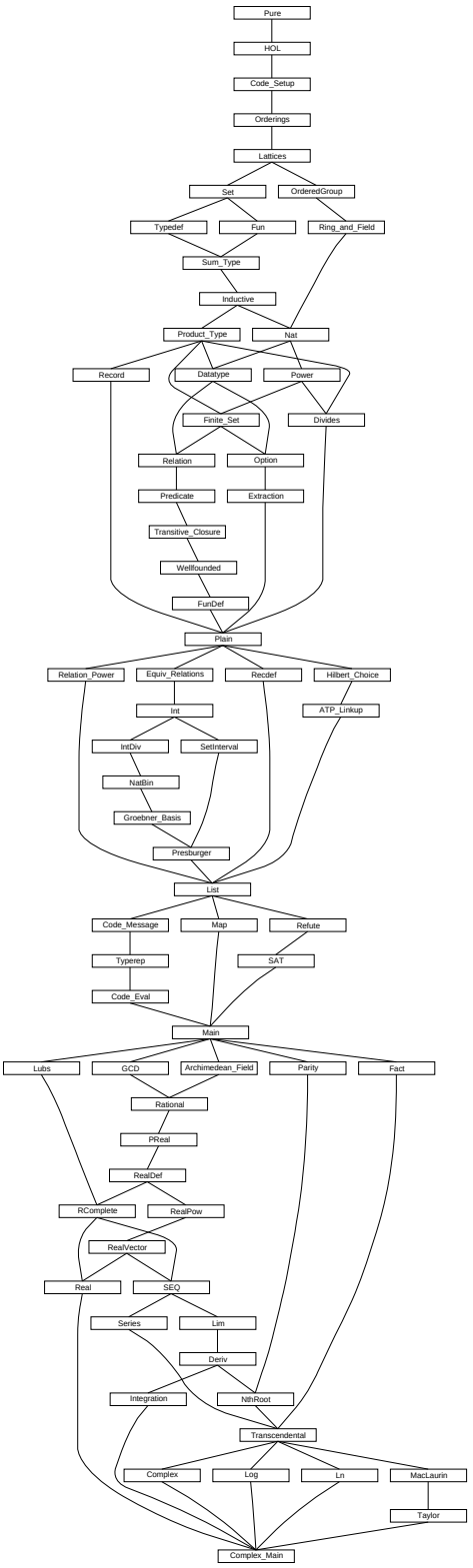
Isabelle/HOL — Higher-Order Logic

April 19, 2009

Contents


Pure
HOL
Code_Setup
Orderings
Lattices
Set
OrderedGroup
Typedef
Fun
Ring_and_Field
Sum_Type
Inductive
Product_Type
Nat
Record
Datatype
Power
Finite_Set
Divides
Relation
Option
Predicate
Extraction
Transitive_Closure
Wellfounded
FunDef
Plain
Relation_Power
Equiv_Relations
Recdef
Hilbert_Choice
Int
ATP_Linkup
IntDiv
SetInterval
NatBin
Groebner_Basis
Presburger
List
Code_Message
Map
Refute
Typerep
SAT
Code_Eval
Main
Lubs
GCD
Archimedean_Field
Parity
Fact
Rational
PReal
RealDef
RComplete
RealPow
RealVector
Real
SEQ
Series
Lim
Deriv
Integration
NthRoot
Transcendental
Complex
Log
Ln
MacLaurin
Taylor
Complex_Main

1 HOL: The basis of Higher-Order Logic

theory *HOL*
imports *Pure*
uses
(*Tools/hologic.ML*)
~~/*src/Tools/IsaPlanner/zipper.ML*
~~/*src/Tools/IsaPlanner/isand.ML*
~~/*src/Tools/IsaPlanner/rw-tools.ML*
~~/*src/Tools/IsaPlanner/rw-inst.ML*
~~/*src/Tools/intuitionistic.ML*
~~/*src/Tools/project-rule.ML*
~~/*src/Provers/hypsubst.ML*
~~/*src/Provers/splitter.ML*
~~/*src/Provers/classical.ML*
~~/*src/Provers/blast.ML*
~~/*src/Provers/clasimp.ML*
~~/*src/Tools/coherent.ML*
~~/*src/Tools/eqsubst.ML*
~~/*src/Provers/quantifier1.ML*
(*Tools/simpdata.ML*)
~~/*src/Tools/random-word.ML*
~~/*src/Tools/atomize-elim.ML*
~~/*src/Tools/induct.ML*
(~~/*src/Tools/induct-tacs.ML*)
~~/*src/Tools/value.ML*
~~/*src/Tools/code/code-name.ML*
~~/*src/Tools/code/code-funcgr.ML*
~~/*src/Tools/code/code-wellsorted.ML*
~~/*src/Tools/code/code-thingol.ML*
~~/*src/Tools/code/code-printer.ML*
~~/*src/Tools/code/code-target.ML*
~~/*src/Tools/code/code-ml.ML*
~~/*src/Tools/code/code-haskell.ML*
~~/*src/Tools/nbe.ML*
(*Tools/recfun-codegen.ML*)
begin

setup ⟪ *Intuitionistic.method-setup iprover* ⟫

1.1 Primitive logic

1.1.1 Core syntax

classes *type*
defaultsort *type*
setup ⟪ *ObjectLogic.add-base-sort* @{*sort type*} ⟫

arities
fun :: (*type*, *type*) *type*

itself :: (*type*) *type*

global

typedecl *bool*

judgment
Trueprop :: *bool* => *prop* ((-) *5*)

consts
Not :: *bool* => *bool* (~ - [*40*] *40*)
True :: *bool*
False :: *bool*

The :: ('*a* => *bool*) => '*a*
All :: ('*a* => *bool*) => *bool* (**binder** *ALL* *10*)
Ex :: ('*a* => *bool*) => *bool* (**binder** *EX* *10*)
Ex1 :: ('*a* => *bool*) => *bool* (**binder** *EX*! *10*)
Let :: ['*a*, '*a* => '*b*] => '*b*

op = :: ['*a*, '*a*] => *bool* (**infixl** = *50*)
op & :: [*bool*, *bool*] => *bool* (**infixr** & *35*)
op | :: [*bool*, *bool*] => *bool* (**infixr** | *30*)
op --> :: [*bool*, *bool*] => *bool* (**infixr** --> *25*)

local

consts
If :: [*bool*, '*a*, '*a*] => '*a* ((*if* (-)/ *then* (-)/ *else* (-)) *10*)

1.1.2 Additional concrete syntax

notation (**output**)
op = (**infix** = *50*)

abbreviation
not-equal :: ['*a*, '*a*] => *bool* (**infixl** ~= *50*) **where**
x ~= *y* == ~ (*x* = *y*)

notation (**output**)
not-equal (**infix** ~= *50*)

notation (*xsymbols*)
Not (¬ - [*40*] *40*) **and**
op & (**infixr** ∧ *35*) **and**
op | (**infixr** ∨ *30*) **and**
op --> (**infixr** ⟶ *25*) **and**
not-equal (**infix** ≠ *50*)

notation (*HTML* **output**)
Not (¬ - [*40*] *40*) **and**
op & (**infixr** ∧ *35*) **and**
op | (**infixr** ∨ *30*) **and**
not-equal (**infix** ≠ *50*)

abbreviation (*iff*)
iff :: [*bool*, *bool*] => *bool* (**infixr** <–> *25*) **where**
A <–> *B* == *A* = *B*

notation (*xsymbols*)
iff (**infixr** ⟷ *25*)

nonterminals
letbinds letbind
case-syn cases-syn

syntax
-The :: [*pttrn*, *bool*] => *'a* ((*3THE -./ -*) [*0*, *10*] *10*)

-bind :: [*pttrn*, *'a*] => *letbind* ((*2- =/ -*) *10*)
:: *letbind* => *letbinds* (-)
-binds :: [*letbind*, *letbinds*] => *letbinds* (-;/ -)
-Let :: [*letbinds*, *'a*] => *'a* ((*let* (-)/ *in* (-)) *10*)

-case-syntax:: [*'a*, *cases-syn*] => *'b* ((*case - of/ -*) *10*)
-case1 :: [*'a*, *'b*] => *case-syn* ((*2- =>/ -*) *10*)
:: *case-syn* => *cases-syn* (-)
-case2 :: [*case-syn*, *cases-syn*] => *cases-syn* (-/ | -)

translations
THE x. P == *The* (%*x. P*)
-Let (*-binds b bs*) *e* == *-Let b* (*-Let bs e*)
let x = a in e == *Let a* (%*x. e*)

print-translation ⟪
(∗ *To avoid eta−contraction of body:* ∗)
[(*The*, *fn* [*Abs abs*] =>
let val (*x*,*t*) = *atomic-abs-tr' abs*
in Syntax.const -The $ *x* $ *t end*)]
⟫

syntax (*xsymbols*)
-case1 :: [*'a*, *'b*] => *case-syn* ((*2- ⇒/ -*) *10*)

notation (*xsymbols*)
All (**binder** ∀ *10*) **and**
Ex (**binder** ∃ *10*) **and**

Ex1 (**binder** $\exists!$ *10*)

notation (*HTML* **output**)
All (**binder** $\forall$ *10*) **and**
Ex (**binder** $\exists$ *10*) **and**
Ex1 (**binder** $\exists!$ *10*)

notation (*HOL*)
All (**binder** ! *10*) **and**
Ex (**binder** *?* *10*) **and**
Ex1 (**binder** *?!* *10*)

1.1.3 Axioms and basic definitions

axioms
refl: $t = (t::'a)$
subst: $s = t \Longrightarrow P\ s \Longrightarrow P\ t$
ext: $(!!x::'a.\ (f\ x ::'b) = g\ x) ==> (\%x.\ f\ x) = (\%x.\ g\ x)$
— Extensionality is built into the meta-logic, and this rule expresses a related property. It is an eta-expanded version of the traditional rule, and similar to the ABS rule of HOL

the-eq-trivial: $(THE\ x.\ x = a) = (a::'a)$

impI: $(P ==> Q) ==> P-->Q$
mp: $[|\ P-->Q;\ P\ |] ==> Q$

defs
True-def: $True \quad == ((\%x::bool.\ x) = (\%x.\ x))$
All-def: $All(P) \quad == (P = (\%x.\ True))$
Ex-def: $Ex(P) \quad == !Q.\ (!x.\ P\ x --> Q) --> Q$
False-def: $False \quad == (!P.\ P)$
not-def: $\sim P \quad == P-->False$
and-def: $P\ \&\ Q \quad == !R.\ (P-->Q-->R) --> R$
or-def: $P \mid Q \quad == !R.\ (P-->R) --> (Q-->R) --> R$
Ex1-def: $Ex1(P) \quad == ?\ x.\ P(x)\ \&\ (!\ y.\ P(y) --> y=x)$

axioms
iff: $(P-->Q) --> (Q-->P) --> (P=Q)$
True-or-False: $(P=True) \mid (P=False)$

defs
Let-def: $Let\ s\ f == f(s)$
if-def: $If\ P\ x\ y == THE\ z::'a.\ (P=True --> z=x)\ \&\ (P=False --> z=y)$

finalconsts
op =

op −−>
The

axiomatization
undefined :: *'a*

abbreviation (*input*)
arbitrary ≡ *undefined*

1.1.4 Generic classes and algebraic operations

class *default* =
fixes *default* :: *'a*

class *zero* =
fixes *zero* :: *'a* (*0*)

class *one* =
fixes *one* :: *'a* (*1*)

hide (**open**) *const zero one*

class *plus* =
fixes *plus* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** + *65*)

class *minus* =
fixes *minus* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** − *65*)

class *uminus* =
fixes *uminus* :: *'a* ⇒ *'a* (− - [*81*] *80*)

class *times* =
fixes *times* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** ∗ *70*)

class *inverse* =
fixes *inverse* :: *'a* ⇒ *'a*
and *divide* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** */* *70*)

class *abs* =
fixes *abs* :: *'a* ⇒ *'a*
begin

notation (*xsymbols*)
abs (|-|)

notation (*HTML* **output**)
abs (|-|)

end

class *sgn* =
  fixes *sgn* :: *'a* ⇒ *'a*

class *ord* =
  fixes *less-eq* :: *'a* ⇒ *'a* ⇒ *bool*
    and *less* :: *'a* ⇒ *'a* ⇒ *bool*
begin

notation
  less-eq (*op* <=) **and**
  less-eq ((-/ <= -) [*51*, *51*] *50*) **and**
  less (*op* <) **and**
  less ((-/ < -) [*51*, *51*] *50*)

notation (*xsymbols*)
  less-eq (*op* ≤) **and**
  less-eq ((-/ ≤ -) [*51*, *51*] *50*)

notation (*HTML* **output**)
  less-eq (*op* ≤) **and**
  less-eq ((-/ ≤ -) [*51*, *51*] *50*)

abbreviation (*input*)
  greater-eq (**infix** >= *50*) **where**
  x >= *y* ≡ *y* <= *x*

notation (*input*)
  greater-eq (**infix** ≥ *50*)

abbreviation (*input*)
  greater (**infix** > *50*) **where**
  x > *y* ≡ *y* < *x*

end

syntax
  -index1 :: *index* (1)
translations
  (*index*) 1 => (*index*) ◇

typed-print-translation ⟪
let
  fun tr' c = (*c*, *fn show-sorts* => *fn T* => *fn ts* =>
    if (*not o null*) *ts orelse T* = *dummyT orelse not* (! *show-types*) *andalso can Term.dest-Type T then raise Match*
    else Syntax.const Syntax.constrainC $ *Syntax.const c* $ *Syntax.term-of-typ show-sorts T*);
in map tr' [@{*const-syntax HOL.one*}, @{*const-syntax HOL.zero*}] *end*;

⟩⟩ — show types that are presumably too general

1.2 Fundamental rules

1.2.1 Equality

lemma *sym*: $s = t \Longrightarrow t = s$
by (*erule subst*) (*rule refl*)

lemma *ssubst*: $t = s \Longrightarrow P\ s \Longrightarrow P\ t$
by (*drule sym*) (*erule subst*)

lemma *trans*: $[\![r{=}s;\ s{=}t\ ]\!] \Longrightarrow r{=}t$
by (*erule subst*)

lemma *meta-eq-to-obj-eq*:
assumes *meq*: $A \equiv B$
shows $A = B$
by (*unfold meq*) (*rule refl*)

Useful with *erule* for proving equalities from known equalities.

lemma *box-equals*: $[\![a{=}b;\ \ a{=}c;\ \ b{=}d\ ]\!] \Longrightarrow c{=}d$
apply (*rule trans*)
apply (*rule trans*)
apply (*rule sym*)
apply *assumption*+
done

For calculational reasoning:

lemma *forw-subst*: $a = b \Longrightarrow P\ b \Longrightarrow P\ a$
by (*rule ssubst*)

lemma *back-subst*: $P\ a \Longrightarrow a = b \Longrightarrow P\ b$
by (*rule subst*)

1.2.2 Congruence rules for application

lemma *fun-cong*: $(f::'a{\Rightarrow}'b) = g \Longrightarrow f(x){=}g(x)$
apply (*erule subst*)
apply (*rule refl*)
done

lemma *arg-cong*: $x{=}y \Longrightarrow f(x){=}f(y)$
apply (*erule subst*)
apply (*rule refl*)
done

lemma *arg-cong2*: $[\![\ a = b;\ c = d\ ]\!] \Longrightarrow f\ a\ c = f\ b\ d$
apply (*erule ssubst*)+

apply (*rule refl*)
done

lemma *cong*: [| $f = g$; $(x::'a) = y$ |] ==> $f(x) = g(y)$
apply (*erule subst*)+
apply (*rule refl*)
done

1.2.3 Equality of booleans – iff

lemma *iffI*: **assumes** P ==> Q **and** Q ==> P **shows** $P=Q$
by (*iprover intro*: *iff* [*THEN mp*, *THEN mp*] *impI assms*)

lemma *iffD2*: [| $P=Q$; Q |] ==> P
by (*erule ssubst*)

lemma *rev-iffD2*: [| Q; $P=Q$ |] ==> P
by (*erule iffD2*)

lemma *iffD1*: $Q = P \Longrightarrow Q \Longrightarrow P$
by (*drule sym*) (*rule iffD2*)

lemma *rev-iffD1*: $Q \Longrightarrow Q = P \Longrightarrow P$
by (*drule sym*) (*rule rev-iffD2*)

lemma *iffE*:
assumes *major*: $P=Q$
and *minor*: [| P --> Q; Q --> P |] ==> R
shows R
by (*iprover intro*: *minor impI major* [*THEN iffD2*] *major* [*THEN iffD1*])

1.2.4 True

lemma *TrueI*: *True*
unfolding *True-def* **by** (*rule refl*)

lemma *eqTrueI*: P ==> $P = True$
by (*iprover intro*: *iffI TrueI*)

lemma *eqTrueE*: $P = True$ ==> P
by (*erule iffD2*) (*rule TrueI*)

1.2.5 Universal quantifier

lemma *allI*: **assumes** !!$x::'a$. $P(x)$ **shows** $ALL\ x.\ P(x)$
unfolding *All-def* **by** (*iprover intro*: *ext eqTrueI assms*)

lemma *spec*: $ALL\ x::'a.\ P(x)$ ==> $P(x)$
apply (*unfold All-def*)
apply (*rule eqTrueE*)

apply (*erule fun-cong*)
done

lemma *allE*:
assumes *major*: *ALL x*. *P*(*x*)
and *minor*: *P*(*x*) ==> *R*
shows *R*
by (*iprover intro*: *minor major* [*THEN spec*])

lemma *all-dupE*:
assumes *major*: *ALL x*. *P*(*x*)
and *minor*: [| *P*(*x*); *ALL x*. *P*(*x*) |] ==> *R*
shows *R*
by (*iprover intro*: *minor major major* [*THEN spec*])

1.2.6 False

Depends upon *spec*; it is impossible to do propositional logic before quantifiers!

lemma *FalseE*: *False* ==> *P*
apply (*unfold False-def*)
apply (*erule spec*)
done

lemma *False-neq-True*: *False* = *True* ==> *P*
by (*erule eqTrueE* [*THEN FalseE*])

1.2.7 Negation

lemma *notI*:
assumes *P* ==> *False*
shows ~*P*
apply (*unfold not-def*)
apply (*iprover intro*: *impI assms*)
done

lemma *False-not-True*: *False* ~= *True*
apply (*rule notI*)
apply (*erule False-neq-True*)
done

lemma *True-not-False*: *True* ~= *False*
apply (*rule notI*)
apply (*drule sym*)
apply (*erule False-neq-True*)
done

lemma *notE*: [| ~*P*; *P* |] ==> *R*
apply (*unfold not-def*)

apply (*erule mp* [*THEN FalseE*])
apply *assumption*
done

lemma *notI2*: $(P \Longrightarrow \neg\ Pa) \Longrightarrow (P \Longrightarrow Pa) \Longrightarrow \neg\ P$
by (*erule notE* [*THEN notI*]) (*erule meta-mp*)

