* ∧ *total-on A r*

definition *strict-linear-order-on A r* ≡ *trans r* ∧ *irrefl r* ∧ *total-on A r*

definition *well-order-on A r* ≡ *linear-order-on A r* ∧ *wf*(*r* − *Id*)

lemmas *order-on-defs* =
preorder-on-def partial-order-on-def linear-order-on-def

strict-linear-order-on-def well-order-on-def

lemma *preorder-on-empty*[*simp*]: *preorder-on* {} {}
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*)

68.2 Orders on the field

abbreviation *Refl r* ≡ *refl-on* (*Field r*) *r*

abbreviation *Preorder r* ≡ *preorder-on* (*Field r*) *r*

abbreviation *Partial-order r* ≡ *partial-order-on* (*Field r*) *r*

abbreviation *Total r* ≡ *total-on* (*Field r*) *r*

abbreviation *Linear-order r* ≡ *linear-order-on* (*Field r*) *r*

abbreviation *Well-order r* ≡ *well-order-on* (*Field r*) *r*

lemma *subset-Image-Image-iff*:
⟦ *Preorder r*; *A* ⊆ *Field r*; *B* ⊆ *Field r*⟧ ⟹

r `` $A \subseteq r$ `` $B \longleftrightarrow (\forall a{\in}A.\exists b{\in}B.\ (b,a){:}r)$
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* $r]\!] \Longrightarrow r$ `` $\{a\} \subseteq r$ `` $\{b\} \longleftrightarrow (b,a){:}r$
by(*simp add*:*subset-Image-Image-iff*)

lemma *Refl-antisym-eq-Image1-Image1-iff*:
$[\![$*Refl* r; *antisym* r; a:*Field* r; b:*Field* $r]\!] \Longrightarrow r$ `` $\{a\} = r$ `` $\{b\} \longleftrightarrow a{=}b$
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* $r]\!] \Longrightarrow r$ `` $\{a\} = r$ `` $\{b\} \longleftrightarrow a{=}b$
by(*auto simp*:*order-on-defs Refl-antisym-eq-Image1-Image1-iff*)

68.3 Orders on a type

abbreviation *strict-linear-order* $\equiv$ *strict-linear-order-on UNIV*

abbreviation *linear-order* $\equiv$ *linear-order-on UNIV*

abbreviation *well-order* $r \equiv$ *well-order-on UNIV*

end

69 Zorn: Zorn's Lemma

theory *Zorn*
imports *Order-Relation Main*
begin

definition *chain-subset* :: *'a set set* $\Rightarrow$ *bool* ($chain_{\subseteq}$) **where**
$chain_{\subseteq}\ C \equiv \forall A{\in}C.\forall B{\in}C.\ A \subseteq B \vee B \subseteq A$

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

definition
chain :: *'a set set => 'a set set set* **where**
chain S $= \{F.\ F \subseteq S$ & $chain_{\subseteq}\ F\}$

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

definition
maxchain :: *'a set set => 'a set set set* **where**

maxchain S = {*c*. *c* ∈ *chain S* & *super S c* = {}}

definition
succ :: ['*a set set*,'*a set set*] => '*a set set* **where**
succ S c =
(*if c* ∉ *chain S* | *c* ∈ *maxchain S*
then c else SOME c'. *c'* ∈ *super S c*)

inductive-set
TFin :: '*a set set* => '*a set set set*
for *S* :: '*a set set*
where
succI: *x* ∈ *TFin S* ==> *succ S x* ∈ *TFin S*
| *Pow-UnionI*: *Y* ∈ *Pow*(*TFin S*) ==> *Union*(*Y*) ∈ *TFin S*

69.1 Mathematical Preamble

lemma *Union-lemma0*:
(∀ *x* ∈ *C*. *x* ⊆ *A* | *B* ⊆ *x*) ==> *Union*(*C*) ⊆ *A* | *B* ⊆ *Union*(*C*)
by *blast*

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

lemma *Abrial-axiom1*: *x* ⊆ *succ S x*
apply (*auto simp add*: *succ-def super-def maxchain-def*)
apply (*rule contrapos-np*, *assumption*)
apply (*rule-tac Q*=λ*S*. *xa* ∈ *S* **in** *someI2*, *blast*+)
done

lemmas *TFin-UnionI* = *TFin.Pow-UnionI* [*OF PowI*]

lemma *TFin-induct*:
assumes *H*: *n* ∈ *TFin S*
and *I*: !!*x*. *x* ∈ *TFin S* ==> *P x* ==> *P* (*succ S x*)
!!*Y*. *Y* ⊆ *TFin S* ==> *Ball Y P* ==> *P*(*Union Y*)
shows *P n* **using** *H*
apply (*induct rule*: *TFin.induct* [**where** *P*=*P*])
apply (*blast intro*: *I*)+
done

lemma *succ-trans*: *x* ⊆ *y* ==> *x* ⊆ *succ S y*
apply (*erule subset-trans*)
apply (*rule Abrial-axiom1*)
done