1.2.8 Implication

lemma *impE*:
assumes $P-->Q\ P\ Q\ ==>\ R$
shows R
by (*iprover intro*: *assms mp*)

lemma *rev-mp*: $[|\ P;\ \ P\ -->\ Q\ |]\ ==>\ Q$
by (*iprover intro*: *mp*)

lemma *contrapos-nn*:
assumes *major*: $\sim Q$
and *minor*: $P==>Q$
shows $\sim P$
by (*iprover intro*: *notI minor major* [*THEN notE*])

lemma *contrapos-pn*:
assumes *major*: Q
and *minor*: $P\ ==>\ \sim Q$
shows $\sim P$
by (*iprover intro*: *notI minor major notE*)

lemma *not-sym*: $t\ \sim=\ s\ ==>\ s\ \sim=\ t$
by (*erule contrapos-nn*) (*erule sym*)

lemma *eq-neq-eq-imp-neq*: $[|\ x\ =\ a\ ;\ a\ \sim=\ b;\ b\ =\ y\ |]\ ==>\ x\ \sim=\ y$
by (*erule subst*, *erule ssubst*, *assumption*)

lemma *rev-contrapos*:
assumes *pq*: $P\ ==>\ Q$
and *nq*: $\sim Q$
shows $\sim P$
apply (*rule nq* [*THEN contrapos-nn*])
apply (*erule pq*)
done

1.2.9 Existential quantifier

lemma *exI*: $P\ x\ ==>\ EX\ x::'a.\ P\ x$
apply (*unfold Ex-def*)

apply (*iprover intro*: *allI allE impI mp*)
done

lemma *exE*:
assumes *major*: *EX x::'a. P(x)*
and *minor*: !!*x*. *P*(*x*) ==> *Q*
shows *Q*
apply (*rule major* [*unfolded Ex-def*, *THEN spec*, *THEN mp*])
apply (*iprover intro*: *impI* [*THEN allI*] *minor*)
done

1.2.10 Conjunction

lemma *conjI*: [| *P*; *Q* |] ==> *P&Q*
apply (*unfold and-def*)
apply (*iprover intro*: *impI* [*THEN allI*] *mp*)
done

lemma *conjunct1*: [| *P* & *Q* |] ==> *P*
apply (*unfold and-def*)
apply (*iprover intro*: *impI dest*: *spec mp*)
done

lemma *conjunct2*: [| *P* & *Q* |] ==> *Q*
apply (*unfold and-def*)
apply (*iprover intro*: *impI dest*: *spec mp*)
done

lemma *conjE*:
assumes *major*: *P&Q*
and *minor*: [| *P*; *Q* |] ==> *R*
shows *R*
apply (*rule minor*)
apply (*rule major* [*THEN conjunct1*])
apply (*rule major* [*THEN conjunct2*])
done

lemma *context-conjI*:
assumes *P P* ==> *Q* **shows** *P* & *Q*
by (*iprover intro*: *conjI assms*)

1.2.11 Disjunction

lemma *disjI1*: *P* ==> *P|Q*
apply (*unfold or-def*)
apply (*iprover intro*: *allI impI mp*)
done

lemma *disjI2*: *Q* ==> *P|Q*
apply (*unfold or-def*)

apply (*iprover intro*: *allI impI mp*)
done

lemma *disjE*:
  assumes *major*: $P|Q$
    and *minorP*: $P ==> R$
    and *minorQ*: $Q ==> R$
  shows R
by (*iprover intro*: *minorP minorQ impI*
            major [*unfolded or-def*, *THEN spec*, *THEN mp*, *THEN mp*])

1.2.12 Classical logic

lemma *classical*:
  assumes *prem*: $\sim P ==> P$
  shows P
apply (*rule True-or-False* [*THEN disjE*, *THEN eqTrueE*])
apply *assumption*
apply (*rule notI* [*THEN prem*, *THEN eqTrueI*])
apply (*erule subst*)
apply *assumption*
done

lemmas *ccontr* = *FalseE* [*THEN classical*, *standard*]

lemma *rev-notE*:
  assumes *premp*: P
    and *premnot*: $\sim R ==> \sim P$
  shows R
apply (*rule ccontr*)
apply (*erule notE* [*OF premnot premp*])
done

lemma *notnotD*: $\sim\sim P ==> P$
apply (*rule classical*)
apply (*erule notE*)
apply *assumption*
done

lemma *contrapos-pp*:
  assumes *p1*: Q
    and *p2*: $\sim P ==> \sim Q$
  shows P
by (*iprover intro*: *classical p1 p2 notE*)

1.2.13 Unique existence

lemma *ex1I*:

assumes $P\ a$ $!!x.\ P(x) ==> x=a$
shows $EX!\ x.\ P(x)$
by (*unfold Ex1-def*, *iprover intro*: *assms exI conjI allI impI*)

Sometimes easier to use: the premises have no shared variables. Safe!

lemma *ex-ex1I*:
assumes *ex-prem*: $EX\ x.\ P(x)$
and *eq*: $!!x\ y.\ [|\ P(x);\ P(y)\ |] ==> x=y$
shows $EX!\ x.\ P(x)$
by (*iprover intro*: *ex-prem* [*THEN exE*] *ex1I eq*)

lemma *ex1E*:
assumes *major*: $EX!\ x.\ P(x)$
and *minor*: $!!x.\ [|\ P(x);\ \ ALL\ y.\ P(y) --> y=x\ |] ==> R$
shows R
apply (*rule major* [*unfolded Ex1-def*, *THEN exE*])
apply (*erule conjE*)
apply (*iprover intro*: *minor*)
done

lemma *ex1-implies-ex*: $EX!\ x.\ P\ x ==> EX\ x.\ P\ x$
apply (*erule ex1E*)
apply (*rule exI*)
apply *assumption*
done

1.2.14 THE: definite description operator

lemma *the-equality*:
assumes *prema*: $P\ a$
and *premx*: $!!x.\ P\ x ==> x=a$
shows $(THE\ x.\ P\ x) = a$
apply (*rule trans* [*OF - the-eq-trivial*])
apply (*rule-tac f* = *The* **in** *arg-cong*)
apply (*rule ext*)
apply (*rule iffI*)
apply (*erule premx*)
apply (*erule ssubst*, *rule prema*)
done

lemma *theI*:
assumes $P\ a$ **and** $!!x.\ P\ x ==> x=a$
shows $P\ (THE\ x.\ P\ x)$
by (*iprover intro*: *assms the-equality* [*THEN ssubst*])

lemma *theI'*: $EX!\ x.\ P\ x ==> P\ (THE\ x.\ P\ x)$
apply (*erule ex1E*)
apply (*erule theI*)
apply (*erule allE*)

apply (*erule mp*)
apply *assumption*
done

lemma *theI2*:
assumes *P a* !!*x*. *P x* ==> *x*=*a* !!*x*. *P x* ==> *Q x*
shows *Q* (*THE x*. *P x*)
by (*iprover intro*: *assms theI*)

lemma *the1I2*: **assumes** *EX*! *x*. *P x* ⋀*x*. *P x* ⟹ *Q x* **shows** *Q* (*THE x*. *P x*)
by(*iprover intro*:*assms*(*2*) *theI2*[**where** *P*=*P* **and** *Q*=*Q*] *ex1E*[*OF assms*(*1*)]
elim:*allE impE*)

lemma *the1-equality* [*elim?*]: [| *EX*!*x*. *P x*; *P a* |] ==> (*THE x*. *P x*) = *a*
apply (*rule the-equality*)
apply *assumption*
apply (*erule ex1E*)
apply (*erule all-dupE*)
apply (*drule mp*)
apply *assumption*
apply (*erule ssubst*)
apply (*erule allE*)
apply (*erule mp*)
apply *assumption*
done

lemma *the-sym-eq-trivial*: (*THE y*. *x*=*y*) = *x*
apply (*rule the-equality*)
apply (*rule refl*)
apply (*erule sym*)
done

1.2.15 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
assumes ~*Q* ==> *P* **shows** *P*|*Q*
apply (*rule classical*)
apply (*iprover intro*: *assms disjI1 disjI2 notI elim*: *notE*)
done

lemma *excluded-middle*: ~*P* | *P*
by (*iprover intro*: *disjCI*)

case distinction as a natural deduction rule. Note that ¬ *P* is the second case, not the first

lemma *case-split* [*case-names True False*]:
assumes *prem1*: *P* ==> *Q*

and *prem2*: ~*P* ==> *Q*
shows *Q*
apply (*rule excluded-middle* [*THEN disjE*])
apply (*erule prem2*)
apply (*erule prem1*)
done

lemma *impCE*:
assumes *major*: *P*-->*Q*
and *minor*: ~*P* ==> *R* *Q* ==> *R*
shows *R*
apply (*rule excluded-middle* [*of P*, *THEN disjE*])
apply (*iprover intro*: *minor major* [*THEN mp*])+
done

lemma *impCE'*:
assumes *major*: *P*-->*Q*
and *minor*: *Q* ==> *R* ~*P* ==> *R*
shows *R*
apply (*rule excluded-middle* [*of P*, *THEN disjE*])
apply (*iprover intro*: *minor major* [*THEN mp*])+
done

lemma *iffCE*:
assumes *major*: *P*=*Q*
and *minor*: [| *P*; *Q* |] ==> *R* [| ~*P*; ~*Q* |] ==> *R*
shows *R*
apply (*rule major* [*THEN iffE*])
apply (*iprover intro*: *minor elim*: *impCE notE*)
done

lemma *exCI*:
assumes *ALL x*. ~*P*(*x*) ==> *P*(*a*)
shows *EX x*. *P*(*x*)
apply (*rule ccontr*)
apply (*iprover intro*: *assms exI allI notI notE* [*of* ∃ *x*. *P x*])
done

1.2.16 Intuitionistic Reasoning

lemma *impE'*:
assumes *1*: *P* --> *Q*
and *2*: *Q* ==> *R*
and *3*: *P* --> *Q* ==> *P*
shows *R*
proof −

from *3* **and** *1* **have** P **.**
with *1* **have** Q **by** (*rule impE*)
with *2* **show** R **.**
qed

lemma *allE'*:
assumes *1*: $ALL\ x.\ P\ x$
and *2*: $P\ x \Longrightarrow ALL\ x.\ P\ x \Longrightarrow Q$
shows Q
proof −
from *1* **have** $P\ x$ **by** (*rule spec*)
from *this* **and** *1* **show** Q **by** (*rule 2*)
qed

lemma *notE'*:
assumes *1*: $\sim P$
and *2*: $\sim P \Longrightarrow P$
shows R
proof −
from *2* **and** *1* **have** P **.**
with *1* **show** R **by** (*rule notE*)
qed

lemma *TrueE*: $True \Longrightarrow P \Longrightarrow P$ **.**
lemma *notFalseE*: $\sim False \Longrightarrow P \Longrightarrow P$ **.**

lemmas [*Pure.elim*!] = *disjE iffE FalseE conjE exE TrueE notFalseE*
and [*Pure.intro*!] = *iffI conjI impI TrueI notI allI refl*
and [*Pure.elim 2*] = *allE notE' impE'*
and [*Pure.intro*] = *exI disjI2 disjI1*

lemmas [*trans*] = *trans*
and [*sym*] = *sym not-sym*
and [*Pure.elim?*] = *iffD1 iffD2 impE*

use *Tools/hologic.ML*

1.2.17 Atomizing meta-level connectives

axiomatization where
eq-reflection: $x = y \Longrightarrow x \equiv y$

lemma *atomize-all* [*atomize*]: $(!!x.\ P\ x) == Trueprop\ (ALL\ x.\ P\ x)$
proof
assume $!!x.\ P\ x$
then show $ALL\ x.\ P\ x$ **..**
next
assume $ALL\ x.\ P\ x$
then show $!!x.\ P\ x$ **by** (*rule allE*)

qed

lemma *atomize-imp* [*atomize*]: (*A* ==> *B*) == *Trueprop* (*A* --> *B*)
proof
  assume *r*: *A* ==> *B*
  show *A* --> *B* **by** (*rule impI*) (*rule r*)
next
  assume *A* --> *B* **and** *A*
  then show *B* **by** (*rule mp*)
qed

lemma *atomize-not*: (*A* ==> *False*) == *Trueprop* (~*A*)
proof
  assume *r*: *A* ==> *False*
  show ~*A* **by** (*rule notI*) (*rule r*)
next
  assume ~*A* **and** *A*
  then show *False* **by** (*rule notE*)
qed

lemma *atomize-eq* [*atomize*]: (*x* == *y*) == *Trueprop* (*x* = *y*)
proof
  assume *x* == *y*
  show *x* = *y* **by** (*unfold* ‹*x* == *y*›) (*rule refl*)
next
  assume *x* = *y*
  then show *x* == *y* **by** (*rule eq-reflection*)
qed

lemma *atomize-conj* [*atomize*]: (*A* &&& *B*) == *Trueprop* (*A* & *B*)
proof
  assume *conj*: *A* &&& *B*
  show *A* & *B*
  proof (*rule conjI*)
    from *conj* **show** *A* **by** (*rule conjunctionD1*)
    from *conj* **show** *B* **by** (*rule conjunctionD2*)
  qed
next
  assume *conj*: *A* & *B*
  show *A* &&& *B*
  proof −
    from *conj* **show** *A* **..**
    from *conj* **show** *B* **..**
  qed
qed

lemmas [*symmetric*, *rulify*] = *atomize-all atomize-imp*
  and [*symmetric*, *defn*] = *atomize-all atomize-imp atomize-eq*

1.2.18 Atomizing elimination rules

setup *AtomizeElim.setup*

lemma *atomize-exL*[*atomize-elim*]: (!!*x*. *P x* ==> *Q*) == ((*EX x*. *P x*) ==> *Q*)
by *rule iprover+*

lemma *atomize-conjL*[*atomize-elim*]: (*A* ==> *B* ==> *C*) == (*A* & *B* ==> *C*)
by *rule iprover+*

lemma *atomize-disjL*[*atomize-elim*]: ((*A* ==> *C*) ==> (*B* ==> *C*) ==> *C*)
== ((*A* | *B* ==> *C*) ==> *C*)
by *rule iprover+*

lemma *atomize-elimL*[*atomize-elim*]: (!!*B*. (*A* ==> *B*) ==> *B*) == *Trueprop A*
..

1.3 Package setup

1.3.1 Classical Reasoner setup

lemma *imp-elim*: *P* --> *Q* ==> (~ *R* ==> *P*) ==> (*Q* ==> *R*) ==> *R*
by (*rule classical*) *iprover*

lemma *swap*: ~ *P* ==> (~ *R* ==> *P*) ==> *R*
by (*rule classical*) *iprover*

lemma *thin-refl*:
⋀*X*. ⟦ *x*=*x*; *PROP W* ⟧ ⟹ *PROP W* **.**

ML ⟪
structure Hypsubst = HypsubstFun(
struct
structure Simplifier = Simplifier
val dest-eq = HOLogic.dest-eq
val dest-Trueprop = HOLogic.dest-Trueprop
val dest-imp = HOLogic.dest-imp
val eq-reflection = @{thm eq-reflection}
val rev-eq-reflection = @{thm meta-eq-to-obj-eq}
val imp-intr = @{thm impI}
val rev-mp = @{thm rev-mp}
val subst = @{thm subst}
val sym = @{thm sym}
val thin-refl = @{thm thin-refl};
val prop-subst = @{lemma PROP P t ==> PROP prop (x = t ==> PROP P x)
by (unfold prop-def) (drule eq-reflection, unfold)}
end);
open Hypsubst;

```
structure Classical = ClassicalFun(
struct
  val imp-elim = @{thm imp-elim}
  val not-elim = @{thm notE}
  val swap = @{thm swap}
  val classical = @{thm classical}
  val sizef = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
end);

structure BasicClassical: BASIC-CLASSICAL = Classical;
open BasicClassical;

ML-Antiquote.value claset
  (Scan.succeed Classical.local-claset-of (ML-Context.the-local-context ()));

structure ResAtpset = NamedThmsFun(val name = atp val description = ATP
rules);

structure ResBlacklist = NamedThmsFun(val name = noatp val description = the-
orems blacklisted for ATP);
⟩⟩
```

ResBlacklist holds theorems blacklisted to sledgehammer. These theorems typically produce clauses that are prolific (match too many equality or membership literals) and relate to seldom-used facts. Some duplicate other rules.

```
setup ⟨⟨
let
  (*prevent substitution on bool*)
  fun hyp-subst-tac' i thm = if i <= Thm.nprems-of thm andalso
    Term.exists-Const (fn (op =, Type (-, [T, -])) => T <> Type (bool, []) | - =>
false)
      (nth (Thm.prems-of thm) (i - 1)) then Hypsubst.hyp-subst-tac i thm else
no-tac thm;
in
  Hypsubst.hypsubst-setup
  #> ContextRules.addSWrapper (fn tac => hyp-subst-tac' ORELSE' tac)
  #> Classical.setup
  #> ResAtpset.setup
  #> ResBlacklist.setup
end
⟩⟩

declare iffI [intro!]
  and notI [intro!]
  and impI [intro!]
  and disjCI [intro!]
  and conjI [intro!]
  and TrueI [intro!]
```

and *refl* [*intro*!]

declare *iffCE* [*elim*!]
and *FalseE* [*elim*!]
and *impCE* [*elim*!]
and *disjE* [*elim*!]
and *conjE* [*elim*!]
and *conjE* [*elim*!]

declare *ex-ex1I* [*intro*!]
and *allI* [*intro*!]
and *the-equality* [*intro*]
and *exI* [*intro*]

declare *exE* [*elim*!]
allE [*elim*]

ML ⟪ *val HOL-cs* = @{*claset*} ⟫

lemma *contrapos-np*: ~ *Q* ==> (~ *P* ==> *Q*) ==> *P*
apply (*erule swap*)
apply (*erule* (*1*) *meta-mp*)
done

declare *ex-ex1I* [*rule del*, *intro*! *2*]
and *ex1I* [*intro*]

lemmas [*intro?*] = *ext*
and [*elim?*] = *ex1-implies-ex*

lemma *alt-ex1E* [*elim*!]:
assumes *major*: ∃!*x*. *P x*
and *prem*: ⋀*x*. ⟦ *P x*; ∀ *y y'*. *P y* ∧ *P y'* ⟶ *y* = *y'* ⟧ ⟹ *R*
shows *R*
apply (*rule ex1E* [*OF major*])
apply (*rule prem*)
apply (*tactic* ⟪ *ares-tac* @{*thms allI*} *1* ⟫)+
apply (*tactic* ⟪ *etac* (*Classical.dup-elim* @{*thm allE*}) *1* ⟫)
apply *iprover*
done

ML ⟪
structure Blast = *BlastFun*
(
type claset = *Classical.claset*
val equality-name = @{*const-name op* =}
val not-name = @{*const-name Not*}
val notE = @{*thm notE*}

val ccontr = @{thm ccontr}
val contr-tac = Classical.contr-tac
val dup-intr = Classical.dup-intr
val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
val rep-cs = Classical.rep-cs
val cla-modifiers = Classical.cla-modifiers
val cla-meth' = Classical.cla-meth'
);
val blast-tac = Blast.blast-tac;
⟩⟩

setup *Blast.setup*

1.3.2 Simplifier

lemma *eta-contract-eq*: $(\%s.\ f\ s) = f$ **..**

lemma *simp-thms*:
shows *not-not*: $(\sim\ \sim\ P) = P$
and *Not-eq-iff*: $((\sim P) = (\sim Q)) = (P = Q)$
and
$(P\ \sim= Q) = (P = (\sim Q))$
$(P \mid \sim P) = True \quad (\sim P \mid P) = True$
$(x = x) = True$
and *not-True-eq-False*: $(\neg\ True) = False$
and *not-False-eq-True*: $(\neg\ False) = True$
and
$(\sim P) \sim= P \ \ P \sim= (\sim P)$
$(True=P) = P$
and *eq-True*: $(P = True) = P$
and $(False=P) = (\sim P)$
and *eq-False*: $(P = False) = (\neg\ P)$
and
$(True \longrightarrow P) = P \ \ (False \longrightarrow P) = True$
$(P \longrightarrow True) = True \ \ (P \longrightarrow P) = True$
$(P \longrightarrow False) = (\sim P) \ \ (P \longrightarrow \sim P) = (\sim P)$
$(P\ \&\ True) = P \ \ (True\ \&\ P) = P$
$(P\ \&\ False) = False \ \ (False\ \&\ P) = False$
$(P\ \&\ P) = P \ \ (P\ \&\ (P\ \&\ Q)) = (P\ \&\ Q)$
$(P\ \&\ \sim P) = False \quad (\sim P\ \&\ P) = False$
$(P \mid True) = True \ \ (True \mid P) = True$
$(P \mid False) = P \ \ (False \mid P) = P$
$(P \mid P) = P \ \ (P \mid (P \mid Q)) = (P \mid Q)$ **and**
$(ALL\ x.\ P) = P \ \ (EX\ x.\ P) = P \ \ EX\ x.\ x=t \ \ EX\ x.\ t=x$
— needed for the one-point-rule quantifier simplification procs
— essential for termination!! **and**
$!!P.\ (EX\ x.\ x=t\ \&\ P(x)) = P(t)$
$!!P.\ (EX\ x.\ t=x\ \&\ P(x)) = P(t)$
$!!P.\ (ALL\ x.\ x=t \longrightarrow P(x)) = P(t)$

!!P. $(ALL\ x.\ t{=}x\ {-}{-}{>}\ P(x)) = P(t)$
by (*blast*, *blast*, *blast*, *blast*, *blast*, *iprover*+)

lemma *disj-absorb*: $(A \mid A) = A$
by *blast*

lemma *disj-left-absorb*: $(A \mid (A \mid B)) = (A \mid B)$
by *blast*

lemma *conj-absorb*: $(A \,\&\, A) = A$
by *blast*

lemma *conj-left-absorb*: $(A \,\&\, (A \,\&\, B)) = (A \,\&\, B)$
by *blast*

lemma *eq-ac*:
shows *eq-commute*: $(a{=}b) = (b{=}a)$
and *eq-left-commute*: $(P{=}(Q{=}R)) = (Q{=}(P{=}R))$
and *eq-assoc*: $((P{=}Q){=}R) = (P{=}(Q{=}R))$ **by** (*iprover*, *blast*+)
lemma *neq-commute*: $(a^{\sim}{=}b) = (b^{\sim}{=}a)$ **by** *iprover*

lemma *conj-comms*:
shows *conj-commute*: $(P\&Q) = (Q\&P)$
and *conj-left-commute*: $(P\&(Q\&R)) = (Q\&(P\&R))$ **by** *iprover*+
lemma *conj-assoc*: $((P\&Q)\&R) = (P\&(Q\&R))$ **by** *iprover*

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

lemma *disj-comms*:
shows *disj-commute*: $(P|Q) = (Q|P)$
and *disj-left-commute*: $(P|(Q|R)) = (Q|(P|R))$ **by** *iprover*+
lemma *disj-assoc*: $((P|Q)|R) = (P|(Q|R))$ **by** *iprover*

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

lemma *conj-disj-distribL*: $(P\&(Q|R)) = (P\&Q \mid P\&R)$ **by** *iprover*
lemma *conj-disj-distribR*: $((P|Q)\&R) = (P\&R \mid Q\&R)$ **by** *iprover*

lemma *disj-conj-distribL*: $(P|(Q\&R)) = ((P|Q) \,\&\, (P|R))$ **by** *iprover*
lemma *disj-conj-distribR*: $((P\&Q)|R) = ((P|R) \,\&\, (Q|R))$ **by** *iprover*

lemma *imp-conjR*: $(P\ {-}{-}{>}\ (Q\&R)) = ((P{-}{-}{>}Q) \,\&\, (P{-}{-}{>}R))$ **by** *iprover*
lemma *imp-conjL*: $((P\&Q)\ {-}{-}{>}R) \ = (P\ {-}{-}{>}\ (Q\ {-}{-}{>}\ R))$ **by** *iprover*
lemma *imp-disjL*: $((P|Q)\ {-}{-}{>}\ R) = ((P{-}{-}{>}R)\&(Q{-}{-}{>}R))$ **by** *iprover*

These two are specialized, but *imp-disj-not1* is useful in *Auth/Yahalom*.

lemma *imp-disj-not1*: $(P\ {-}{-}{>}\ Q \mid R) = (^{\sim}Q\ {-}{-}{>}\ P\ {-}{-}{>}\ R)$ **by** *blast*
lemma *imp-disj-not2*: $(P\ {-}{-}{>}\ Q \mid R) = (^{\sim}R\ {-}{-}{>}\ P\ {-}{-}{>}\ Q)$ **by** *blast*

lemma *imp-disj1*: $((P\text{--}{>}Q)|R) = (P\text{--}{>}\ Q|R)$ **by** *blast*
lemma *imp-disj2*: $(Q|(P\text{--}{>}R)) = (P\text{--}{>}\ Q|R)$ **by** *blast*

lemma *imp-cong*: $(P = P') ==> (P' ==> (Q = Q')) ==> ((P \text{ --> } Q) = (P' \text{ --> } Q'))$
by *iprover*

lemma *de-Morgan-disj*: $(\sim(P \mid Q)) = (\sim P \ \& \ \sim Q)$ **by** *iprover*
lemma *de-Morgan-conj*: $(\sim(P \ \& \ Q)) = (\sim P \mid \sim Q)$ **by** *blast*
lemma *not-imp*: $(\sim(P \text{ --> } Q)) = (P \ \& \ \sim Q)$ **by** *blast*
lemma *not-iff*: $(P \sim= Q) = (P = (\sim Q))$ **by** *blast*
lemma *disj-not1*: $(\sim P \mid Q) = (P \text{ --> } Q)$ **by** *blast*
lemma *disj-not2*: $(P \mid \sim Q) = (Q \text{ --> } P)$ — changes orientation :-(
by *blast*
lemma *imp-conv-disj*: $(P \text{ --> } Q) = ((\sim P) \mid Q)$ **by** *blast*

lemma *iff-conv-conj-imp*: $(P = Q) = ((P \text{ --> } Q) \ \& \ (Q \text{ --> } P))$ **by** *iprover*

lemma *cases-simp*: $((P \text{ --> } Q) \ \& \ (\sim P \text{ --> } Q)) = Q$
— Avoids duplication of subgoals after *split-if*, when the true and false
— cases boil down to the same thing.
by *blast*

lemma *not-all*: $(\sim\ (!\ x.\ P(x))) = (?\ x.\sim P(x))$ **by** *blast*
lemma *imp-all*: $((!\ x.\ P\ x) \text{ --> } Q) = (?\ x.\ P\ x \text{ --> } Q)$ **by** *blast*
lemma *not-ex*: $(\sim\ (?\ x.\ P(x))) = (!\ x.\sim P(x))$ **by** *iprover*
lemma *imp-ex*: $((?\ x.\ P\ x) \text{ --> } Q) = (!\ x.\ P\ x \text{ --> } Q)$ **by** *iprover*
lemma *all-not-ex*: $(ALL\ x.\ P\ x) = (\sim\ (EX\ x.\ \sim\ P\ x\ ))$ **by** *blast*

declare *All-def* [*noatp*]

lemma *ex-disj-distrib*: $(?\ x.\ P(x) \mid Q(x)) = ((?\ x.\ P(x)) \mid (?\ x.\ Q(x)))$ **by** *iprover*
lemma *all-conj-distrib*: $(!x.\ P(x) \ \& \ Q(x)) = ((!\ x.\ P(x)) \ \& \ (!\ x.\ Q(x)))$ **by** *iprover*

The & congruence rule: not included by default! May slow rewrite proofs down by as much as 50%

lemma *conj-cong*:
$(P = P') ==> (P' ==> (Q = Q')) ==> ((P \ \& \ Q) = (P' \ \& \ Q'))$
by *iprover*

lemma *rev-conj-cong*:
$(Q = Q') ==> (Q' ==> (P = P')) ==> ((P \ \& \ Q) = (P' \ \& \ Q'))$
by *iprover*

The | congruence rule: not included by default!

lemma *disj-cong*:
$(P = P') ==> (\sim P' ==> (Q = Q')) ==> ((P \mid Q) = (P' \mid Q'))$

by *blast*

if-then-else rules

lemma *if-True*: (*if True then x else y*) = *x*
by (*unfold if-def*) *blast*

lemma *if-False*: (*if False then x else y*) = *y*
by (*unfold if-def*) *blast*

lemma *if-P*: *P* ==> (*if P then x else y*) = *x*
by (*unfold if-def*) *blast*

lemma *if-not-P*: ~*P* ==> (*if P then x else y*) = *y*
by (*unfold if-def*) *blast*

lemma *split-if*: *P* (*if Q then x else y*) = ((*Q* --> *P*(*x*)) & (~*Q* --> *P*(*y*)))
apply (*rule case-split* [*of Q*])
apply (*simplesubst if-P*)
prefer *3* **apply** (*simplesubst if-not-P*, *blast*+)
done

lemma *split-if-asm*: *P* (*if Q then x else y*) = (~((*Q* & ~*P x*) | (~*Q* & ~*P y*)))
by (*simplesubst split-if*, *blast*)

lemmas *if-splits* [*noatp*] = *split-if split-if-asm*

lemma *if-cancel*: (*if c then x else x*) = *x*
by (*simplesubst split-if*, *blast*)

lemma *if-eq-cancel*: (*if x = y then y else x*) = *x*
by (*simplesubst split-if*, *blast*)

lemma *if-bool-eq-conj*: (*if P then Q else R*) = ((*P*-->*Q*) & (~*P*-->*R*))
— This form is useful for expanding *if*s on the RIGHT of the ==> symbol.
by (*rule split-if*)

lemma *if-bool-eq-disj*: (*if P then Q else R*) = ((*P*&*Q*) | (~*P*&*R*))
— And this form is useful for expanding *if*s on the LEFT.
apply (*simplesubst split-if*, *blast*)
done

lemma *Eq-TrueI*: *P* ==> *P* == *True* **by** (*unfold atomize-eq*) *iprover*
lemma *Eq-FalseI*: ~*P* ==> *P* == *False* **by** (*unfold atomize-eq*) *iprover*

let rules for simproc

lemma *Let-folded*: *f x* ≡ *g x* ⟹ *Let x f* ≡ *Let x g*
by (*unfold Let-def*)

lemma *Let-unfold*: $f\ x \equiv g \Longrightarrow$ *Let* $x\ f \equiv g$
by (*unfold Let-def*)

The following copy of the implication operator is useful for fine-tuning congruence rules. It instructs the simplifier to simplify its premise.

constdefs
simp-implies :: [*prop*, *prop*] => *prop* (**infixr** =*simp*=> *1*)
[*code del*]: *simp-implies* $\equiv$ *op* ==>

lemma *simp-impliesI*:
assumes *PQ*: (*PROP P* $\Longrightarrow$ *PROP Q*)
shows *PROP P* =*simp*=> *PROP Q*
apply (*unfold simp-implies-def*)
apply (*rule PQ*)
apply *assumption*
done

lemma *simp-impliesE*:
assumes *PQ*: *PROP P* =*simp*=> *PROP Q*
and *P*: *PROP P*
and *QR*: *PROP Q* $\Longrightarrow$ *PROP R*
shows *PROP R*
apply (*rule QR*)
apply (*rule PQ* [*unfolded simp-implies-def*])
apply (*rule P*)
done

lemma *simp-implies-cong*:
assumes *PP'* :*PROP P* == *PROP P'*
and *P'QQ'*: *PROP P'* ==> (*PROP Q* == *PROP Q'*)
shows (*PROP P* =*simp*=> *PROP Q*) == (*PROP P'* =*simp*=> *PROP Q'*)
proof (*unfold simp-implies-def*, *rule equal-intr-rule*)
assume *PQ*: *PROP P* $\Longrightarrow$ *PROP Q*
and *P'*: *PROP P'*
from *PP'* [*symmetric*] **and** *P'* **have** *PROP P*
by (*rule equal-elim-rule1*)
then have *PROP Q* **by** (*rule PQ*)
with *P'QQ'* [*OF P'*] **show** *PROP Q'* **by** (*rule equal-elim-rule1*)
next
assume *P'Q'*: *PROP P'* $\Longrightarrow$ *PROP Q'*
and *P*: *PROP P*
from *PP'* **and** *P* **have** *P'*: *PROP P'* **by** (*rule equal-elim-rule1*)
then have *PROP Q'* **by** (*rule P'Q'*)
with *P'QQ'* [*OF P'*, *symmetric*] **show** *PROP Q*
by (*rule equal-elim-rule1*)
qed

lemma *uncurry*:
assumes $P \longrightarrow Q \longrightarrow R$

shows $P \wedge Q \longrightarrow R$
using *assms* **by** *blast*

lemma *iff-allI*:
assumes $\bigwedge x.\ P\ x = Q\ x$
shows $(\forall x.\ P\ x) = (\forall x.\ Q\ x)$
using *assms* **by** *blast*

lemma *iff-exI*:
assumes $\bigwedge x.\ P\ x = Q\ x$
shows $(\exists x.\ P\ x) = (\exists x.\ Q\ x)$
using *assms* **by** *blast*

lemma *all-comm*:
$(\forall x\ y.\ P\ x\ y) = (\forall y\ x.\ P\ x\ y)$
by *blast*

lemma *ex-comm*:
$(\exists x\ y.\ P\ x\ y) = (\exists y\ x.\ P\ x\ y)$
by *blast*

use *Tools/simpdata.ML*
ML ⟪ *open Simpdata* ⟫

setup ⟪
Simplifier.method-setup Splitter.split-modifiers
#> Simplifier.map-simpset (K Simpdata.simpset-simprocs)
#> Splitter.setup
#> clasimp-setup
#> EqSubst.setup
⟫

Simproc for proving $(y = x) == False$ from premise $\sim(x = y)$:

simproc-setup *neq* $(x = y) =$ ⟪ *fn - =>*
let
val neq-to-EQ-False = @{thm not-sym} RS @{thm Eq-FalseI};
fun is-neq eq lhs rhs thm =
(case Thm.prop-of thm of
- $ (Not $ (eq' $ l' $ r')) =>
Not = HOLogic.Not andalso eq' = eq andalso
r' aconv lhs andalso l' aconv rhs
| - => false);
fun proc ss ct =
(case Thm.term-of ct of
eq $ lhs $ rhs =>
(case find-first (is-neq eq lhs rhs) (Simplifier.prems-of-ss ss) of
SOME thm => SOME (thm RS neq-to-EQ-False)
| NONE => NONE)
| - => NONE);

```
in proc end;
⟩⟩

simproc-setup let-simp (Let x f) = ⟨⟨
let
  val (f-Let-unfold, x-Let-unfold) =
    let val [(- $ (f $ x) $ -)] = prems-of @{thm Let-unfold}
    in (cterm-of @{theory} f, cterm-of @{theory} x) end
  val (f-Let-folded, x-Let-folded) =
    let val [(- $ (f $ x) $ -)] = prems-of @{thm Let-folded}
    in (cterm-of @{theory} f, cterm-of @{theory} x) end;
  val g-Let-folded =
    let val [(- $ - $ (g $ -))] = prems-of @{thm Let-folded}
    in cterm-of @{theory} g end;
  fun count-loose (Bound i) k = if i >= k then 1 else 0
    | count-loose (s $ t) k = count-loose s k + count-loose t k
    | count-loose (Abs (-, -, t)) k = count-loose  t (k + 1)
    | count-loose - - = 0;
  fun is-trivial-let (Const (@{const-name Let}, -) $ x $ t) =
   case t
    of Abs (-, -, t') => count-loose t' 0 <= 1
     | - => true;
in fn - => fn ss => fn ct => if is-trivial-let (Thm.term-of ct)
  then SOME @{thm Let-def} (*no or one ocurrenc of bound variable*)
  else let (*Norbert Schirmer's case*)
    val ctxt = Simplifier.the-context ss;
    val thy = ProofContext.theory-of ctxt;
    val t = Thm.term-of ct;
    val ([t'], ctxt') = Variable.import-terms false [t] ctxt;
  in Option.map (hd o Variable.export ctxt' ctxt o single)
    (case t' of Const (@{const-name Let},-) $ x $ f => (* x and f are already in
normal form *)
      if is-Free x orelse is-Bound x orelse is-Const x
      then SOME @{thm Let-def}
      else
        let
          val n = case f of (Abs (x, -, -)) => x | - => x;
          val cx = cterm-of thy x;
          val {T = xT, ...} = rep-cterm cx;
          val cf = cterm-of thy f;
          val fx-g = Simplifier.rewrite ss (Thm.capply cf cx);
          val (- $ - $ g) = prop-of fx-g;
          val g' = abstract-over (x,g);
        in (if (g aconv g')
            then
              let
                val rl =
                    cterm-instantiate [(f-Let-unfold, cf), (x-Let-unfold, cx)] @{thm
Let-unfold};
```

```
            in SOME (rl OF [fx-g]) end
             else if Term.betapply (f, x) aconv g then NONE (*avoid identity
conversion*)
          else let
              val abs-g'= Abs (n,xT,g');
              val g'x = abs-g'$x;
              val g-g'x = symmetric (beta-conversion false (cterm-of thy g'x));
              val rl = cterm-instantiate
                      [(f-Let-folded, cterm-of thy f), (x-Let-folded, cx),
                       (g-Let-folded, cterm-of thy abs-g')]
                      @{thm Let-folded};
             in SOME (rl OF [transitive fx-g g-g'x])
             end)
      end
  | - => NONE)
 end
end ⟩⟩
```