Lemma 1 of section 3.1

lemma *TFin-linear-lemma1*:
[| *n* ∈ *TFin S*; *m* ∈ *TFin S*;
∀ *x* ∈ *TFin S*. *x* ⊆ *m* −−> *x* = *m* | *succ S x* ⊆ *m*
|] ==> *n* ⊆ *m* | *succ S m* ⊆ *n*
apply (*erule TFin-induct*)

apply (*erule-tac* [*2*] *Union-lemma0*)
apply (*blast del*: *subsetI intro*: *succ-trans*)
done

Lemma 2 of section 3.2

lemma *TFin-linear-lemma2*:
m ∈ *TFin S* ==> ∀ *n* ∈ *TFin S*. *n* ⊆ *m* −−> *n*=*m* | *succ S n* ⊆ *m*
apply (*erule TFin-induct*)
apply (*rule impI* [*THEN ballI*])

case split using *TFin-linear-lemma1*

apply (*rule-tac n1* = *n* **and** *m1* = *x* **in** *TFin-linear-lemma1* [*THEN disjE*], *assumption*+)
apply (*drule-tac x* = *n* **in** *bspec*, *assumption*)
apply (*blast del*: *subsetI intro*: *succ-trans*, *blast*)

second induction step

apply (*rule impI* [*THEN ballI*])
apply (*rule Union-lemma0* [*THEN disjE*])
apply (*rule-tac* [*3*] *disjI2*)
prefer *2* **apply** *blast*
apply (*rule ballI*)
apply (*rule-tac n1* = *n* **and** *m1* = *x* **in** *TFin-linear-lemma1* [*THEN disjE*], *assumption*+, *auto*)
apply (*blast intro*!: *Abrial-axiom1* [*THEN subsetD*])
done

Re-ordering the premises of Lemma 2

lemma *TFin-subsetD*:
[| *n* ⊆ *m*; *m* ∈ *TFin S*; *n* ∈ *TFin S* |] ==> *n*=*m* | *succ S n* ⊆ *m*
by (*rule TFin-linear-lemma2* [*rule-format*])

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

lemma *TFin-subset-linear*: [| *m* ∈ *TFin S*; *n* ∈ *TFin S*|] ==> *n* ⊆ *m* | *m* ⊆ *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* ∈ *TFin S*; *m* ∈ *TFin S*; *m* = *succ S m* |] ==> *n* ⊆ *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

69.2 Hausdorff's Theorem: Every Set Contains a Maximal Chain.

NB: We assume the partial ordering is $\subseteq$, the subset relation!

lemma *empty-set-mem-chain*: $(\{\} :: 'a\ set\ set) \in chain\ S$
by (*unfold chain-def chain-subset-def*) *auto*

lemma *super-subset-chain*: $super\ S\ c \subseteq chain\ S$
by (*unfold super-def*) *blast*

lemma *maxchain-subset-chain*: $maxchain\ S \subseteq chain\ S$
by (*unfold maxchain-def*) *blast*

lemma *mem-super-Ex*: $c \in chain\ S - maxchain\ S ==> EX\ d.\ d \in super\ S\ c$
by (*unfold super-def maxchain-def*) *auto*

lemma *select-super*:
$c \in chain\ S - maxchain\ S ==> (\epsilon\ c'.\ c' : super\ S\ c) : super\ S\ c$
apply (*erule mem-super-Ex* [*THEN exE*])
apply (*rule someI2* [**where** $Q = \%X.\ X : super\ S\ c$], *auto*)
done

lemma *select-not-equals*:
$c \in chain\ S - maxchain\ S ==> (\epsilon\ c'.\ c' : super\ S\ c) \neq c$
apply (*rule notI*)
apply (*drule select-super*)
apply (*simp add*: *super-def less-le*)
done

lemma *succI3*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c = (\epsilon\ c'.\ c' : super\ S\ c)$
by (*unfold succ-def*) (*blast intro*!: *if-not-P*)

lemma *succ-not-equals*: $c \in chain\ S - maxchain\ S ==> succ\ S\ c \neq c$
apply (*frule succI3*)
apply (*simp* (*no-asm-simp*))
apply (*rule select-not-equals*, *assumption*)
done

lemma *TFin-chain-lemma4*: $c \in TFin\ S \Longrightarrow (c :: 'a\ set\ set)$: *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*: $\exists\, c.\ (c :: 'a\ set\ set)$: *maxchain S*
apply (*rule-tac x* = *Union* (*TFin S*) **in** *exI*)
apply (*rule classical*)
apply (*subgoal-tac succ S* (*Union* (*TFin S*)) = *Union* (*TFin S*))
prefer *2*
apply (*blast intro*!: *TFin-UnionI equal-succ-Union* [*THEN iffD2*, *symmetric*])
apply (*cut-tac subset-refl* [*THEN TFin-UnionI*, *THEN TFin-chain-lemma4*])
apply (*drule DiffI* [*THEN succ-not-equals*], *blast*+)
done