lemma *True-implies-equals*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
proof
 assume $True \Longrightarrow PROP\ P$
 from *this* [*OF TrueI*] **show** $PROP\ P$ **.**
next
 assume $PROP\ P$
 then show $PROP\ P$ **.**
qed

lemma *ex-simps*:
 $!!P\ Q.\ (EX\ x.\ P\ x\ \&\ Q) = ((EX\ x.\ P\ x)\ \&\ Q)$
 $!!P\ Q.\ (EX\ x.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x.\ Q\ x))$
 $!!P\ Q.\ (EX\ x.\ P\ x\ |\ Q) = ((EX\ x.\ P\ x)\ |\ Q)$
 $!!P\ Q.\ (EX\ x.\ P\ |\ Q\ x) = (P\ |\ (EX\ x.\ Q\ x))$
 $!!P\ Q.\ (EX\ x.\ P\ x\ --{>}\ Q) = ((ALL\ x.\ P\ x)\ --{>}\ Q)$
 $!!P\ Q.\ (EX\ x.\ P\ --{>}\ Q\ x) = (P\ --{>}\ (EX\ x.\ Q\ x))$
 — Miniscoping: pushing in existential quantifiers.
 by (*iprover* | *blast*)+

lemma *all-simps*:
 $!!P\ Q.\ (ALL\ x.\ P\ x\ \&\ Q) = ((ALL\ x.\ P\ x)\ \&\ Q)$
 $!!P\ Q.\ (ALL\ x.\ P\ \&\ Q\ x) = (P\ \&\ (ALL\ x.\ Q\ x))$
 $!!P\ Q.\ (ALL\ x.\ P\ x\ |\ Q) = ((ALL\ x.\ P\ x)\ |\ Q)$
 $!!P\ Q.\ (ALL\ x.\ P\ |\ Q\ x) = (P\ |\ (ALL\ x.\ Q\ x))$
 $!!P\ Q.\ (ALL\ x.\ P\ x\ --{>}\ Q) = ((EX\ x.\ P\ x)\ --{>}\ Q)$
 $!!P\ Q.\ (ALL\ x.\ P\ --{>}\ Q\ x) = (P\ --{>}\ (ALL\ x.\ Q\ x))$
 — Miniscoping: pushing in universal quantifiers.
 by (*iprover* | *blast*)+

lemmas [*simp*] =
 triv-forall-equality

True-implies-equals
if-True
if-False
if-cancel
if-eq-cancel
imp-disjL

conj-assoc
disj-assoc
de-Morgan-conj
de-Morgan-disj
imp-disj1
imp-disj2
not-imp
disj-not1
not-all
not-ex
cases-simp
the-eq-trivial
the-sym-eq-trivial
ex-simps
all-simps
simp-thms

lemmas [*cong*] = *imp-cong simp-implies-cong*
lemmas [*split*] = *split-if*

ML ⟨⟨ *val HOL-ss* = @{*simpset*} ⟩⟩

Simplifies x assuming c and y assuming c

lemma *if-cong*:
assumes $b = c$
and $c \Longrightarrow x = u$
and $\neg\ c \Longrightarrow y = v$
shows (*if b then x else y*) = (*if c then u else v*)
unfolding *if-def* **using** *assms* **by** *simp*

Prevents simplification of x and y: faster and allows the execution of functional programs.

lemma *if-weak-cong* [*cong*]:
assumes $b = c$
shows (*if b then x else y*) = (*if c then x else y*)
using *assms* **by** (*rule arg-cong*)

Prevents simplification of t: much faster

lemma *let-weak-cong*:
assumes $a = b$
shows (*let x* = *a in t x*) = (*let x* = *b in t x*)
using *assms* **by** (*rule arg-cong*)

To tidy up the result of a simproc. Only the RHS will be simplified.

lemma *eq-cong2*:
assumes $u = u'$
shows $(t \equiv u) \equiv (t \equiv u')$
using *assms* **by** *simp*

lemma *if-distrib*:
$f\ (\mathit{if}\ c\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ c\ \mathit{then}\ f\ x\ \mathit{else}\ f\ y)$
by *simp*

This lemma restricts the effect of the rewrite rule u=v to the left-hand side of an equality. Used in {*Integ*,*Real*}/*simproc.ML*

lemma *restrict-to-left*:
assumes $x = y$
shows $(x = z) = (y = z)$
using *assms* **by** *simp*

1.3.3 Generic cases and induction

Rule projections:

ML ⟪
structure ProjectRule = ProjectRuleFun
(
val conjunct1 = @{thm conjunct1}
val conjunct2 = @{thm conjunct2}
val mp = @{thm mp}
)
⟫

constdefs
induct-forall **where** $\mathit{induct\text{-}forall}\ P == \forall x.\ P\ x$
induct-implies **where** $\mathit{induct\text{-}implies}\ A\ B == A \longrightarrow B$
induct-equal **where** $\mathit{induct\text{-}equal}\ x\ y == x = y$
induct-conj **where** $\mathit{induct\text{-}conj}\ A\ B == A \wedge B$

lemma *induct-forall-eq*: $(!!x.\ P\ x) == \mathit{Trueprop}\ (\mathit{induct\text{-}forall}\ (\lambda x.\ P\ x))$
by (*unfold atomize-all induct-forall-def*)

lemma *induct-implies-eq*: $(A ==> B) == \mathit{Trueprop}\ (\mathit{induct\text{-}implies}\ A\ B)$
by (*unfold atomize-imp induct-implies-def*)

lemma *induct-equal-eq*: $(x == y) == \mathit{Trueprop}\ (\mathit{induct\text{-}equal}\ x\ y)$
by (*unfold atomize-eq induct-equal-def*)

lemma *induct-conj-eq*: $(A\ \&\&\&\ B) == \mathit{Trueprop}\ (\mathit{induct\text{-}conj}\ A\ B)$
by (*unfold atomize-conj induct-conj-def*)

lemmas *induct-atomize = induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq*

lemmas *induct-rulify* [*symmetric*, *standard*] = *induct-atomize*
lemmas *induct-rulify-fallback* =
induct-forall-def induct-implies-def induct-equal-def induct-conj-def

lemma *induct-forall-conj*: *induct-forall* (λ*x*. *induct-conj* (*A x*) (*B x*)) =
induct-conj (*induct-forall A*) (*induct-forall B*)
by (*unfold induct-forall-def induct-conj-def*) *iprover*

lemma *induct-implies-conj*: *induct-implies C* (*induct-conj A B*) =
induct-conj (*induct-implies C A*) (*induct-implies C B*)
by (*unfold induct-implies-def induct-conj-def*) *iprover*

lemma *induct-conj-curry*: (*induct-conj A B* ==> *PROP C*) == (*A* ==> *B* ==>
PROP C)
proof
assume *r*: *induct-conj A B* ==> *PROP C* **and** *A B*
show *PROP C* **by** (*rule r*) (*simp add*: *induct-conj-def* ‹*A*› ‹*B*›)
next
assume *r*: *A* ==> *B* ==> *PROP C* **and** *induct-conj A B*
show *PROP C* **by** (*rule r*) (*simp-all add*: ‹*induct-conj A B*› [*unfolded induct-conj-def*])
qed

lemmas *induct-conj* = *induct-forall-conj induct-implies-conj induct-conj-curry*

hide *const induct-forall induct-implies induct-equal induct-conj*

Method setup.

ML ⟪
structure Induct = *InductFun*
(
val cases-default = @{*thm case-split*}
val atomize = @{*thms induct-atomize*}
val rulify = @{*thms induct-rulify*}
val rulify-fallback = @{*thms induct-rulify-fallback*}
)
⟫

setup *Induct.setup*

use ~~/*src*/*Tools*/*induct-tacs.ML*
setup *InductTacs.setup*

1.3.4 Coherent logic

ML ⟪
structure Coherent = *CoherentFun*
(
val atomize-elimL = @{*thm atomize-elimL*}

val atomize-exL = @{thm atomize-exL}
val atomize-conjL = @{thm atomize-conjL}
val atomize-disjL = @{thm atomize-disjL}
val operator-names =
[@{const-name op |}, @{const-name op &}, @{const-name Ex}]
);
⟩

setup *Coherent.setup*

1.4 Other simple lemmas and lemma duplicates

lemma *Let-0* [*simp*]: *Let 0 f = f 0*
unfolding *Let-def* **..**

lemma *Let-1* [*simp*]: *Let 1 f = f 1*
unfolding *Let-def* **..**

lemma *ex1-eq* [*iff*]: *EX! x. x = t EX! x. t = x*
by *blast+*

lemma *choice-eq*: *(ALL x. EX! y. P x y) = (EX! f. ALL x. P x (f x))*
apply (*rule iffI*)
apply (*rule-tac a = %x. THE y. P x y* **in** *ex1I*)
apply (*fast dest!*: *theI'*)
apply (*fast intro*: *ext the1-equality* [*symmetric*])
apply (*erule ex1E*)
apply (*rule allI*)
apply (*rule ex1I*)
apply (*erule spec*)
apply (*erule-tac x = %z. if z = x then y else f z* **in** *allE*)
apply (*erule impE*)
apply (*rule allI*)
apply (*case-tac xa = x*)
apply (*drule-tac* [*3*] *x = x* **in** *fun-cong*, *simp-all*)
done

lemma *mk-left-commute*:
fixes *f* (**infix** ⊗ *60*)
assumes *a*: ⋀*x y z. (x ⊗ y) ⊗ z = x ⊗ (y ⊗ z)* **and**
c: ⋀*x y. x ⊗ y = y ⊗ x*
shows *x ⊗ (y ⊗ z) = y ⊗ (x ⊗ z)*
by (*rule trans* [*OF trans* [*OF c a*] *arg-cong* [*OF c*, *of f y*]])

lemmas *eq-sym-conv = eq-commute*

lemma *nnf-simps*:
(¬(P ∧ Q)) = (¬ P ∨ ¬ Q) (¬ (P ∨ Q)) = (¬ P ∧ ¬Q) (P ⟶ Q) = (¬P ∨ Q)

$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q))$ $(\neg(P = Q)) = ((P \wedge \neg Q) \vee (\neg P \wedge Q))$
$(\neg \neg(P)) = P$
by *blast+*

1.5 Basic ML bindings

ML ⟪
val FalseE = @{thm FalseE}
val Let-def = @{thm Let-def}
val TrueI = @{thm TrueI}
val allE = @{thm allE}
val allI = @{thm allI}
val all-dupE = @{thm all-dupE}
val arg-cong = @{thm arg-cong}
val box-equals = @{thm box-equals}
val ccontr = @{thm ccontr}
val classical = @{thm classical}
val conjE = @{thm conjE}
val conjI = @{thm conjI}
val conjunct1 = @{thm conjunct1}
val conjunct2 = @{thm conjunct2}
val disjCI = @{thm disjCI}
val disjE = @{thm disjE}
val disjI1 = @{thm disjI1}
val disjI2 = @{thm disjI2}
val eq-reflection = @{thm eq-reflection}
val ex1E = @{thm ex1E}
val ex1I = @{thm ex1I}
val ex1-implies-ex = @{thm ex1-implies-ex}
val exE = @{thm exE}
val exI = @{thm exI}
val excluded-middle = @{thm excluded-middle}
val ext = @{thm ext}
val fun-cong = @{thm fun-cong}
val iffD1 = @{thm iffD1}
val iffD2 = @{thm iffD2}
val iffI = @{thm iffI}
val impE = @{thm impE}
val impI = @{thm impI}
val meta-eq-to-obj-eq = @{thm meta-eq-to-obj-eq}
val mp = @{thm mp}
val notE = @{thm notE}
val notI = @{thm notI}
val not-all = @{thm not-all}
val not-ex = @{thm not-ex}
val not-iff = @{thm not-iff}
val not-not = @{thm not-not}
val not-sym = @{thm not-sym}
val refl = @{thm refl}

val rev-mp = @{thm rev-mp}
val spec = @{thm spec}
val ssubst = @{thm ssubst}
val subst = @{thm subst}
val sym = @{thm sym}
val trans = @{thm trans}
⟩⟩

1.6 Code generator basics – see further theory *Code-Setup*

Equality

class *eq* =
  fixes *eq* :: *'a* ⇒ *'a* ⇒ *bool*
  assumes *eq-equals*: *eq x y* ⟷ *x* = *y*
begin

lemma *eq*: *eq* = (*op* =)
  by (*rule ext eq-equals*)+

lemma *eq-refl*: *eq x x* ⟷ *True*
  unfolding *eq* **by** *rule*+

end

Module setup

use *Tools/recfun-codegen.ML*

setup ⟨⟨
  Code-ML.setup
  #> *Code-Haskell.setup*
  #> *Nbe.setup*
  #> *Codegen.setup*
  #> *RecfunCodegen.setup*
⟩⟩

1.7 Nitpick hooks

This will be relocated once Nitpick is moved to HOL.

ML ⟨⟨
structure Nitpick-Const-Def-Thms = NamedThmsFun
(
  val name = nitpick-const-def
  val description = alternative definitions of constants as needed by Nitpick
)
structure Nitpick-Const-Simp-Thms = NamedThmsFun
(
  val name = nitpick-const-simp
  val description = equational specification of constants as needed by Nitpick

```
)
structure Nitpick-Const-Psimp-Thms = NamedThmsFun
(
  val name = nitpick-const-psimp
  val description = partial equational specification of constants as needed by Nitpick
)
structure Nitpick-Ind-Intro-Thms = NamedThmsFun
(
  val name = nitpick-ind-intro
   val description = introduction rules for (co)inductive predicates as needed by
Nitpick
)
⟩⟩
setup ⟨⟨ Nitpick-Const-Def-Thms.setup
        #> Nitpick-Const-Simp-Thms.setup
        #> Nitpick-Const-Psimp-Thms.setup
        #> Nitpick-Ind-Intro-Thms.setup ⟩⟩
```

1.8 Legacy tactics and ML bindings

```
ML ⟨⟨
fun strip-tac i = REPEAT (resolve-tac [impI, allI] i);

(* combination of (spec RS spec RS ...(j times) ... spec RS mp) *)
local
  fun wrong-prem (Const (All, -) $ (Abs (-, -, t))) = wrong-prem t
    | wrong-prem (Bound -) = true
    | wrong-prem - = false;
  val filter-right = filter (not o wrong-prem o HOLogic.dest-Trueprop o hd o Thm.prems-of);
in
  fun smp i = funpow i (fn m => filter-right ([spec] RL m)) ([mp]);
  fun smp-tac j = EVERY'[dresolve-tac (smp j), atac];
end;

val all-conj-distrib = thm all-conj-distrib;
val all-simps = thms all-simps;
val atomize-not = thm atomize-not;
val case-split = thm case-split;
val cases-simp = thm cases-simp;
val choice-eq = thm choice-eq
val cong = thm cong
val conj-comms = thms conj-comms;
val conj-cong = thm conj-cong;
val de-Morgan-conj = thm de-Morgan-conj;
val de-Morgan-disj = thm de-Morgan-disj;
val disj-assoc = thm disj-assoc;
val disj-comms = thms disj-comms;
val disj-cong = thm disj-cong;
val eq-ac = thms eq-ac;
```

val eq-cong2 = thm eq-cong2
val Eq-FalseI = thm Eq-FalseI;
val Eq-TrueI = thm Eq-TrueI;
val Ex1-def = thm Ex1-def
val ex-disj-distrib = thm ex-disj-distrib;
val ex-simps = thms ex-simps;
val if-cancel = thm if-cancel;
val if-eq-cancel = thm if-eq-cancel;
val if-False = thm if-False;
val iff-conv-conj-imp = thm iff-conv-conj-imp;
val iff = thm iff
val if-splits = thms if-splits;
val if-True = thm if-True;
val if-weak-cong = thm if-weak-cong
val imp-all = thm imp-all;
val imp-cong = thm imp-cong;
val imp-conjL = thm imp-conjL;
val imp-conjR = thm imp-conjR;
val imp-conv-disj = thm imp-conv-disj;
val simp-implies-def = thm simp-implies-def;
val simp-thms = thms simp-thms;
val split-if = thm split-if;
val the1-equality = thm the1-equality
val theI = thm theI
val theI' = thm theI'
val True-implies-equals = thm True-implies-equals;
val nnf-conv = Simplifier.rewrite (HOL-basic-ss addsimps simp-thms @ @{thms nnf-simps})

⟩⟩

end

2 Code-Setup: Setup of code generators and related tools

theory *Code-Setup*
imports *HOL*
begin

2.1 Generic code generator foundation

Datatypes

code-datatype *True False*

code-datatype *TYPE('a::{})*

code-datatype *Trueprop prop*

Code equations

lemma [*code*]:
  shows (*True* $\Longrightarrow$ *PROP P*) $\equiv$ *PROP P*
    and (*False* $\Longrightarrow$ *Q*) $\equiv$ *Trueprop True*
    and (*PROP P* $\Longrightarrow$ *True*) $\equiv$ *Trueprop True*
    and (*Q* $\Longrightarrow$ *False*) $\equiv$ *Trueprop* ($\neg$ *Q*) **by** (*auto intro*!: *equal-intr-rule*)

lemma [*code*]:
  shows *False* $\wedge$ *x* $\longleftrightarrow$ *False*
    and *True* $\wedge$ *x* $\longleftrightarrow$ *x*
    and *x* $\wedge$ *False* $\longleftrightarrow$ *False*
    and *x* $\wedge$ *True* $\longleftrightarrow$ *x* **by** *simp-all*

lemma [*code*]:
  shows *False* $\vee$ *x* $\longleftrightarrow$ *x*
    and *True* $\vee$ *x* $\longleftrightarrow$ *True*
    and *x* $\vee$ *False* $\longleftrightarrow$ *x*
    and *x* $\vee$ *True* $\longleftrightarrow$ *True* **by** *simp-all*

lemma [*code*]:
  shows $\neg$ *True* $\longleftrightarrow$ *False*
    and $\neg$ *False* $\longleftrightarrow$ *True* **by** (*rule HOL.simp-thms*)+

lemmas [*code*] = *Let-def if-True if-False*

lemmas [*code*, *code unfold*, *symmetric*, *code post*] = *imp-conv-disj*

Equality

context *eq*
begin

lemma *equals-eq* [*code inline*, *code*]: *op* = $\equiv$ *eq*
  by (*rule eq-reflection*) (*rule ext*, *rule ext*, *rule sym*, *rule eq-equals*)

declare *eq* [*code unfold*, *code inline del*]

declare *equals-eq* [*symmetric*, *code post*]

end

declare *simp-thms*(*6*) [*code nbe*]

hide (**open**) *const eq*
hide *const eq*

setup ⟪
  Code-Unit.add-const-alias @{*thm equals-eq*}

⟫

Cases

lemma *Let-case-cert*:
  assumes *CASE* ≡ (λ*x*. *Let x f*)
  shows *CASE x* ≡ *f x*
  using *assms* **by** *simp-all*

lemma *If-case-cert*:
  assumes *CASE* ≡ (λ*b*. *If b f g*)
  shows (*CASE True* ≡ *f*) &&& (*CASE False* ≡ *g*)
  using *assms* **by** *simp-all*

setup ⟪
  Code.add-case @{*thm Let-case-cert*}
  #> *Code.add-case* @{*thm If-case-cert*}
  #> *Code.add-undefined* @{*const-name undefined*}
⟫

code-abort *undefined*

2.2 Generic code generator preprocessor

setup ⟪
  Code.map-pre (*K HOL-basic-ss*)
  #> *Code.map-post* (*K HOL-basic-ss*)
⟫

2.3 Generic code generator target languages

type bool

code-type *bool*
  (*SML bool*)
  (*OCaml bool*)
  (*Haskell Bool*)

code-const *True* **and** *False* **and** *Not* **and** *op* & **and** *op* | **and** *If*
  (*SML true* **and** *false* **and** *not*
    and infixl *1 andalso* **and infixl** *0 orelse*
    and !(*if* (-)/ *then* (-)/ *else* (-)))
  (*OCaml true* **and** *false* **and** *not*
    and infixl *4* && **and infixl** *2* ||
    and !(*if* (-)/ *then* (-)/ *else* (-)))
  (*Haskell True* **and** *False* **and** *not*
    and infixl *3* && **and infixl** *2* ||
    and !(*if* (-)/ *then* (-)/ *else* (-)))

code-reserved *SML*
  bool true false not

code-reserved *OCaml*
bool not

using built-in Haskell equality

code-class *eq*
(*Haskell Eq*)

code-const *eq-class.eq*
(*Haskell* **infixl** *4 ==*)

code-const *op =*
(*Haskell* **infixl** *4 ==*)

undefined

code-const *undefined*
(*SML* !(*raise/ Fail/ undefined*))
(*OCaml failwith/ undefined*)
(*Haskell error/ undefined*)

2.4 SML code generator setup

types-code
bool (*bool*)
attach (*term-of*) ⟪
fun term-of-bool b = if b then HOLogic.true-const else HOLogic.false-const;
⟫
attach (*test*) ⟪
fun gen-bool i =
let val b = one-of [*false*, *true*]
in (*b*, *fn* () => *term-of-bool b*) *end;*
⟫
prop (*bool*)
attach (*term-of*) ⟪
fun term-of-prop b =
HOLogic.mk-Trueprop (*if b then HOLogic.true-const else HOLogic.false-const*);
⟫

consts-code
Trueprop ((-))
True (*true*)
False (*false*)
Not (*Bool.not*)
op | ((- *orelse/* -))
op & ((- *andalso/* -))
If ((*if* -/ *then* -/ *else* -))

setup ⟪
let

```
fun eq-codegen thy defs dep thyname b t gr =
    (case strip-comb t of
       (Const (op =, Type (-, [Type (fun, -), -])), -) => NONE
     | (Const (op =, -), [t, u]) =>
         let
           val (pt, gr') = Codegen.invoke-codegen thy defs dep thyname false t gr;
          val (pu, gr'') = Codegen.invoke-codegen thy defs dep thyname false u gr';
             val (-, gr''') = Codegen.invoke-tycodegen thy defs dep thyname false
HOLogic.boolT gr'';
         in
           SOME (Codegen.parens
             (Pretty.block [pt, Codegen.str  =, Pretty.brk 1, pu]), gr''')
         end
     | (t as Const (op =, -), ts) => SOME (Codegen.invoke-codegen
         thy defs dep thyname b (Codegen.eta-expand t ts 2) gr)
     | - => NONE);

in
  Codegen.add-codegen eq-codegen eq-codegen
end
⟩⟩
```

2.5 Evaluation and normalization by evaluation

```
setup ⟨⟨
  Value.add-evaluator (SML, Codegen.eval-term o ProofContext.theory-of)
⟩⟩

ML ⟨⟨
structure Eval-Method =
struct

val eval-ref : (unit -> bool) option ref = ref NONE;

end;
⟩⟩

oracle eval-oracle = ⟨⟨ fn ct =>
  let
    val thy = Thm.theory-of-cterm ct;
    val t = Thm.term-of ct;
    val dummy = @{cprop True};
  in case try HOLogic.dest-Trueprop t
   of SOME t' => if Code-ML.eval-term
        (Eval-Method.eval-ref, Eval-Method.eval-ref) thy t' []
       then Thm.capply (Thm.capply @{cterm op ≡ :: prop ⇒ prop ⇒ prop} ct)
dummy
       else dummy
```

```
    | NONE => dummy
  end
⟩⟩

ML ⟨⟨
fun gen-eval-method conv ctxt = SIMPLE-METHOD'
   (CONVERSION (Conv.params-conv (~1) (K (Conv.concl-conv (~1) conv)) ctxt)
    THEN' rtac TrueI)
⟩⟩

method-setup eval = ⟨⟨ Scan.succeed (gen-eval-method eval-oracle) ⟩⟩
  solve goal by evaluation

method-setup evaluation = ⟨⟨ Scan.succeed (gen-eval-method Codegen.evaluation-conv)
⟩⟩
  solve goal by evaluation

method-setup normalization = ⟨⟨
  Scan.succeed (K (SIMPLE-METHOD' (CONVERSION Nbe.norm-conv THEN'
(fn k => TRY (rtac TrueI k)))))
⟩⟩ solve goal by normalization
```

2.6 Quickcheck

```
setup ⟨⟨
  Quickcheck.add-generator (SML, Codegen.test-term)
⟩⟩

quickcheck-params [size = 5, iterations = 50]

end
```

3 Orderings: Abstract orderings

```
theory Orderings
imports Code-Setup
uses ~~/src/Provers/order.ML
begin
```

3.1 Quasi orders

```
class preorder = ord +
  assumes less-le-not-le: x < y ⟷ x ≤ y ∧ ¬ (y ≤ x)
  and order-refl [iff]: x ≤ x
  and order-trans: x ≤ y ⟹ y ≤ z ⟹ x ≤ z
begin
```

Reflexivity.

lemma *eq-refl*: $x = y \Longrightarrow x \leq y$
— This form is useful with the classical reasoner.
by (*erule ssubst*) (*rule order-refl*)

lemma *less-irrefl* [*iff*]: $\neg\ x < x$
by (*simp add*: *less-le-not-le*)

lemma *less-imp-le*: $x < y \Longrightarrow x \leq y$
unfolding *less-le-not-le* **by** *blast*

Asymmetry.

lemma *less-not-sym*: $x < y \Longrightarrow \neg\ (y < x)$
by (*simp add*: *less-le-not-le*)

lemma *less-asym*: $x < y \Longrightarrow (\neg\ P \Longrightarrow y < x) \Longrightarrow P$
by (*drule less-not-sym*, *erule contrapos-np*) *simp*

Transitivity.

lemma *less-trans*: $x < y \Longrightarrow y < z \Longrightarrow x < z$
by (*auto simp add*: *less-le-not-le intro*: *order-trans*)

lemma *le-less-trans*: $x \leq y \Longrightarrow y < z \Longrightarrow x < z$
by (*auto simp add*: *less-le-not-le intro*: *order-trans*)

lemma *less-le-trans*: $x < y \Longrightarrow y \leq z \Longrightarrow x < z$
by (*auto simp add*: *less-le-not-le intro*: *order-trans*)

Useful for simplification, but too risky to include by default.

lemma *less-imp-not-less*: $x < y \Longrightarrow (\neg\ y < x) \longleftrightarrow \mathit{True}$
by (*blast elim*: *less-asym*)

lemma *less-imp-triv*: $x < y \Longrightarrow (y < x \longrightarrow P) \longleftrightarrow \mathit{True}$
by (*blast elim*: *less-asym*)

Transitivity rules for calculational reasoning

lemma *less-asym'*: $a < b \Longrightarrow b < a \Longrightarrow P$
by (*rule less-asym*)

Dual order

lemma *dual-preorder*:
preorder (*op* $\geq$) (*op* $>$)
proof qed (*auto simp add*: *less-le-not-le intro*: *order-trans*)

end

3.2 Partial orders

class *order* = *preorder* +
  assumes *antisym*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
begin

Reflexivity.

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$
by (*auto simp add*: *less-le-not-le intro*: *antisym*)

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x = y$
  — NOT suitable for iff, since it can cause PROOF FAILED.
by (*simp add*: *less-le*) *blast*

lemma *le-imp-less-or-eq*: $x \leq y \Longrightarrow x < y \vee x = y$
unfolding *less-le* **by** *blast*

Useful for simplification, but too risky to include by default.

lemma *less-imp-not-eq*: $x < y \Longrightarrow (x = y) \longleftrightarrow \mathit{False}$
by *auto*

lemma *less-imp-not-eq2*: $x < y \Longrightarrow (y = x) \longleftrightarrow \mathit{False}$
by *auto*

Transitivity rules for calculational reasoning

lemma *neq-le-trans*: $a \neq b \Longrightarrow a \leq b \Longrightarrow a < b$
by (*simp add*: *less-le*)

lemma *le-neq-trans*: $a \leq b \Longrightarrow a \neq b \Longrightarrow a < b$
by (*simp add*: *less-le*)

Asymmetry.

lemma *eq-iff*: $x = y \longleftrightarrow x \leq y \wedge y \leq x$
by (*blast intro*: *antisym*)

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x = y$
by (*blast intro*: *antisym*)

lemma *less-imp-neq*: $x < y \Longrightarrow x \neq y$
by (*erule contrapos-pn*, *erule subst*, *rule less-irrefl*)

Least value operator

definition (**in** *ord*)
  Least :: $('a \Rightarrow \mathit{bool}) \Rightarrow 'a$ (**binder** *LEAST* 10) **where**
  Least $P = (\mathit{THE}\ x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow x \leq y))$

lemma *Least-equality*:
  assumes $P\ x$
    and $\bigwedge y.\ P\ y \Longrightarrow x \leq y$

shows *Least* $P = x$
unfolding *Least-def* **by** (*rule the-equality*)
(*blast intro*: *assms antisym*)+

lemma *LeastI2-order*:
assumes $P\ x$
and $\bigwedge y.\ P\ y \Longrightarrow x \leq y$
and $\bigwedge x.\ P\ x \Longrightarrow \forall\, y.\ P\ y \longrightarrow x \leq y \Longrightarrow Q\ x$
shows Q (*Least* P)
unfolding *Least-def* **by** (*rule theI2*)
(*blast intro*: *assms antisym*)+

Dual order

lemma *dual-order*:
order ($op \geq$) ($op >$)
by (*intro-locales*, *rule dual-preorder*) (*unfold-locales*, *rule antisym*)

end

3.3 Linear (total) orders

class *linorder* = *order* +
assumes *linear*: $x \leq y \vee y \leq x$
begin

lemma *less-linear*: $x < y \vee x = y \vee y < x$
unfolding *less-le* **using** *less-le linear* **by** *blast*

lemma *le-less-linear*: $x \leq y \vee y < x$
by (*simp add*: *le-less less-linear*)

lemma *le-cases* [*case-names le ge*]:
$(x \leq y \Longrightarrow P) \Longrightarrow (y \leq x \Longrightarrow P) \Longrightarrow P$
using *linear* **by** *blast*

lemma *linorder-cases* [*case-names less equal greater*]:
$(x < y \Longrightarrow P) \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow (y < x \Longrightarrow P) \Longrightarrow P$
using *less-linear* **by** *blast*

lemma *not-less*: $\neg\ x < y \longleftrightarrow y \leq x$
apply (*simp add*: *less-le*)
using *linear* **apply** (*blast intro*: *antisym*)
done

lemma *not-less-iff-gr-or-eq*:
$\neg(x < y) \longleftrightarrow (x > y \mid x = y)$
apply(*simp add*:*not-less le-less*)
apply *blast*
done

lemma *not-le*: $\neg\ x \leq y \longleftrightarrow y < x$
apply (*simp add*: *less-le*)
using *linear* **apply** (*blast intro*: *antisym*)
done

lemma *neq-iff*: $x \neq y \longleftrightarrow x < y \vee y < x$
by (*cut-tac* $x = x$ **and** $y = y$ **in** *less-linear*, *auto*)

lemma *neqE*: $x \neq y \Longrightarrow (x < y \Longrightarrow R) \Longrightarrow (y < x \Longrightarrow R) \Longrightarrow R$
by (*simp add*: *neq-iff*) *blast*

lemma *antisym-conv1*: $\neg\ x < y \Longrightarrow x \leq y \longleftrightarrow x = y$
by (*blast intro*: *antisym dest*: *not-less* [*THEN iffD1*])

lemma *antisym-conv2*: $x \leq y \Longrightarrow \neg\ x < y \longleftrightarrow x = y$
by (*blast intro*: *antisym dest*: *not-less* [*THEN iffD1*])

lemma *antisym-conv3*: $\neg\ y < x \Longrightarrow \neg\ x < y \longleftrightarrow x = y$
by (*blast intro*: *antisym dest*: *not-less* [*THEN iffD1*])

lemma *leI*: $\neg\ x < y \Longrightarrow y \leq x$
unfolding *not-less* **.**

lemma *leD*: $y \leq x \Longrightarrow \neg\ x < y$
unfolding *not-less* **.**

lemma *not-leE*: $\neg\ y \leq x \Longrightarrow x < y$
unfolding *not-le* **.**

Dual order

lemma *dual-linorder*:
linorder (*op* $\geq$) (*op* $>$)
by (*rule linorder.intro*, *rule dual-order*) (*unfold-locales*, *rule linear*)

min/max

definition (**in** *ord*) *min* :: $'a \Rightarrow 'a \Rightarrow 'a$ **where**
[*code del*]: *min a b* = (*if a* $\leq$ *b then a else b*)

definition (**in** *ord*) *max* :: $'a \Rightarrow 'a \Rightarrow 'a$ **where**
[*code del*]: *max a b* = (*if a* $\leq$ *b then b else a*)

lemma *min-le-iff-disj*:
min x y $\leq z \longleftrightarrow x \leq z \vee y \leq z$
unfolding *min-def* **using** *linear* **by** (*auto intro*: *order-trans*)

lemma *le-max-iff-disj*:
$z \leq$ *max x y* $\longleftrightarrow z \leq x \vee z \leq y$

unfolding *max-def* **using** *linear* **by** (*auto intro*: *order-trans*)

lemma *min-less-iff-disj*:
  $min\ x\ y < z \longleftrightarrow x < z \vee y < z$
unfolding *min-def le-less* **using** *less-linear* **by** (*auto intro*: *less-trans*)

lemma *less-max-iff-disj*:
  $z < max\ x\ y \longleftrightarrow z < x \vee z < y$
unfolding *max-def le-less* **using** *less-linear* **by** (*auto intro*: *less-trans*)

lemma *min-less-iff-conj* [*simp*]:
  $z < min\ x\ y \longleftrightarrow z < x \wedge z < y$
unfolding *min-def le-less* **using** *less-linear* **by** (*auto intro*: *less-trans*)

lemma *max-less-iff-conj* [*simp*]:
  $max\ x\ y < z \longleftrightarrow x < z \wedge y < z$
unfolding *max-def le-less* **using** *less-linear* **by** (*auto intro*: *less-trans*)

lemma *split-min* [*noatp*]:
  $P\ (min\ i\ j) \longleftrightarrow (i \leq j \longrightarrow P\ i) \wedge (\neg\ i \leq j \longrightarrow P\ j)$
by (*simp add*: *min-def*)

lemma *split-max* [*noatp*]:
  $P\ (max\ i\ j) \longleftrightarrow (i \leq j \longrightarrow P\ j) \wedge (\neg\ i \leq j \longrightarrow P\ i)$
by (*simp add*: *max-def*)

end

Explicit dictionaries for code generation

lemma *min-ord-min* [*code*, *code unfold*, *code inline del*]:
  $min = ord.min\ (op \leq)$
  by (*rule ext*)+ (*simp add*: *min-def ord.min-def*)

declare *ord.min-def* [*code*]

lemma *max-ord-max* [*code*, *code unfold*, *code inline del*]:
  $max = ord.max\ (op \leq)$
  by (*rule ext*)+ (*simp add*: *max-def ord.max-def*)

declare *ord.max-def* [*code*]

3.4 Reasoning tools setup

ML ⟪

```
signature ORDERS =
sig
  val print-structures: Proof.context -> unit
  val setup: theory -> theory
```

```
  val order-tac: thm list -> Proof.context -> int -> tactic
end;

structure Orders: ORDERS =
struct

(** Theory and context data **)

fun struct-eq ((s1: string, ts1), (s2, ts2)) =
  (s1 = s2) andalso eq-list (op aconv) (ts1, ts2);

structure Data = GenericDataFun
(
  type T = ((string * term list) * Order-Tac.less-arith) list;
    (* Order structures:
       identifier of the structure, list of operations and record of theorems
       needed to set up the transitivity reasoner,
       identifier and operations identify the structure uniquely. *)
  val empty = [];
  val extend = I;
  fun merge - = AList.join struct-eq (K fst);
);

fun print-structures ctxt =
  let
    val structs = Data.get (Context.Proof ctxt);
    fun pretty-term t = Pretty.block
      [Pretty.quote (Syntax.pretty-term ctxt t), Pretty.brk 1,
        Pretty.str ::, Pretty.brk 1,
        Pretty.quote (Syntax.pretty-typ ctxt (type-of t))];
    fun pretty-struct ((s, ts), -) = Pretty.block
      [Pretty.str s, Pretty.str :, Pretty.brk 1,
       Pretty.enclose ( ) (Pretty.breaks (map pretty-term ts))];
  in
    Pretty.writeln (Pretty.big-list Order structures: (map pretty-struct structs))
  end;


(** Method **)

fun struct-tac ((s, [eq, le, less]), thms) prems =
  let
    fun decomp thy (@{const Trueprop} $ t) =
      let
        fun excluded t =
          (* exclude numeric types: linear arithmetic subsumes transitivity *)
          let val T = type-of t
          in
    T = HOLogic.natT orelse T = HOLogic.intT orelse T = HOLogic.realT
```

```
        end;
fun rel (bin-op $ t1 $ t2) =
            if excluded t1 then NONE
            else if Pattern.matches thy (eq, bin-op) then SOME (t1, =, t2)
            else if Pattern.matches thy (le, bin-op) then SOME (t1, <=, t2)
            else if Pattern.matches thy (less, bin-op) then SOME (t1, <, t2)
            else NONE
  | rel - = NONE;
fun dec (Const (@{const-name Not}, -) $ t) = (case rel t
      of NONE => NONE
       | SOME (t1, rel, t2) => SOME (t1, ~ ^ rel, t2))
         | dec x = rel x;
      in dec t end
      | decomp thy - = NONE;
  in
    case s of
      order => Order-Tac.partial-tac decomp thms prems
    | linorder => Order-Tac.linear-tac decomp thms prems
     | - => error (Unknown kind of order ' ^ s ^ ' encountered in transitivity
reasoner.)
  end

fun order-tac prems ctxt =
  FIRST' (map (fn s => CHANGED o struct-tac s prems) (Data.get (Context.Proof
ctxt)));


(** Attribute **)

fun add-struct-thm s tag =
  Thm.declaration-attribute
   (fn thm => Data.map (AList.map-default struct-eq (s, Order-Tac.empty TrueI)
(Order-Tac.update tag thm)));
fun del-struct s =
  Thm.declaration-attribute
    (fn - => Data.map (AList.delete struct-eq s));

val attrib-setup =
  Attrib.setup @{binding order}
    (Scan.lift ((Args.add -- Args.name >> (fn (-, s) => SOME s) || Args.del
>> K NONE) --|
      Args.colon (* FIXME || Scan.succeed true *) ) -- Scan.lift Args.name --
      Scan.repeat Args.term
      >> (fn ((SOME tag, n), ts) => add-struct-thm (n, ts) tag
           | ((NONE, n), ts) => del-struct (n, ts)))
    theorems controlling transitivity reasoner;


(** Diagnostic command **)
```

```
val - =
  OuterSyntax.improper-command print-orders
    print order structures available to transitivity reasoner OuterKeyword.diag
    (Scan.succeed (Toplevel.no-timing o Toplevel.unknown-context o
        Toplevel.keep (print-structures o Toplevel.context-of)));


(** Setup **)

val setup =
  Method.setup @{binding order} (Scan.succeed (SIMPLE-METHOD' o order-tac
[]))
    transitivity reasoner #>
  attrib-setup;

end;

⟫
```

setup *Orders.setup*

Declarations to set up transitivity reasoner of partial and linear orders.