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

lemma *chain-extend*:
[| $c \in$ *chain S*; $z \in S$; $\forall x \in c.\ x \subseteq (z :: 'a\ set)$ |] $\Longrightarrow \{z\}\ Un\ c \in$ *chain S*
by (*unfold chain-def chain-subset-def*) *blast*

lemma *chain-Union-upper*: [| $c \in$ *chain S*; $x \in c$ |] $\Longrightarrow x \subseteq Union(c)$
by *auto*

lemma *chain-ball-Union-upper*: $c \in$ *chain S* $\Longrightarrow \forall x \in c.\ x \subseteq Union(c)$
by *auto*

lemma *maxchain-Zorn*:
[| $c \in$ *maxchain S*; $u \in S$; $Union(c) \subseteq u$ |] $\Longrightarrow Union(c) = u$
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* $\Longrightarrow \exists y \in S.\ \forall z \in S.\ y \subseteq z \longrightarrow y = z$
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

69.4 Alternative version of Zorn's Lemma

lemma *Zorn-Lemma2*:
∀ *c* ∈ *chain S*. ∃ *y* ∈ *S*. ∀ *x* ∈ *c*. *x* ⊆ *y*
==> ∃ *y* ∈ *S*. ∀ *x* ∈ *S*. (*y* :: 'a *set*) ⊆ *x* --> *y* = *x*
apply (*cut-tac Hausdorff maxchain-subset-chain*)
apply (*erule exE*)
apply (*drule subsetD*, *assumption*)
apply (*drule bspec*, *assumption*, *erule bexE*)
apply (*rule-tac x* = *y* **in** *bexI*)
prefer *2* **apply** *assumption*
apply *clarify*
apply (*rule ccontr*)
apply (*frule-tac z* = *x* **in** *chain-extend*)
apply (*assumption*, *blast*)
apply (*unfold maxchain-def super-def less-le*)
apply (*blast elim*!: *equalityCE*)
done

Various other lemmas

lemma *chainD*: [| *c* ∈ *chain S*; *x* ∈ *c*; *y* ∈ *c* |] ==> *x* ⊆ *y* | *y* ⊆ *x*
by (*unfold chain-def chain-subset-def*) *blast*

lemma *chainD2*: !!(*c* :: 'a *set set*). *c* ∈ *chain S* ==> *c* ⊆ *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*: $C \subseteq ?S$ **and** *2*: $\forall A{\in}C.\forall B{\in}C.\ A{\subseteq}B \mid B{\subseteq}A$
let $?A = \{x{\in}Field\ r.\ \exists M{\in}C.\ M = ?B\ x\}$
have $C = ?B\ `\ ?A$ **using** *1* **by**(*auto simp*: *image-def*)
have $?A{\in}Chain\ r$
proof (*simp add*:*Chain-def*, *intro allI impI*, *elim conjE*)
fix *a b*
assume $a \in Field\ r\ ?B\ a \in C\ b \in Field\ r\ ?B\ b \in C$
hence $?B\ a \subseteq ?B\ b \vee ?B\ b \subseteq ?B\ a$ **using** *2* **by** *auto*
thus $(a,\ b) \in r \vee (b,\ a) \in r$ **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* $\forall a{\in}?A.\ (a,u) : r$ **using** *u* **by** *auto*
have $\forall A{\in}C.\ A \subseteq r\hat{}{-}1\ ``\ \{u\}$ (**is** *?P u*)
proof *auto*
fix *a B* **assume** *aB*: *B*:*C a*:*B*
with *1* **obtain** *x* **where** *x*:*Field r* $B = r\hat{}{-}1\ ``\ \{x\}$ **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* $B = r\hat{}{-}1\ ``\ \{m\}$
$\forall x{\in}Field\ r.\ B \subseteq r\hat{}{-}1\ ``\ \{x\} \longrightarrow B = r\hat{}{-}1\ ``\ \{x\}$
by *auto*
hence $\forall a{\in}Field\ r.\ (m,\ a) \in r \longrightarrow a = m$ **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* :: $((\prime a{*}\prime a)set * (\prime a{*}\prime a)set)set$ **where**
$init\text{-}seg\text{-}of == \{(r,s).\ r \subseteq s \wedge (\forall\ a\ b\ c.\ (a,b){:}s \wedge (b,c){:}r \longrightarrow (a,b){:}r)\}$

abbreviation *initialSegmentOf* :: $(\prime a{*}\prime a)set \Rightarrow (\prime a{*}\prime a)set \Rightarrow bool$
(**infix** *initial'-segment'-of 55*) **where**
$r\ initial\text{-}segment\text{-}of\ s == (r,s){:}init\text{-}seg\text{-}of$

lemma *refl-on-init-seg-of*[*simp*]: *r initial-segment-of r*
by(*simp add*:*init-seg-of-def*)

lemma *trans-init-seg-of*:
r initial-segment-of s $\Longrightarrow$ *s initial-segment-of t* $\Longrightarrow$ *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 $\Longrightarrow$ *s initial-segment-of r* $\Longrightarrow$ $r{=}s$

by(*auto simp*:*init-seg-of-def*)

lemma *Chain-init-seg-of-Union*:
  $R \in$ *Chain init-seg-of* $\Longrightarrow$ $r{\in}R \Longrightarrow$ *r initial-segment-of* $\bigcup R$
by(*auto simp add*:*init-seg-of-def Chain-def Ball-def*) *blast*