context *order*
begin

declare *less-irrefl* [*THEN notE*, *order add less-reflE*: *order op* = :: *'a* ⇒ *'a* ⇒ *bool op* <= *op* <]

declare *order-refl* [*order add le-refl*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *less-imp-le* [*order add less-imp-le*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *antisym* [*order add eqI*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *eq-refl* [*order add eqD1*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *sym* [*THEN eq-refl*, *order add eqD2*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *less-trans* [*order add less-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *less-le-trans* [*order add less-le-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *le-less-trans* [*order add le-less-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *order-trans* [*order add le-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *le-neq-trans* [*order add le-neq-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *neq-le-trans* [*order add neq-le-trans*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *less-imp-neq* [*order add less-imp-neq*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *eq-neq-eq-imp-neq* [*order add eq-neq-eq-imp-neq*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *not-sym* [*order add not-sym*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]

end

context *linorder*
begin

declare [[*order del*: *order op* = :: *'a* => *'a* => *bool op* <= *op* <]]

declare *less-irrefl* [*THEN notE*, *order add less-reflE*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *order-refl* [*order add le-refl*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *less-imp-le* [*order add less-imp-le*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *not-less* [*THEN iffD2*, *order add not-lessI*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *not-le* [*THEN iffD2*, *order add not-leI*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *not-less* [*THEN iffD1*, *order add not-lessD*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *not-le* [*THEN iffD1*, *order add not-leD*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *antisym* [*order add eqI*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *eq-refl* [*order add eqD1*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *sym* [*THEN eq-refl*, *order add eqD2*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *less-trans* [*order add less-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *less-le-trans* [*order add less-le-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *le-less-trans* [*order add le-less-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *order-trans* [*order add le-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *le-neq-trans* [*order add le-neq-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *neq-le-trans* [*order add neq-le-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *less-imp-neq* [*order add less-imp-neq*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *eq-neq-eq-imp-neq* [*order add eq-neq-eq-imp-neq*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

declare *not-sym* [*order add not-sym*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <]

end

setup ⟪
let

fun prp t thm = (#*prop* (*rep-thm thm*) = *t*);

fun prove-antisym-le sg ss ((*le as Const*(-, *T*)) $ *r* $ *s*) =
  let val prems = *prems-of-ss ss*;
      val less = *Const* (@{*const-name less*}, *T*);
      val t = *HOLogic.mk-Trueprop*(*le* $ *s* $ *r*);
  in case find-first (*prp t*) *prems of*
       NONE =>

```
      let val t = HOLogic.mk-Trueprop(HOLogic.Not $ (less $ r $ s))
      in case find-first (prp t) prems of
          NONE => NONE
        | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv1}))
      end
  | SOME thm => SOME(mk-meta-eq(thm RS @{thm order-class.antisym-conv}))
 end
 handle THM - => NONE;

fun prove-antisym-less sg ss (NotC $ ((less as Const(-,T)) $ r $ s)) =
 let val prems = prems-of-ss ss;
     val le = Const (@{const-name less-eq}, T);
     val t = HOLogic.mk-Trueprop(le $ r $ s);
 in case find-first (prp t) prems of
      NONE =>
      let val t = HOLogic.mk-Trueprop(NotC $ (less $ s $ r))
      in case find-first (prp t) prems of
          NONE => NONE
        | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv3}))
      end
  | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv2}))
 end
 handle THM - => NONE;

fun add-simprocs procs thy =
 Simplifier.map-simpset (fn ss => ss
   addsimprocs (map (fn (name, raw-ts, proc) =>
     Simplifier.simproc thy name raw-ts proc) procs)) thy;
fun add-solver name tac =
 Simplifier.map-simpset (fn ss => ss addSolver
  mk-solver' name (fn ss => tac (Simplifier.prems-of-ss ss) (Simplifier.the-context
ss)));

in
 add-simprocs [
     (antisym le, [(x::'a::order) <= y], prove-antisym-le),
     (antisym less, [~ (x::'a::linorder) < y], prove-antisym-less)
   ]
 #> add-solver Transitivity Orders.order-tac
 (* Adding the transitivity reasoners also as safe solvers showed a slight
    speed up, but the reasoning strength appears to be not higher (at least
    no breaking of additional proofs in the entire HOL distribution, as
    of 5 March 2004, was observed). *)
end
⟩
```

3.5 Name duplicates

lemmas *order-less-le = less-le*

lemmas *order-eq-refl = preorder-class.eq-refl*
lemmas *order-less-irrefl = preorder-class.less-irrefl*
lemmas *order-le-less = order-class.le-less*
lemmas *order-le-imp-less-or-eq = order-class.le-imp-less-or-eq*
lemmas *order-less-imp-le = preorder-class.less-imp-le*
lemmas *order-less-imp-not-eq = order-class.less-imp-not-eq*
lemmas *order-less-imp-not-eq2 = order-class.less-imp-not-eq2*
lemmas *order-neq-le-trans = order-class.neq-le-trans*
lemmas *order-le-neq-trans = order-class.le-neq-trans*

lemmas *order-antisym = antisym*
lemmas *order-less-not-sym = preorder-class.less-not-sym*
lemmas *order-less-asym = preorder-class.less-asym*
lemmas *order-eq-iff = order-class.eq-iff*
lemmas *order-antisym-conv = order-class.antisym-conv*
lemmas *order-less-trans = preorder-class.less-trans*
lemmas *order-le-less-trans = preorder-class.le-less-trans*
lemmas *order-less-le-trans = preorder-class.less-le-trans*
lemmas *order-less-imp-not-less = preorder-class.less-imp-not-less*
lemmas *order-less-imp-triv = preorder-class.less-imp-triv*
lemmas *order-less-asym' = preorder-class.less-asym'*

lemmas *linorder-linear = linear*
lemmas *linorder-less-linear = linorder-class.less-linear*
lemmas *linorder-le-less-linear = linorder-class.le-less-linear*
lemmas *linorder-le-cases = linorder-class.le-cases*
lemmas *linorder-not-less = linorder-class.not-less*
lemmas *linorder-not-le = linorder-class.not-le*
lemmas *linorder-neq-iff = linorder-class.neq-iff*
lemmas *linorder-neqE = linorder-class.neqE*
lemmas *linorder-antisym-conv1 = linorder-class.antisym-conv1*
lemmas *linorder-antisym-conv2 = linorder-class.antisym-conv2*
lemmas *linorder-antisym-conv3 = linorder-class.antisym-conv3*

3.6 Bounded quantifiers

syntax
-All-less :: [*idt*, *'a*, *bool*] => *bool* ((*3ALL -<-./ -*) [*0*, *0*, *10*] *10*)
-Ex-less :: [*idt*, *'a*, *bool*] => *bool* ((*3EX -<-./ -*) [*0*, *0*, *10*] *10*)
-All-less-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3ALL -<=-./ -*) [*0*, *0*, *10*] *10*)
-Ex-less-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3EX -<=-./ -*) [*0*, *0*, *10*] *10*)

-All-greater :: [*idt*, *'a*, *bool*] => *bool* ((*3ALL ->-./ -*) [*0*, *0*, *10*] *10*)
-Ex-greater :: [*idt*, *'a*, *bool*] => *bool* ((*3EX ->-./ -*) [*0*, *0*, *10*] *10*)
-All-greater-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3ALL ->=-./ -*) [*0*, *0*, *10*] *10*)
-Ex-greater-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3EX ->=-./ -*) [*0*, *0*, *10*] *10*)

syntax (*xsymbols*)
-All-less :: [*idt*, *'a*, *bool*] => *bool* ((*3∀ -<-./ -*) [*0*, *0*, *10*] *10*)

-Ex-less :: [*idt*, *'a*, *bool*] => *bool* ((*3*∃ -<-./ -) [*0*, *0*, *10*] *10*)
-All-less-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*∀ -≤-./ -) [*0*, *0*, *10*] *10*)
-Ex-less-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*∃ -≤-./ -) [*0*, *0*, *10*] *10*)

-All-greater :: [*idt*, *'a*, *bool*] => *bool* ((*3*∀ ->-./ -) [*0*, *0*, *10*] *10*)
-Ex-greater :: [*idt*, *'a*, *bool*] => *bool* ((*3*∃ ->-./ -) [*0*, *0*, *10*] *10*)
-All-greater-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*∀ -≥-./ -) [*0*, *0*, *10*] *10*)
-Ex-greater-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*∃ -≥-./ -) [*0*, *0*, *10*] *10*)

syntax (*HOL*)
-All-less :: [*idt*, *'a*, *bool*] => *bool* ((*3*! -<-./ -) [*0*, *0*, *10*] *10*)
-Ex-less :: [*idt*, *'a*, *bool*] => *bool* ((*3?* -<-./ -) [*0*, *0*, *10*] *10*)
-All-less-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*! -<=-./ -) [*0*, *0*, *10*] *10*)
-Ex-less-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3?* -<=-./ -) [*0*, *0*, *10*] *10*)

syntax (*HTML* **output**)
-All-less :: [*idt*, *'a*, *bool*] => *bool* ((*3*∀ -<-./ -) [*0*, *0*, *10*] *10*)
-Ex-less :: [*idt*, *'a*, *bool*] => *bool* ((*3*∃ -<-./ -) [*0*, *0*, *10*] *10*)
-All-less-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*∀ -≤-./ -) [*0*, *0*, *10*] *10*)
-Ex-less-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*∃ -≤-./ -) [*0*, *0*, *10*] *10*)

-All-greater :: [*idt*, *'a*, *bool*] => *bool* ((*3*∀ ->-./ -) [*0*, *0*, *10*] *10*)
-Ex-greater :: [*idt*, *'a*, *bool*] => *bool* ((*3*∃ ->-./ -) [*0*, *0*, *10*] *10*)
-All-greater-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*∀ -≥-./ -) [*0*, *0*, *10*] *10*)
-Ex-greater-eq :: [*idt*, *'a*, *bool*] => *bool* ((*3*∃ -≥-./ -) [*0*, *0*, *10*] *10*)

translations
ALL x<y. P => *ALL x. x < y ⟶ P*
EX x<y. P => *EX x. x < y ∧ P*
ALL x<=y. P => *ALL x. x <= y ⟶ P*
EX x<=y. P => *EX x. x <= y ∧ P*
ALL x>y. P => *ALL x. x > y ⟶ P*
EX x>y. P => *EX x. x > y ∧ P*
ALL x>=y. P => *ALL x. x >= y ⟶ P*
EX x>=y. P => *EX x. x >= y ∧ P*

print-translation ⟪
let
val All-binder = Syntax.binder-name @{*const-syntax All*};
val Ex-binder = Syntax.binder-name @{*const-syntax Ex*};
val impl = @{*const-syntax op* −−>};
val conj = @{*const-syntax op* &};
val less = @{*const-syntax less*};
val less-eq = @{*const-syntax less-eq*};

val trans =
[((*All-binder*, *impl*, *less*), (*-All-less*, *-All-greater*)),
((*All-binder*, *impl*, *less-eq*), (*-All-less-eq*, *-All-greater-eq*)),
((*Ex-binder*, *conj*, *less*), (*-Ex-less*, *-Ex-greater*)),

```
    ((Ex-binder, conj, less-eq), (-Ex-less-eq, -Ex-greater-eq))];

  fun matches-bound v t =
    case t of (Const (-bound, -) $ Free (v', -)) => (v = v')
            | - => false
  fun contains-var v = Term.exists-subterm (fn Free (x, -) => x = v | - => false)
  fun mk v c n P = Syntax.const c $ Syntax.mark-bound v $ n $ P

  fun tr' q = (q,
    fn [Const (-bound, -) $ Free (v, -), Const (c, -) $ (Const (d, -) $ t $ u) $ P]
=>
      (case AList.lookup (op =) trans (q, c, d) of
        NONE => raise Match
      | SOME (l, g) =>
          if matches-bound v t andalso not (contains-var v u) then mk v l u P
          else if matches-bound v u andalso not (contains-var v t) then mk v g t P
          else raise Match)
    | - => raise Match);
in [tr' All-binder, tr' Ex-binder] end
⟫
```

3.7 Transitivity reasoning

context *ord*
begin

lemma *ord-le-eq-trans*: $a \leq b \Longrightarrow b = c \Longrightarrow a \leq c$
  by (*rule subst*)

lemma *ord-eq-le-trans*: $a = b \Longrightarrow b \leq c \Longrightarrow a \leq c$
  by (*rule ssubst*)

lemma *ord-less-eq-trans*: $a < b \Longrightarrow b = c \Longrightarrow a < c$
  by (*rule subst*)

lemma *ord-eq-less-trans*: $a = b \Longrightarrow b < c \Longrightarrow a < c$
  by (*rule ssubst*)

end

lemma *order-less-subst2*: $(a::'a::order) < b \Longrightarrow f\ b < (c::'c::order) \Longrightarrow$
  $(!!x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow f\ a < c$
proof −
  assume *r*: $!!x\ y.\ x < y \Longrightarrow f\ x < f\ y$
  assume $a < b$ **hence** $f\ a < f\ b$ **by** (*rule r*)
  also assume $f\ b < c$
  finally (*order-less-trans*) **show** *?thesis* **.**
qed

lemma *order-less-subst1*: $(a::'a::order) < f\ b \Longrightarrow (b::'b::order) < c \Longrightarrow$
$(!!x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
proof −
assume *r*: $!!x\ y.\ x < y \Longrightarrow f\ x < f\ y$
assume $a < f\ b$
also assume $b < c$ **hence** $f\ b < f\ c$ **by** (*rule r*)
finally (*order-less-trans*) **show** *?thesis* **.**
qed

lemma *order-le-less-subst2*: $(a::'a::order) <= b \Longrightarrow f\ b < (c::'c::order) \Longrightarrow$
$(!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y) \Longrightarrow f\ a < c$
proof −
assume *r*: $!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y$
assume $a <= b$ **hence** $f\ a <= f\ b$ **by** (*rule r*)
also assume $f\ b < c$
finally (*order-le-less-trans*) **show** *?thesis* **.**
qed

lemma *order-le-less-subst1*: $(a::'a::order) <= f\ b \Longrightarrow (b::'b::order) < c \Longrightarrow$
$(!!x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
proof −
assume *r*: $!!x\ y.\ x < y \Longrightarrow f\ x < f\ y$
assume $a <= f\ b$
also assume $b < c$ **hence** $f\ b < f\ c$ **by** (*rule r*)
finally (*order-le-less-trans*) **show** *?thesis* **.**
qed

lemma *order-less-le-subst2*: $(a::'a::order) < b \Longrightarrow f\ b <= (c::'c::order) \Longrightarrow$
$(!!x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow f\ a < c$
proof −
assume *r*: $!!x\ y.\ x < y \Longrightarrow f\ x < f\ y$
assume $a < b$ **hence** $f\ a < f\ b$ **by** (*rule r*)
also assume $f\ b <= c$
finally (*order-less-le-trans*) **show** *?thesis* **.**
qed

lemma *order-less-le-subst1*: $(a::'a::order) < f\ b \Longrightarrow (b::'b::order) <= c \Longrightarrow$
$(!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y) \Longrightarrow a < f\ c$
proof −
assume *r*: $!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y$
assume $a < f\ b$
also assume $b <= c$ **hence** $f\ b <= f\ c$ **by** (*rule r*)
finally (*order-less-le-trans*) **show** *?thesis* **.**
qed

lemma *order-subst1*: $(a::'a::order) <= f\ b \Longrightarrow (b::'b::order) <= c \Longrightarrow$
$(!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y) \Longrightarrow a <= f\ c$
proof −
assume *r*: $!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y$

assume $a <= f\ b$
also assume $b <= c$ **hence** $f\ b <= f\ c$ **by** (*rule* r)
finally (*order-trans*) **show** *?thesis* **.**
qed

lemma *order-subst2*: $(a::'a::order) <= b ==> f\ b <= (c::'c::order) ==>$
$(!!x\ y.\ x <= y ==> f\ x <= f\ y) ==> f\ a <= c$
proof −
assume r: $!!x\ y.\ x <= y ==> f\ x <= f\ y$
assume $a <= b$ **hence** $f\ a <= f\ b$ **by** (*rule* r)
also assume $f\ b <= c$
finally (*order-trans*) **show** *?thesis* **.**
qed

lemma *ord-le-eq-subst*: $a <= b ==> f\ b = c ==>$
$(!!x\ y.\ x <= y ==> f\ x <= f\ y) ==> f\ a <= c$
proof −
assume r: $!!x\ y.\ x <= y ==> f\ x <= f\ y$
assume $a <= b$ **hence** $f\ a <= f\ b$ **by** (*rule* r)
also assume $f\ b = c$
finally (*ord-le-eq-trans*) **show** *?thesis* **.**
qed

lemma *ord-eq-le-subst*: $a = f\ b ==> b <= c ==>$
$(!!x\ y.\ x <= y ==> f\ x <= f\ y) ==> a <= f\ c$
proof −
assume r: $!!x\ y.\ x <= y ==> f\ x <= f\ y$
assume $a = f\ b$
also assume $b <= c$ **hence** $f\ b <= f\ c$ **by** (*rule* r)
finally (*ord-eq-le-trans*) **show** *?thesis* **.**
qed

lemma *ord-less-eq-subst*: $a < b ==> f\ b = c ==>$
$(!!x\ y.\ x < y ==> f\ x < f\ y) ==> f\ a < c$
proof −
assume r: $!!x\ y.\ x < y ==> f\ x < f\ y$
assume $a < b$ **hence** $f\ a < f\ b$ **by** (*rule* r)
also assume $f\ b = c$
finally (*ord-less-eq-trans*) **show** *?thesis* **.**
qed

lemma *ord-eq-less-subst*: $a = f\ b ==> b < c ==>$
$(!!x\ y.\ x < y ==> f\ x < f\ y) ==> a < f\ c$
proof −
assume r: $!!x\ y.\ x < y ==> f\ x < f\ y$
assume $a = f\ b$
also assume $b < c$ **hence** $f\ b < f\ c$ **by** (*rule* r)
finally (*ord-eq-less-trans*) **show** *?thesis* **.**
qed