lemma *chain-subset-trans-Union*:
  $chain_{\subseteq}$ $R \Longrightarrow \forall\, r{\in}R.$ *trans r* $\Longrightarrow$ *trans*($\bigcup R$)
apply(*auto simp add*:*chain-subset-def*)
apply(*simp* (*no-asm-use*) *add*:*trans-def*)
apply (*metis subsetD*)
done

lemma *chain-subset-antisym-Union*:
  $chain_{\subseteq}$ $R \Longrightarrow \forall\, r{\in}R.$ *antisym r* $\Longrightarrow$ *antisym*($\bigcup R$)
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-seg-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 ∨ 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 ⟹ p − s initial-segment-of q − s
unfolding init-seg-of-def by blast

lemma Chain-inits-DiffI:
 R ∈ Chain init-seg-of ⟹ {r − s |r. r ∈ R} ∈ Chain init-seg-of
unfolding Chain-def by (blast intro: initial-segment-of-Diff)

theorem well-ordering: ∃ r::('a*'a)set. Well-order r ∧ Field r = UNIV
proof−
— The initial segment relation on well-orders:
 let ?WO = {r::('a*'a)set. Well-order r}
 def I ≡ init-seg-of ∩ ?WO × ?WO
 have I-init: I ⊆ init-seg-of by(auto simp:I-def)
 hence subch: !!R. R : Chain I ⟹ chain⊆ R
  by(auto simp:init-seg-of-def chain-subset-def Chain-def)
 have Chain-wo: !!R r. R ∈ Chain I ⟹ r ∈ R ⟹ Well-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 ∈ Chain I
   hence Ris: R ∈ Chain init-seg-of using mono-Chain[OF I-init] by blast
   have subch: chain⊆ R using ‹R : Chain I› I-init
    by(auto simp:init-seg-of-def chain-subset-def Chain-def)
   have ∀ r∈R. Refl r ∀ r∈R. trans r ∀ r∈R. antisym r ∀ r∈R. Total r
        ∀ r∈R. wf(r−Id)
    using Chain-wo[OF ‹R ∈ Chain I›] by(simp-all add:order-on-defs)
   have Refl (⋃ R) using ‹∀ r∈R. Refl r› by(auto simp:refl-on-def)
   moreover have trans (⋃ R)
    by(rule chain-subset-trans-Union[OF subch ‹∀ r∈R. trans r›])
   moreover have antisym(⋃ R)
    by(rule chain-subset-antisym-Union[OF subch ‹∀ r∈R. antisym r›])
   moreover have Total (⋃ R)
    by(rule chain-subset-Total-Union[OF subch ‹∀ r∈R. Total r›])
   moreover have wf((⋃ R)−Id)
   proof−
```

```
    have (⋃ R)−Id = ⋃{r−Id|r. r ∈ R} by blast
    with ‹∀ r∈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 (⋃ R) by(simp add:order-on-defs)
   moreover have ∀ r ∈ R. r initial-segment-of ⋃ R using Ris
    by(simp add: Chain-init-seg-of-Union)
   ultimately have ⋃ R : ?WO ∧ (∀ r∈R. (r,⋃ R) : I)
    using mono-Chain[OF I-init] ‹R ∈ Chain I›
    by(simp (no-asm) add:I-def del:Field-Union)(metis Chain-wo subsetD)
  }
  hence 1: ∀ R ∈ Chain I. ∃ u∈Field I. ∀ r∈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: ∀ r. Well-order r ∧ (m,r):I ⟶ 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 ∉ Field m
— We assume that x is not covered and extend m at the top with x
   have m ≠ {}
   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 ≠ {} 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 ∉ Field m›
    unfolding trans-def Field-def Domain-def Range-def by blast
   moreover have antisym ?m using ‹antisym m› ‹x ∉ 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 ∉ Field m›
     by(auto simp add:wf-eq-minimal Field-def Domain-def Range-def) metis
```

```
      thus ?thesis using ‹wf(m−Id)› ‹x ∉ 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 ∉ 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 ∉ 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: ∃ 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
```

70 List-Prefix: List prefixes and postfixes

```
theory List-Prefix
imports List Main
begin
```

70.1 Prefix order on lists

```
instantiation list :: (type) order
```

begin

definition
prefix-def [*code del*]: $xs \leq ys = (\exists zs.\ ys = xs\ @\ zs)$

definition
strict-prefix-def [*code del*]: $xs < ys = (xs \leq ys \wedge xs \neq (ys::'a\ list))$

instance
by *intro-classes* (*auto simp add*: *prefix-def strict-prefix-def*)

end

lemma *prefixI* [*intro?*]: $ys = xs\ @\ zs \Longrightarrow xs \leq ys$
unfolding *prefix-def* **by** *blast*

lemma *prefixE* [*elim?*]:
assumes $xs \leq ys$
obtains *zs* **where** $ys = xs\ @\ zs$
using *assms* **unfolding** *prefix-def* **by** *blast*

lemma *strict-prefixI'* [*intro?*]: $ys = xs\ @\ z\ \#\ zs \Longrightarrow xs < ys$
unfolding *strict-prefix-def prefix-def* **by** *blast*

lemma *strict-prefixE'* [*elim?*]:
assumes $xs < ys$
obtains *z zs* **where** $ys = xs\ @\ z\ \#\ zs$
proof −
from ‹$xs < ys$› **obtain** *us* **where** $ys = xs\ @\ us$ **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?*]: $xs \leq ys \Longrightarrow xs \neq ys \Longrightarrow xs < (ys::'a\ list)$
unfolding *strict-prefix-def* **by** *blast*

lemma *strict-prefixE* [*elim?*]:
fixes *xs ys* :: $'a\ list$
assumes $xs < ys$
obtains $xs \leq ys$ **and** $xs \neq ys$
using *assms* **unfolding** *strict-prefix-def* **by** *blast*

70.2 Basic properties of prefixes

theorem *Nil-prefix* [*iff*]: $[] \leq xs$
by (*simp add*: *prefix-def*)

theorem *prefix-Nil* [*simp*]: $(xs \leq []) = (xs = [])$
by (*induct xs*) (*simp-all add*: *prefix-def*)

lemma *prefix-snoc* [*simp*]: $(xs \leq ys \mathbin{@} [y]) = (xs = ys \mathbin{@} [y] \vee xs \leq ys)$
proof
assume $xs \leq ys \mathbin{@} [y]$
then obtain *zs* **where** *zs*: $ys \mathbin{@} [y] = xs \mathbin{@} zs$ **..**
show $xs = ys \mathbin{@} [y] \vee xs \leq ys$
by (*metis append-Nil2 butlast-append butlast-snoc prefixI zs*)
next
assume $xs = ys \mathbin{@} [y] \vee xs \leq ys$
then show $xs \leq ys \mathbin{@} [y]$
by (*metis order-eq-iff strict-prefixE strict-prefixI' xt1*(7))
qed

lemma *Cons-prefix-Cons* [*simp*]: $(x \# xs \leq y \# ys) = (x = y \wedge xs \leq ys)$
by (*auto simp add*: *prefix-def*)

lemma *same-prefix-prefix* [*simp*]: $(xs \mathbin{@} ys \leq xs \mathbin{@} zs) = (ys \leq zs)$
by (*induct xs*) *simp-all*

lemma *same-prefix-nil* [*iff*]: $(xs \mathbin{@} ys \leq xs) = (ys = [])$
by (*metis append-Nil2 append-self-conv order-eq-iff prefixI*)

lemma *prefix-prefix* [*simp*]: $xs \leq ys \longrightarrow xs \leq ys \mathbin{@} zs$
by (*metis order-le-less-trans prefixI strict-prefixE strict-prefixI*)

lemma *append-prefixD*: $xs \mathbin{@} ys \leq zs \Longrightarrow xs \leq zs$
by (*auto simp add*: *prefix-def*)

theorem *prefix-Cons*: $(xs \leq y \# ys) = (xs = [] \vee (\exists zs.\ xs = y \# zs \wedge zs \leq ys))$
by (*cases xs*) (*auto simp add*: *prefix-def*)

theorem *prefix-append*:
$(xs \leq ys \mathbin{@} zs) = (xs \leq ys \vee (\exists us.\ xs = ys \mathbin{@} us \wedge us \leq zs))$
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*:
$xs \leq ys \longrightarrow length\ xs < length\ ys \longrightarrow xs \mathbin{@} [ys\ !\ length\ xs] \leq ys$
unfolding *prefix-def*
by (*metis Cons-eq-appendI append-eq-appendI append-eq-conv-conj*
eq-Nil-appendI nth-drop')

theorem *prefix-length-le*: $xs \leq ys \longrightarrow length\ xs \leq length\ ys$
by (*auto simp add*: *prefix-def*)

lemma *prefix-same-cases*:

$(xs_1::'a\ list) \leq ys \Longrightarrow xs_2 \leq ys \Longrightarrow xs_1 \leq xs_2 \vee xs_2 \leq xs_1$
unfolding *prefix-def* **by** (*metis append-eq-append-conv2*)

lemma *set-mono-prefix*: $xs \leq ys \Longrightarrow set\ xs \subseteq set\ ys$
by (*auto simp add*: *prefix-def*)

lemma *take-is-prefix*: $take\ n\ xs \leq xs$
unfolding *prefix-def* **by** (*metis append-take-drop-id*)

lemma *map-prefixI*: $xs \leq ys \Longrightarrow map\ f\ xs \leq map\ f\ ys$
by (*auto simp*: *prefix-def*)

lemma *prefix-length-less*: $xs < ys \Longrightarrow length\ xs < length\ ys$
by (*auto simp*: *strict-prefix-def prefix-def*)

lemma *strict-prefix-simps* [*simp*]:
$xs < [] = False$
$[] < (x\ \#\ xs) = True$
$(x\ \#\ xs) < (y\ \#\ ys) = (x = y \wedge xs < ys)$
by (*simp-all add*: *strict-prefix-def cong*: *conj-cong*)

lemma *take-strict-prefix*: $xs < ys \Longrightarrow take\ n\ xs < ys$
apply (*induct n arbitrary*: *xs ys*)
apply (*case-tac ys*, *simp-all*)[*1*]
apply (*metis order-less-trans strict-prefixI take-is-prefix*)
done

lemma *not-prefix-cases*:
assumes *pfx*: $\neg\ ps \leq ls$
obtains
(*c1*) $ps \neq []$ **and** $ls = []$
| (*c2*) *a as x xs* **where** $ps = a\#as$ **and** $ls = x\#xs$ **and** $x = a$ **and** $\neg\ as \leq xs$
| (*c3*) *a as x xs* **where** $ps = a\#as$ **and** $ls = x\#xs$ **and** $x \neq a$
proof (*cases ps*)
case *Nil* **then show** *?thesis* **using** *pfx* **by** *simp*
next
case (*Cons a as*)
note $c = \langle ps = a\#as \rangle$
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 $\neg\ as \leq xs$ **using** *pfx c Cons True* **by** *simp*
with *c Cons True* **show** *?thesis* **by** (*rule c2*)
next