Note that this list of rules is in reverse order of priorities.

lemmas [*trans*] =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst
ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp

lemmas (**in** *order*) [*trans*] =
neq-le-trans
le-neq-trans

lemmas (**in** *preorder*) [*trans*] =
less-trans
less-asym'
le-less-trans
less-le-trans
order-trans

lemmas (**in** *order*) [*trans*] =
antisym

lemmas (**in** *ord*) [*trans*] =
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans

lemmas [*trans*] =
trans

lemmas *order-trans-rules* =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1

order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst
ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp
neq-le-trans
le-neq-trans
less-trans
less-asym'
le-less-trans
less-le-trans
order-trans
antisym
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans
trans

These support proving chains of decreasing inequalities a ¿= b ¿= c ... in Isar proofs.

lemma *xt1*:
$a = b \Longrightarrow b > c \Longrightarrow a > c$
$a > b \Longrightarrow b = c \Longrightarrow a > c$
$a = b \Longrightarrow b >= c \Longrightarrow a >= c$
$a >= b \Longrightarrow b = c \Longrightarrow a >= c$
$(x::'a::order) >= y \Longrightarrow y >= x \Longrightarrow x = y$
$(x::'a::order) >= y \Longrightarrow y >= z \Longrightarrow x >= z$
$(x::'a::order) > y \Longrightarrow y >= z \Longrightarrow x > z$
$(x::'a::order) >= y \Longrightarrow y > z \Longrightarrow x > z$
$(a::'a::order) > b \Longrightarrow b > a \Longrightarrow P$
$(x::'a::order) > y \Longrightarrow y > z \Longrightarrow x > z$
$(a::'a::order) >= b \Longrightarrow a \sim= b \Longrightarrow a > b$
$(a::'a::order) \sim= b \Longrightarrow a >= b \Longrightarrow a > b$
$a = f\ b \Longrightarrow b > c \Longrightarrow (!!x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow a > f\ c$
$a > b \Longrightarrow f\ b = c \Longrightarrow (!!x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow f\ a > c$
$a = f\ b \Longrightarrow b >= c \Longrightarrow (!!x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow a >= f\ c$
$a >= b \Longrightarrow f\ b = c \Longrightarrow (!!\ x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow f\ a >= c$
by *auto*

lemma *xt2*:
$(a::'a::order) >= f\ b \Longrightarrow b >= c \Longrightarrow (!!x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow a >= f\ c$
by (*subgoal-tac* $f\ b >= f\ c$, *force*, *force*)

lemma *xt3*: $(a::'a::order) >= b ==> (f\ b::'b::order) >= c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> f\ a >= c$
by (*subgoal-tac* $f\ a >= f\ b$, *force*, *force*)

lemma *xt4*: $(a::'a::order) > f\ b ==> (b::'b::order) >= c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> a > f\ c$
by (*subgoal-tac* $f\ b >= f\ c$, *force*, *force*)

lemma *xt5*: $(a::'a::order) > b ==> (f\ b::'b::order) >= c==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> f\ a > c$
by (*subgoal-tac* $f\ a > f\ b$, *force*, *force*)

lemma *xt6*: $(a::'a::order) >= f\ b ==> b > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> a > f\ c$
by (*subgoal-tac* $f\ b > f\ c$, *force*, *force*)

lemma *xt7*: $(a::'a::order) >= b ==> (f\ b::'b::order) > c ==>$
$(!!x\ y.\ x >= y ==> f\ x >= f\ y) ==> f\ a > c$
by (*subgoal-tac* $f\ a >= f\ b$, *force*, *force*)

lemma *xt8*: $(a::'a::order) > f\ b ==> (b::'b::order) > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> a > f\ c$
by (*subgoal-tac* $f\ b > f\ c$, *force*, *force*)

lemma *xt9*: $(a::'a::order) > b ==> (f\ b::'b::order) > c ==>$
$(!!x\ y.\ x > y ==> f\ x > f\ y) ==> f\ a > c$
by (*subgoal-tac* $f\ a > f\ b$, *force*, *force*)

lemmas *xtrans* = *xt1 xt2 xt3 xt4 xt5 xt6 xt7 xt8 xt9*

3.8 Monotonicity, least value operator and min/max

context *order*
begin

definition *mono* :: $('a \Rightarrow 'b::order) \Rightarrow bool$ **where**
$mono\ f \longleftrightarrow (\forall x\ y.\ x \leq y \longrightarrow f\ x \leq f\ y)$

lemma *monoI* [*intro?*]:
fixes $f :: 'a \Rightarrow 'b::order$
shows $(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow mono\ f$
unfolding *mono-def* **by** *iprover*

lemma *monoD* [*dest?*]:
fixes $f :: 'a \Rightarrow 'b::order$
shows $mono\ f \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
unfolding *mono-def* **by** *iprover*

definition *strict-mono* :: $('a \Rightarrow 'b{::}order) \Rightarrow bool$ **where**
  $strict\text{-}mono\ f \longleftrightarrow (\forall x\ y.\ x < y \longrightarrow f\ x < f\ y)$

lemma *strict-monoI* [*intro?*]:
  assumes $\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y$
  shows *strict-mono f*
  using *assms* **unfolding** *strict-mono-def* **by** *auto*

lemma *strict-monoD* [*dest?*]:
  $strict\text{-}mono\ f \Longrightarrow x < y \Longrightarrow f\ x < f\ y$
  unfolding *strict-mono-def* **by** *auto*

lemma *strict-mono-mono* [*dest?*]:
  assumes *strict-mono f*
  shows *mono f*
proof (*rule monoI*)
  fix *x y*
  assume $x \leq y$
  show $f\ x \leq f\ y$
  proof (*cases* $x = y$)
    case *True* **then show** *?thesis* **by** *simp*
  next
    case *False* **with** ‹$x \leq y$› **have** $x < y$ **by** *simp*
    with *assms strict-monoD* **have** $f\ x < f\ y$ **by** *auto*
    then show *?thesis* **by** *simp*
  qed
qed

end

context *linorder*
begin

lemma *strict-mono-eq*:
  assumes *strict-mono f*
  shows $f\ x = f\ y \longleftrightarrow x = y$
proof
  assume $f\ x = f\ y$
  show $x = y$ **proof** (*cases x y rule*: *linorder-cases*)
    case *less* **with** *assms strict-monoD* **have** $f\ x < f\ y$ **by** *auto*
    with ‹$f\ x = f\ y$› **show** *?thesis* **by** *simp*
  next
    case *equal* **then show** *?thesis* **.**
  next
    case *greater* **with** *assms strict-monoD* **have** $f\ y < f\ x$ **by** *auto*
    with ‹$f\ x = f\ y$› **show** *?thesis* **by** *simp*
  qed
qed *simp*

lemma *strict-mono-less-eq*:
assumes *strict-mono f*
shows $f\ x \leq f\ y \longleftrightarrow x \leq y$
proof
assume $x \leq y$
with *assms strict-mono-mono monoD* **show** $f\ x \leq f\ y$ **by** *auto*
next
assume $f\ x \leq f\ y$
show $x \leq y$ **proof** (*rule ccontr*)
assume $\neg\ x \leq y$ **then have** $y < x$ **by** *simp*
with *assms strict-monoD* **have** $f\ y < f\ x$ **by** *auto*
with ‹$f\ x \leq f\ y$› **show** *False* **by** *simp*
qed
qed

lemma *strict-mono-less*:
assumes *strict-mono f*
shows $f\ x < f\ y \longleftrightarrow x < y$
using *assms*
by (*auto simp add*: *less-le Orderings.less-le strict-mono-eq strict-mono-less-eq*)

lemma *min-of-mono*:
fixes $f :: 'a \Rightarrow 'b$::*linorder*
shows *mono f* $\Longrightarrow$ *min* $(f\ m)\ (f\ n) = f\ (min\ m\ n)$
by (*auto simp*: *mono-def Orderings.min-def min-def intro*: *Orderings.antisym*)

lemma *max-of-mono*:
fixes $f :: 'a \Rightarrow 'b$::*linorder*
shows *mono f* $\Longrightarrow$ *max* $(f\ m)\ (f\ n) = f\ (max\ m\ n)$
by (*auto simp*: *mono-def Orderings.max-def max-def intro*: *Orderings.antisym*)

end

lemma *min-leastL*: $(!!x.\ least <= x) ==> min\ least\ x = least$
by (*simp add*: *min-def*)

lemma *max-leastL*: $(!!x.\ least <= x) ==> max\ least\ x = x$
by (*simp add*: *max-def*)

lemma *min-leastR*: $(\bigwedge x::'a::order.\ least \leq x) \Longrightarrow min\ x\ least = least$
apply (*simp add*: *min-def*)
apply (*blast intro*: *order-antisym*)
done

lemma *max-leastR*: $(\bigwedge x::'a::order.\ least \leq x) \Longrightarrow max\ x\ least = x$
apply (*simp add*: *max-def*)
apply (*blast intro*: *order-antisym*)
done

3.9 Top and bottom elements

class *top* = *preorder* +
  fixes *top* :: *'a*
  assumes *top-greatest* [*simp*]: $x \leq top$

class *bot* = *preorder* +
  fixes *bot* :: *'a*
  assumes *bot-least* [*simp*]: $bot \leq x$

3.10 Dense orders

class *dense-linear-order* = *linorder* +
  assumes *gt-ex*: $\exists\, y.\ x < y$
  and *lt-ex*: $\exists\, y.\ y < x$
  and *dense*: $x < y \Longrightarrow (\exists\, z.\ x < z \wedge z < y)$

3.11 Wellorders

class *wellorder* = *linorder* +
  assumes *less-induct* [*case-names less*]: $(\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$
begin

lemma *wellorder-Least-lemma*:
  fixes k :: *'a*
  assumes $P\ k$
  shows $P\ (LEAST\ x.\ P\ x)$ **and** $(LEAST\ x.\ P\ x) \leq k$
proof −
  have $P\ (LEAST\ x.\ P\ x) \wedge (LEAST\ x.\ P\ x) \leq k$
  using *assms* **proof** (*induct k rule*: *less-induct*)
    case (*less x*) **then have** $P\ x$ **by** *simp*
    show *?case* **proof** (*rule classical*)
      assume *assm*: $\neg\ (P\ (LEAST\ a.\ P\ a) \wedge (LEAST\ a.\ P\ a) \leq x)$
      have $\bigwedge y.\ P\ y \Longrightarrow x \leq y$
      proof (*rule classical*)
        fix y
        assume $P\ y$ **and** $\neg\ x \leq y$
        with *less* **have** $P\ (LEAST\ a.\ P\ a)$ **and** $(LEAST\ a.\ P\ a) \leq y$
          by (*auto simp add*: *not-le*)
        with *assm* **have** $x < (LEAST\ a.\ P\ a)$ **and** $(LEAST\ a.\ P\ a) \leq y$
          by *auto*
        then show $x \leq y$ **by** *auto*
      qed
      with ‹$P\ x$› **have** *Least*: $(LEAST\ a.\ P\ a) = x$
        by (*rule Least-equality*)
      with ‹$P\ x$› **show** *?thesis* **by** *simp*
    qed
  qed
  then show $P\ (LEAST\ x.\ P\ x)$ **and** $(LEAST\ x.\ P\ x) \leq k$ **by** *auto*

qed

lemmas *LeastI* = *wellorder-Least-lemma*(*1*)
lemmas *Least-le* = *wellorder-Least-lemma*(*2*)

— The following 3 lemmas are due to Brian Huffman
lemma *LeastI-ex*: $\exists\, x.\ P\ x \Longrightarrow P\ (\mathit{Least}\ P)$
by (*erule exE*) (*erule LeastI*)

lemma *LeastI2*:
$P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (\mathit{Least}\ P)$
by (*blast intro*: *LeastI*)

lemma *LeastI2-ex*:
$\exists\, a.\ P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (\mathit{Least}\ P)$
by (*blast intro*: *LeastI-ex*)

lemma *not-less-Least*: $k < (\mathit{LEAST}\ x.\ P\ x) \Longrightarrow \neg\ P\ k$
apply (*simp* (*no-asm-use*) *add*: *not-le* [*symmetric*])
apply (*erule contrapos-nn*)
apply (*erule Least-le*)
done

end

3.12 Order on bool

instantiation *bool* :: {*order*, *top*, *bot*}
begin

definition
le-bool-def [*code del*]: $P \leq Q \longleftrightarrow P \longrightarrow Q$

definition
less-bool-def [*code del*]: $(P{::}\mathit{bool}) < Q \longleftrightarrow \neg\ P \wedge Q$

definition
top-bool-eq: $\mathit{top} = \mathit{True}$

definition
bot-bool-eq: $\mathit{bot} = \mathit{False}$

instance proof
qed (*auto simp add*: *le-bool-def less-bool-def top-bool-eq bot-bool-eq*)

end

lemma *le-boolI*: $(P \Longrightarrow Q) \Longrightarrow P \leq Q$
by (*simp add*: *le-bool-def*)

lemma *le-boolI'*: $P \longrightarrow Q \Longrightarrow P \leq Q$
by (*simp add*: *le-bool-def*)

lemma *le-boolE*: $P \leq Q \Longrightarrow P \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
by (*simp add*: *le-bool-def*)

lemma *le-boolD*: $P \leq Q \Longrightarrow P \longrightarrow Q$
by (*simp add*: *le-bool-def*)

lemma [*code*]:
$\mathit{False} \leq b \longleftrightarrow \mathit{True}$
$\mathit{True} \leq b \longleftrightarrow b$
$\mathit{False} < b \longleftrightarrow b$
$\mathit{True} < b \longleftrightarrow \mathit{False}$
unfolding *le-bool-def less-bool-def* **by** *simp-all*

3.13 Order on functions

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

definition
le-fun-def [*code del*]: $f \leq g \longleftrightarrow (\forall x.\ f\ x \leq g\ x)$

definition
less-fun-def [*code del*]: $(f::'a \Rightarrow 'b) < g \longleftrightarrow f \leq g \wedge \neg\ (g \leq f)$

instance ..

end

instance *fun* :: (*type*, *preorder*) *preorder* **proof**
qed (*auto simp add*: *le-fun-def less-fun-def*
intro: *order-trans order-antisym intro*!: *ext*)

instance *fun* :: (*type*, *order*) *order* **proof**
qed (*auto simp add*: *le-fun-def intro*: *order-antisym ext*)

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

definition
top-fun-eq: $\mathit{top} = (\lambda x.\ \mathit{top})$

instance proof
qed (*simp add*: *top-fun-eq le-fun-def*)

end

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

definition
bot-fun-eq: $bot = (\lambda x.\ bot)$

instance proof
qed (*simp add*: *bot-fun-eq le-fun-def*)

end

lemma *le-funI*: $(\bigwedge x.\ f\ x \leq g\ x) \Longrightarrow f \leq g$
unfolding *le-fun-def* **by** *simp*

lemma *le-funE*: $f \leq g \Longrightarrow (f\ x \leq g\ x \Longrightarrow P) \Longrightarrow P$
unfolding *le-fun-def* **by** *simp*

lemma *le-funD*: $f \leq g \Longrightarrow f\ x \leq g\ x$
unfolding *le-fun-def* **by** *simp*

Handy introduction and elimination rules for $\leq$ on unary and binary predicates

lemma *predicate1I*:
assumes *PQ*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $P \leq Q$
apply (*rule le-funI*)
apply (*rule le-boolI*)
apply (*rule PQ*)
apply *assumption*
done

lemma *predicate1D* [*Pure.dest*, *dest*]: $P \leq Q \Longrightarrow P\ x \Longrightarrow Q\ x$
apply (*erule le-funE*)
apply (*erule le-boolE*)
apply *assumption*+
done

lemma *predicate2I* [*Pure.intro*!, *intro*!]:
assumes *PQ*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y$
shows $P \leq Q$
apply (*rule le-funI*)+
apply (*rule le-boolI*)
apply (*rule PQ*)
apply *assumption*
done

lemma *predicate2D* [*Pure.dest*, *dest*]: $P \leq Q \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
apply (*erule le-funE*)+

apply (*erule le-boolE*)
apply *assumption*+
done

lemma *rev-predicate1D*: *P x* ==> *P* <= *Q* ==> *Q x*
by (*rule predicate1D*)

lemma *rev-predicate2D*: *P x y* ==> *P* <= *Q* ==> *Q x y*
by (*rule predicate2D*)

end

4 Lattices: Abstract lattices

theory *Lattices*
imports *Orderings*
begin

4.1 Lattices

notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*)

class *lower-semilattice* = *order* +
fixes *inf* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** $\sqcap$ *70*)
assumes *inf-le1* [*simp*]: $x \sqcap y \sqsubseteq x$
and *inf-le2* [*simp*]: $x \sqcap y \sqsubseteq y$
and *inf-greatest*: $x \sqsubseteq y \Longrightarrow x \sqsubseteq z \Longrightarrow x \sqsubseteq y \sqcap z$

class *upper-semilattice* = *order* +
fixes *sup* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** $\sqcup$ *65*)
assumes *sup-ge1* [*simp*]: $x \sqsubseteq x \sqcup y$
and *sup-ge2* [*simp*]: $y \sqsubseteq x \sqcup y$
and *sup-least*: $y \sqsubseteq x \Longrightarrow z \sqsubseteq x \Longrightarrow y \sqcup z \sqsubseteq x$
begin