```
      case False
      with c Cons show ?thesis by (rule c3)
    qed
  qed
qed

lemma not-prefix-induct [consumes 1, case-names Nil Neq Eq]:
  assumes np: ¬ ps ≤ ls
    and base: ⋀x xs. P (x#xs) []
    and r1: ⋀x xs y ys. x ≠ y ⟹ P (x#xs) (y#ys)
    and r2: ⋀x xs y ys. ⟦ x = y; ¬ xs ≤ ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
  shows P ps ls using np
proof (induct ls arbitrary: ps)
  case Nil then show ?case
    by (auto simp: neq-Nil-conv elim!: not-prefix-cases intro!: base)
next
  case (Cons y ys)
  then have npfx: ¬ ps ≤ (y # ys) by simp
  then obtain x xs where pv: ps = x # xs
    by (rule not-prefix-cases) auto
  show ?case by (metis Cons.hyps Cons-prefix-Cons npfx pv r1 r2)
qed
```

70.3 Parallel lists

```
definition
  parallel :: 'a list => 'a list => bool  (infixl ∥ 50) where
  (xs ∥ ys) = (¬ xs ≤ ys ∧ ¬ ys ≤ xs)

lemma parallelI [intro]: ¬ xs ≤ ys ==> ¬ ys ≤ xs ==> xs ∥ ys
  unfolding parallel-def by blast

lemma parallelE [elim]:
  assumes xs ∥ ys
  obtains ¬ xs ≤ ys ∧ ¬ ys ≤ xs
  using assms unfolding parallel-def by blast

theorem prefix-cases:
  obtains xs ≤ ys | ys < xs | xs ∥ ys
  unfolding parallel-def strict-prefix-def by blast

theorem parallel-decomp:
  xs ∥ ys ==> ∃ as b bs c cs. b ≠ c ∧ xs = as @ b # bs ∧ 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 ≤ 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 ≤ 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
```

70.4 Postfix order on lists

```
definition
  postfix :: 'a list => 'a list => bool  ((-/ >>= -) [51, 50] 50) where
  (xs >>= ys) = (∃zs. xs = zs @ ys)

lemma postfixI [intro?]: xs = zs @ ys ==> xs >>= ys
  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

70.5 Executable code

lemma *less-eq-code* [*code*]:
([]::'*a*::{*eq*, *ord*} *list*) ≤ *xs* ⟷ *True*
(*x*::'*a*::{*eq*, *ord*}) # *xs* ≤ [] ⟷ *False*
(*x*::'*a*::{*eq*, *ord*}) # *xs* ≤ *y* # *ys* ⟷ *x* = *y* ∧ *xs* ≤ *ys*
by *simp-all*

lemma *less-code* [*code*]:
xs < ([]::'*a*::{*eq*, *ord*} *list*) ⟷ *False*
[] < (*x*::'*a*::{*eq*, *ord*})# *xs* ⟷ *True*
(*x*::'*a*::{*eq*, *ord*}) # *xs* < *y* # *ys* ⟷ *x* = *y* ∧ *xs* < *ys*
unfolding *strict-prefix-def* **by** *auto*

lemmas [*code*] = *postfix-to-prefix*

end

71 List-lexord: Lexicographic order on lists

theory *List-lexord*
imports *List Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def [*code del*]: (*xs*::('*a*::*ord*) *list*) < *ys* ⟷ (*xs*, *ys*) ∈ *lexord* {(*u*,*v*). *u* < *v*}

definition
list-le-def [*code del*]: (*xs*::('*a*::*ord*) *list*) ≤ *ys* ⟷ (*xs* < *ys* ∨ *xs* = *ys*)

instance ..

end

instance *list* :: (*order*) *order*
proof
fix *xs* :: *'a list*
show *xs* ≤ *xs* **by** (*simp add*: *list-le-def*)
next
fix *xs ys zs* :: *'a list*
assume *xs* ≤ *ys* **and** *ys* ≤ *zs*
then show *xs* ≤ *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* ≤ *ys* **and** *ys* ≤ *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*]: $\neg\ (x < [])$
by (*unfold list-less-def*) *simp*

lemma *Nil-less-Cons* [*simp*]: $[] < a \# x$
by (*unfold list-less-def*) *simp*

lemma *Cons-less-Cons* [*simp*]: $a \# x < b \# y \longleftrightarrow a < b \vee a = b \wedge x < y$
by (*unfold list-less-def*) *simp*

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
by (*unfold list-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

72 Sublist-Order: Sublist Ordering

theory *Sublist-Order*
imports *Main*
begin

This theory defines sublist ordering on lists. A list *ys* is a sublist of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

72.1 Definitions and basic lemmas

instantiation *list* :: (*type*) *order*
begin

inductive *less-eq-list* **where**
  empty [*simp*, *intro*!]: [] ≤ *xs*
  | *drop*: *ys* ≤ *xs* ⟹ *ys* ≤ *x* # *xs*
  | *take*: *ys* ≤ *xs* ⟹ *x* # *ys* ≤ *x* # *xs*

lemmas *ileq-empty* = *empty*
lemmas *ileq-drop* = *drop*
lemmas *ileq-take* = *take*

lemma *ileq-cases* [*cases set*, *case-names empty drop take*]:
  assumes *xs* ≤ *ys*
    and *xs* = [] ⟹ *P*
    and ⋀*z zs*. *ys* = *z* # *zs* ⟹ *xs* ≤ *zs* ⟹ *P*
    and ⋀*x zs ws*. *xs* = *x* # *zs* ⟹ *ys* = *x* # *ws* ⟹ *zs* ≤ *ws* ⟹ *P*
  shows *P*
  using *assms* **by** (*blast elim*: *less-eq-list.cases*)

lemma *ileq-induct* [*induct set*, *case-names empty drop take*]:
  assumes *xs* ≤ *ys*
    and ⋀*zs*. *P* [] *zs*
    and ⋀*z zs ws*. *ws* ≤ *zs* ⟹ *P ws zs* ⟹ *P ws* (*z* # *zs*)
    and ⋀*z zs ws*. *ws* ≤ *zs* ⟹ *P ws zs* ⟹ *P* (*z* # *ws*) (*z* # *zs*)
  shows *P xs ys*
  using *assms* **by** (*induct rule*: *less-eq-list.induct*) *blast*+

definition
  [*code del*]: (*xs* :: 'a *list*) < *ys* ⟷ *xs* ≤ *ys* ∧ ¬ *ys* ≤ *xs*

lemma *ileq-length*: *xs* ≤ *ys* ⟹ *length xs* ≤ *length ys*
  by (*induct rule*: *ileq-induct*) *auto*
lemma *ileq-below-empty* [*simp*]: *xs* ≤ [] ⟷ *xs* = []
  by (*auto dest*: *ileq-length*)

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*!: *ileq-empty ileq-drop ileq-take*)
next
  fix *xs ys* :: 'a *list*

  { **fix** *n*
    have !!*l l'*. ⟦*l*≤*l'*; *l'*≤*l*; *n*=*length l* + *length l'*⟧ ⟹ *l*=*l'*
    proof (*induct n rule*: *nat-less-induct*)

```
    case (1 n l l') from 1.prems(1) show ?case proof (cases rule: ileq-cases)
     case empty with 1.prems(2) show ?thesis by auto
    next
     case (drop a l2') with 1.prems(2) have length l'≤length l length l ≤ length
l2' 1+length l2' = length l' by (auto dest: ileq-length)
     hence False by simp thus ?thesis ..
    next
     case (take a l1' l2') hence LEN': length l1' + length l2' < length l + length
l' by simp
     from 1.prems have LEN: length l' = length l by (auto dest!: ileq-length)
      from 1.prems(2) show ?thesis proof (cases rule: ileq-cases[case-names
empty' drop' take'])
       case empty' with take LEN show ?thesis by simp
      next
       case (drop' ah l2h) with take LEN have length l1' ≤ length l2h 1+length
l2h = length l2' length l2' = length l1' by (auto dest: ileq-length)
       hence False by simp thus ?thesis ..
      next
       case (take' ah l1h l2h)
       with take have 2: ah=a l1h=l2' l2h=l1' l1' ≤ l2' l2' ≤ l1' by auto
       with LEN' 1.hyps 1.prems(3) have l1'=l2' by blast
       with take 2 show ?thesis by simp
      qed
    qed
   qed
  }
  moreover assume xs ≤ ys ys ≤ xs
  ultimately show xs = ys by blast
next
  fix xs ys zs :: 'a list
  {
   fix n
   have !!x y z. [[x ≤ y; y ≤ z; n=length x + length y + length z]] ⟹ x ≤ z
   proof (induct rule: nat-less-induct[case-names I])
    case (I n x y z)
    from I.prems(2) show ?case proof (cases rule: ileq-cases)
     case empty with I.prems(1) show ?thesis by auto
    next
     case (drop a z') hence length x + length y + length z' < length x + length
y + length z by simp
     with I.hyps I.prems(3,1) drop(2) have x≤z' by blast
     with drop(1) show ?thesis by (auto intro: ileq-drop)
    next
       case (take a y' z') from I.prems(1) show ?thesis proof (cases rule:
ileq-cases[case-names empty' drop' take'])
      case empty' thus ?thesis by auto
     next
       case (drop' ah y'h) with take have x≤y' y'≤z' length x + length y' +
length z' < length x + length y + length z by auto
```

```
        with I.hyps I.prems(3) have x≤z' by (blast)
        with take(2) show ?thesis  by (auto intro: ileq-drop)
      next
        case (take' ah x' y'h) with take have 2: x=a#x' x'≤y' y'≤z' length x'
+ length y' + length z' < length x + length y + length z by auto
        with I.hyps I.prems(3) have x'≤z' by blast
        with 2 take(2) show ?thesis by (auto intro: ileq-take)
      qed
    qed
  qed
 }
 moreover assume xs ≤ ys ys ≤ zs
 ultimately show xs ≤ zs by blast
qed

end

lemmas ileq-intros = ileq-empty ileq-drop ileq-take

lemma ileq-drop-many: xs ≤ ys ⟹ xs ≤ zs @ ys
 by (induct zs) (auto intro: ileq-drop)
lemma ileq-take-many: xs ≤ ys ⟹ zs @ xs ≤ zs @ ys
 by (induct zs) (auto intro: ileq-take)

lemma ileq-same-length: xs ≤ ys ⟹ length xs = length ys ⟹ xs = ys
 by (induct rule: ileq-induct) (auto dest: ileq-length)
lemma ileq-same-append [simp]: x # xs ≤ xs ⟷ False
 by (auto dest: ileq-length)

lemma ilt-length [intro]:
 assumes xs < ys
 shows length xs < length ys
proof −
 from assms have xs ≤ ys and xs ≠ ys by (simp-all add: less-le)
 moreover with ileq-length have length xs ≤ length ys by auto
 ultimately show ?thesis by (auto intro: ileq-same-length)
qed

lemma ilt-empty [simp]: [] < xs ⟷ xs ≠ []
 by (unfold less-list-def, auto)
lemma ilt-emptyI: xs ≠ [] ⟹ [] < xs
 by (unfold less-list-def, auto)
lemma ilt-emptyD: [] < xs ⟹ xs ≠ []
 by (unfold less-list-def, auto)
lemma ilt-below-empty[simp]: xs < [] ⟹ False
 by (auto dest: ilt-length)
lemma ilt-drop: xs < ys ⟹ xs < x # ys
 by (unfold less-le) (auto intro: ileq-intros)
lemma ilt-take: xs < ys ⟹ x # xs < x # ys
```

by (*unfold less-le*) (*auto intro*: *ileq-intros*)
lemma *ilt-drop-many*: $xs < ys \Longrightarrow xs < zs \mathbin{@} ys$
by (*induct zs*) (*auto intro*: *ilt-drop*)
lemma *ilt-take-many*: $xs < ys \Longrightarrow zs \mathbin{@} xs < zs \mathbin{@} ys$
by (*induct zs*) (*auto intro*: *ilt-take*)

72.2 Appending elements

lemma *ileq-rev-take*: $xs \leq ys \Longrightarrow xs \mathbin{@} [x] \leq ys \mathbin{@} [x]$
by (*induct rule*: *ileq-induct*) (*auto intro*: *ileq-intros ileq-drop-many*)
lemma *ilt-rev-take*: $xs < ys \Longrightarrow xs \mathbin{@} [x] < ys \mathbin{@} [x]$
by (*unfold less-le*) (*auto dest*: *ileq-rev-take*)
lemma *ileq-rev-drop*: $xs \leq ys \Longrightarrow xs \leq ys \mathbin{@} [x]$
by (*induct rule*: *ileq-induct*) (*auto intro*: *ileq-intros*)
lemma *ileq-rev-drop-many*: $xs \leq ys \Longrightarrow xs \leq ys \mathbin{@} zs$
by (*induct zs rule*: *rev-induct*) (*auto dest*: *ileq-rev-drop*)

72.3 Relation to standard list operations

lemma *ileq-map*: $xs \leq ys \Longrightarrow map\ f\ xs \leq map\ f\ ys$
by (*induct rule*: *ileq-induct*) (*auto intro*: *ileq-intros*)
lemma *ileq-filter-left*[*simp*]: $filter\ f\ xs \leq xs$
by (*induct xs*) (*auto intro*: *ileq-intros*)
lemma *ileq-filter*: $xs \leq ys \Longrightarrow filter\ f\ xs \leq filter\ f\ ys$
by (*induct rule*: *ileq-induct*) (*auto intro*: *ileq-intros*)

end

References

[1] Abrial and Laffitte. Towards the mechanization of the proofs of some classical theorems of set theory. Unpublished.

[2] J. Avigad and K. Donnelly. Formalizing O notation in Isabelle/HOL. In D. Basin and M. Rusiowitch, editors, *Automated Reasoning: second international conference, IJCAR 2004*, pages 357–371. Springer, 2004.

[3] A. Oberschelp. *Rekursionstheorie*. BI-Wissenschafts-Verlag, 1993.

[4] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.