Dual lattice

lemma *dual-lattice*:
lower-semilattice (*op* ≥) (*op* >) *sup*
by (*rule lower-semilattice.intro*, *rule dual-order*)
(*unfold-locales*, *simp-all add*: *sup-least*)

end

class *lattice* = *lower-semilattice* + *upper-semilattice*

4.1.1 Intro and elim rules

context *lower-semilattice*
begin

lemma *le-infI1*[*intro*]:
  assumes $a \sqsubseteq x$
  shows $a \sqcap b \sqsubseteq x$
proof (*rule order-trans*)
  from *assms* **show** $a \sqsubseteq x$ **.**
  show $a \sqcap b \sqsubseteq a$ **by** *simp*
qed
lemmas (**in** −) [*rule del*] = *le-infI1*

lemma *le-infI2*[*intro*]:
  assumes $b \sqsubseteq x$
  shows $a \sqcap b \sqsubseteq x$
proof (*rule order-trans*)
  from *assms* **show** $b \sqsubseteq x$ **.**
  show $a \sqcap b \sqsubseteq b$ **by** *simp*
qed
lemmas (**in** −) [*rule del*] = *le-infI2*

lemma *le-infI*[*intro*!]: $x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcap b$
by(*blast intro*: *inf-greatest*)
lemmas (**in** −) [*rule del*] = *le-infI*

lemma *le-infE* [*elim*!]: $x \sqsubseteq a \sqcap b \Longrightarrow (x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow P) \Longrightarrow P$
  by (*blast intro*: *order-trans*)
lemmas (**in** −) [*rule del*] = *le-infE*

lemma *le-inf-iff* [*simp*]:
  $x \sqsubseteq y \sqcap z = (x \sqsubseteq y \wedge x \sqsubseteq z)$
by *blast*

lemma *le-iff-inf*: $(x \sqsubseteq y) = (x \sqcap y = x)$
  by (*blast intro*: *antisym dest*: *eq-iff* [*THEN iffD1*])

lemma *mono-inf*:
  fixes $f :: 'a \Rightarrow 'b$::*lower-semilattice*
  shows *mono* $f \Longrightarrow f\ (A \sqcap B) \leq f\ A \sqcap f\ B$
  by (*auto simp add*: *mono-def intro*: *Lattices.inf-greatest*)

end

context *upper-semilattice*
begin

lemma *le-supI1*[*intro*]: $x \sqsubseteq a \Longrightarrow x \sqsubseteq a \sqcup b$
  by (*rule order-trans*) *auto*

lemmas (**in** −) [*rule del*] = *le-supI1*

lemma *le-supI2*[*intro*]: $x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcup b$
by (*rule order-trans*) *auto*
lemmas (**in** −) [*rule del*] = *le-supI2*

lemma *le-supI*[*intro*!]: $a \sqsubseteq x \Longrightarrow b \sqsubseteq x \Longrightarrow a \sqcup b \sqsubseteq x$
by (*blast intro*: *sup-least*)
lemmas (**in** −) [*rule del*] = *le-supI*

lemma *le-supE*[*elim*!]: $a \sqcup b \sqsubseteq x \Longrightarrow (a \sqsubseteq x \Longrightarrow b \sqsubseteq x \Longrightarrow P) \Longrightarrow P$
by (*blast intro*: *order-trans*)
lemmas (**in** −) [*rule del*] = *le-supE*

lemma *ge-sup-conv*[*simp*]:
$x \sqcup y \sqsubseteq z = (x \sqsubseteq z \wedge y \sqsubseteq z)$
by *blast*

lemma *le-iff-sup*: $(x \sqsubseteq y) = (x \sqcup y = y)$
by (*blast intro*: *antisym dest*: *eq-iff* [*THEN iffD1*])

lemma *mono-sup*:
fixes $f :: 'a \Rightarrow 'b$::*upper-semilattice*
shows *mono* $f \Longrightarrow f\ A \sqcup f\ B \leq f\ (A \sqcup B)$
by (*auto simp add*: *mono-def intro*: *Lattices.sup-least*)

end

4.1.2 Equational laws

context *lower-semilattice*
begin

lemma *inf-commute*: $(x \sqcap y) = (y \sqcap x)$
by (*blast intro*: *antisym*)

lemma *inf-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
by (*blast intro*: *antisym*)

lemma *inf-idem*[*simp*]: $x \sqcap x = x$
by (*blast intro*: *antisym*)

lemma *inf-left-idem*[*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
by (*blast intro*: *antisym*)

lemma *inf-absorb1*: $x \sqsubseteq y \Longrightarrow x \sqcap y = x$
by (*blast intro*: *antisym*)

lemma *inf-absorb2*: $y \sqsubseteq x \Longrightarrow x \sqcap y = y$

by (*blast intro*: *antisym*)

lemma *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$
by (*blast intro*: *antisym*)

lemmas *inf-ACI* = *inf-commute inf-assoc inf-left-commute inf-left-idem*

end

context *upper-semilattice*
begin

lemma *sup-commute*: $(x \sqcup y) = (y \sqcup x)$
by (*blast intro*: *antisym*)

lemma *sup-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
by (*blast intro*: *antisym*)

lemma *sup-idem*[*simp*]: $x \sqcup x = x$
by (*blast intro*: *antisym*)

lemma *sup-left-idem*[*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
by (*blast intro*: *antisym*)

lemma *sup-absorb1*: $y \sqsubseteq x \Longrightarrow x \sqcup y = x$
by (*blast intro*: *antisym*)

lemma *sup-absorb2*: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
by (*blast intro*: *antisym*)

lemma *sup-left-commute*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
by (*blast intro*: *antisym*)

lemmas *sup-ACI* = *sup-commute sup-assoc sup-left-commute sup-left-idem*

end

context *lattice*
begin

lemma *inf-sup-absorb*: $x \sqcap (x \sqcup y) = x$
by (*blast intro*: *antisym inf-le1 inf-greatest sup-ge1*)

lemma *sup-inf-absorb*: $x \sqcup (x \sqcap y) = x$
by (*blast intro*: *antisym sup-ge1 sup-least inf-le1*)

lemmas *ACI* = *inf-ACI sup-ACI*

lemmas *inf-sup-ord* = *inf-le1 inf-le2 sup-ge1 sup-ge2*

Towards distributivity

lemma *distrib-sup-le*: $x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap (x \sqcup z)$
by *blast*

lemma *distrib-inf-le*: $(x \sqcap y) \sqcup (x \sqcap z) \sqsubseteq x \sqcap (y \sqcup z)$
by *blast*

If you have one of them, you have them all.

lemma *distrib-imp1*:
assumes *D*: $!!x\ y\ z.\ x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
shows $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
proof−
have $x \sqcup (y \sqcap z) = (x \sqcup (x \sqcap z)) \sqcup (y \sqcap z)$ **by**(*simp add*:*sup-inf-absorb*)
also have $\ldots = x \sqcup (z \sqcap (x \sqcup y))$ **by**(*simp add*:*D inf-commute sup-assoc*)
also have $\ldots = ((x \sqcup y) \sqcap x) \sqcup ((x \sqcup y) \sqcap z)$
by(*simp add*:*inf-sup-absorb inf-commute*)
also have $\ldots = (x \sqcup y) \sqcap (x \sqcup z)$ **by**(*simp add*:*D*)
finally show *?thesis* **.**
qed

lemma *distrib-imp2*:
assumes *D*: $!!x\ y\ z.\ x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
shows $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
proof−
have $x \sqcap (y \sqcup z) = (x \sqcap (x \sqcup z)) \sqcap (y \sqcup z)$ **by**(*simp add*:*inf-sup-absorb*)
also have $\ldots = x \sqcap (z \sqcup (x \sqcap y))$ **by**(*simp add*:*D sup-commute inf-assoc*)
also have $\ldots = ((x \sqcap y) \sqcup x) \sqcap ((x \sqcap y) \sqcup z)$
by(*simp add*:*sup-inf-absorb sup-commute*)
also have $\ldots = (x \sqcap y) \sqcup (x \sqcap z)$ **by**(*simp add*:*D*)
finally show *?thesis* **.**
qed

lemma *modular-le*: $x \sqsubseteq z \Longrightarrow x \sqcup (y \sqcap z) \sqsubseteq (x \sqcup y) \sqcap z$
by *blast*

end

4.2 Distributive lattices

class *distrib-lattice* = *lattice* +
assumes *sup-inf-distrib1*: $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

context *distrib-lattice*
begin

lemma *sup-inf-distrib2*:

$(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
by(*simp add*:*ACI sup-inf-distrib1*)

lemma *inf-sup-distrib1*:
$x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
by(*rule distrib-imp2*[*OF sup-inf-distrib1*])

lemma *inf-sup-distrib2*:
$(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
by(*simp add*:*ACI inf-sup-distrib1*)

lemmas *distrib* =
sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2

end

4.3 Uniqueness of inf and sup

lemma (**in** *lower-semilattice*) *inf-unique*:
fixes f (**infixl** $\triangle$ *70*)
assumes *le1*: $\bigwedge x\ y.\ x \triangle y \leq x$ **and** *le2*: $\bigwedge x\ y.\ x \triangle y \leq y$
and *greatest*: $\bigwedge x\ y\ z.\ x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \triangle z$
shows $x \sqcap y = x \triangle y$
proof (*rule antisym*)
show $x \triangle y \leq x \sqcap y$ **by** (*rule le-infI*) (*rule le1*, *rule le2*)
next
have *leI*: $\bigwedge x\ y\ z.\ x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \triangle z$ **by** (*blast intro*: *greatest*)
show $x \sqcap y \leq x \triangle y$ **by** (*rule leI*) *simp-all*
qed

lemma (**in** *upper-semilattice*) *sup-unique*:
fixes f (**infixl** ∇ *70*)
assumes *ge1* [*simp*]: $\bigwedge x\ y.\ x \leq x \nabla y$ **and** *ge2*: $\bigwedge x\ y.\ y \leq x \nabla y$
and *least*: $\bigwedge x\ y\ z.\ y \leq x \Longrightarrow z \leq x \Longrightarrow y \nabla z \leq x$
shows $x \sqcup y = x \nabla y$
proof (*rule antisym*)
show $x \sqcup y \leq x \nabla y$ **by** (*rule le-supI*) (*rule ge1*, *rule ge2*)
next
have *leI*: $\bigwedge x\ y\ z.\ x \leq z \Longrightarrow y \leq z \Longrightarrow x \nabla y \leq z$ **by** (*blast intro*: *least*)
show $x \nabla y \leq x \sqcup y$ **by** (*rule leI*) *simp-all*
qed

4.4 *min*/*max* on linear orders as special case of *op* $\sqcap$/*op* $\sqcup$

lemma (**in** *linorder*) *distrib-lattice-min-max*:
distrib-lattice (*op* $\leq$) (*op* $<$) *min max*
proof
have *aux*: $\bigwedge x\ y :: 'a.\ x < y \Longrightarrow y \leq x \Longrightarrow x = y$
by (*auto simp add*: *less-le antisym*)
fix $x\ y\ z$

show *max x (min y z) = min (max x y) (max x z)*
unfolding *min-def max-def*
by *auto*
qed (*auto simp add*: *min-def max-def not-le less-imp-le*)

interpretation *min-max*: *distrib-lattice op* $\leq$ :: *'a*::*linorder* $\Rightarrow$ *'a* $\Rightarrow$ *bool op* $<$ *min max*
by (*rule distrib-lattice-min-max*)

lemma *inf-min*: *inf* = (*min* :: *'a*::{*lower-semilattice*, *linorder*} $\Rightarrow$ *'a* $\Rightarrow$ *'a*)
by (*rule ext*)+ (*auto intro*: *antisym*)

lemma *sup-max*: *sup* = (*max* :: *'a*::{*upper-semilattice*, *linorder*} $\Rightarrow$ *'a* $\Rightarrow$ *'a*)
by (*rule ext*)+ (*auto intro*: *antisym*)

lemmas *le-maxI1* = *min-max.sup-ge1*
lemmas *le-maxI2* = *min-max.sup-ge2*

lemmas *max-ac* = *min-max.sup-assoc min-max.sup-commute*
mk-left-commute [*of max*, *OF min-max.sup-assoc min-max.sup-commute*]

lemmas *min-ac* = *min-max.inf-assoc min-max.inf-commute*
mk-left-commute [*of min*, *OF min-max.inf-assoc min-max.inf-commute*]

Now we have inherited antisymmetry as an intro-rule on all linear orders. This is a problem because it applies to bool, which is undesirable.

lemmas [*rule del*] = *min-max.le-infI min-max.le-supI*
min-max.le-supE min-max.le-infE min-max.le-supI1 min-max.le-supI2
min-max.le-infI1 min-max.le-infI2

4.5 Bool as lattice

instantiation *bool* :: *distrib-lattice*
begin

definition
inf-bool-eq: $P \sqcap Q \longleftrightarrow P \wedge Q$

definition
sup-bool-eq: $P \sqcup Q \longleftrightarrow P \vee Q$

instance
by *intro-classes* (*auto simp add*: *inf-bool-eq sup-bool-eq le-bool-def*)

end

4.6 Fun as lattice

instantiation *fun* :: (*type*, *lattice*) *lattice*

begin

definition
inf-fun-eq [*code del*]: $f \sqcap g = (\lambda x.\ f\ x \sqcap g\ x)$

definition
sup-fun-eq [*code del*]: $f \sqcup g = (\lambda x.\ f\ x \sqcup g\ x)$

instance
apply *intro-classes*
unfolding *inf-fun-eq sup-fun-eq*
apply (*auto intro*: *le-funI*)
apply (*rule le-funI*)
apply (*auto dest*: *le-funD*)
apply (*rule le-funI*)
apply (*auto dest*: *le-funD*)
done

end

instance *fun* :: (*type*, *distrib-lattice*) *distrib-lattice*
by *default* (*auto simp add*: *inf-fun-eq sup-fun-eq sup-inf-distrib1*)

redundant bindings

lemmas *inf-aci* = *inf-ACI*
lemmas *sup-aci* = *sup-ACI*

no-notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*)

end

5 Set: Set theory for higher-order logic

theory *Set*
imports *Lattices*
begin

A set in HOL is simply a predicate.

5.1 Basic syntax

global

types *'a set = 'a => bool*

consts
Collect :: *('a => bool) => 'a set* — comprehension
op : :: *'a => 'a set => bool* — membership
insert :: *'a => 'a set => 'a set*
Ball :: *'a set => ('a => bool) => bool* — bounded universal quantifiers

Bex :: *'a set => ('a => bool) => bool* — bounded existential quantifiers
Bex1 :: *'a set => ('a => bool) => bool* — bounded unique existential quantifiers
Pow :: *'a set => 'a set set* — powerset
image :: *('a => 'b) => 'a set => 'b set* (**infixr** *‘ 90*)

local

notation
op : (*op :*) **and**
op : (*(-/ : -) [50, 51] 50*)

abbreviation
not-mem x A == ~ (x : A) — non-membership

notation
not-mem (*op ~:*) **and**
not-mem (*(-/ ~: -) [50, 51] 50*)

notation (*xsymbols*)
op : (*op ∈*) **and**
op : (*(-/ ∈ -) [50, 51] 50*) **and**
not-mem (*op ∉*) **and**
not-mem (*(-/ ∉ -) [50, 51] 50*)

notation (*HTML* **output**)
op : (*op ∈*) **and**
op : (*(-/ ∈ -) [50, 51] 50*) **and**
not-mem (*op ∉*) **and**
not-mem (*(-/ ∉ -) [50, 51] 50*)

syntax
@Coll :: *pttrn => bool => 'a set* (*(1{-./ -})*)

translations
{x. P} == *Collect (%x. P)*

definition *empty* :: *'a set* ({}) **where**
empty ≡ {x. False}

definition *UNIV* :: *'a set* **where**
UNIV ≡ {*x*. *True*}

syntax
@*Finset* :: *args* => *'a set* ({(-)})

translations
{*x*, *xs*} == *insert x* {*xs*}
{*x*} == *insert x* {}

definition *Int* :: *'a set* ⇒ *'a set* ⇒ *'a set* (**infixl** *Int 70*) **where**
A Int B ≡ {*x*. *x* ∈ *A* ∧ *x* ∈ *B*}

definition *Un* :: *'a set* ⇒ *'a set* ⇒ *'a set* (**infixl** *Un 65*) **where**
A Un B ≡ {*x*. *x* ∈ *A* ∨ *x* ∈ *B*}

notation (*xsymbols*)
Int (**infixl** ∩ *70*) **and**
Un (**infixl** ∪ *65*)

notation (*HTML* **output**)
Int (**infixl** ∩ *70*) **and**
Un (**infixl** ∪ *65*)

syntax
-Ball :: *pttrn* => *'a set* => *bool* => *bool* ((*3ALL* -:-./ -) [*0*, *0*, *10*] *10*)
-Bex :: *pttrn* => *'a set* => *bool* => *bool* ((*3EX* -:-./ -) [*0*, *0*, *10*] *10*)
-Bex1 :: *pttrn* => *'a set* => *bool* => *bool* ((*3EX*! -:-./ -) [*0*, *0*, *10*] *10*)
-Bleast :: *id* => *'a set* => *bool* => *'a* ((*3LEAST* -:-./ -) [*0*, *0*, *10*] *10*)

syntax (*HOL*)
-Ball :: *pttrn* => *'a set* => *bool* => *bool* ((*3*! -:-./ -) [*0*, *0*, *10*] *10*)
-Bex :: *pttrn* => *'a set* => *bool* => *bool* ((*3?* -:-./ -) [*0*, *0*, *10*] *10*)
-Bex1 :: *pttrn* => *'a set* => *bool* => *bool* ((*3?*! -:-./ -) [*0*, *0*, *10*] *10*)

syntax (*xsymbols*)
-Ball :: *pttrn* => *'a set* => *bool* => *bool* ((*3*∀ -∈-./ -) [*0*, *0*, *10*] *10*)
-Bex :: *pttrn* => *'a set* => *bool* => *bool* ((*3*∃ -∈-./ -) [*0*, *0*, *10*] *10*)
-Bex1 :: *pttrn* => *'a set* => *bool* => *bool* ((*3*∃!-∈-./ -) [*0*, *0*, *10*] *10*)
-Bleast :: *id* => *'a set* => *bool* => *'a* ((*3LEAST*-∈-./ -) [*0*, *0*, *10*] *10*)

syntax (*HTML* **output**)
-Ball :: *pttrn* => *'a set* => *bool* => *bool* ((*3*∀ -∈-./ -) [*0*, *0*, *10*] *10*)
-Bex :: *pttrn* => *'a set* => *bool* => *bool* ((*3*∃ -∈-./ -) [*0*, *0*, *10*] *10*)
-Bex1 :: *pttrn* => *'a set* => *bool* => *bool* ((*3*∃!-∈-./ -) [*0*, *0*, *10*] *10*)

translations

ALL x:A. P == Ball A (%x. P)
EX x:A. P == Bex A (%x. P)
EX! x:A. P == Bex1 A (%x. P)
LEAST x:A. P => LEAST x. x:A & P

definition *INTER* :: *'a set ⇒ ('a ⇒ 'b set) ⇒ 'b set* **where**
INTER A B ≡ {y. ∀x∈A. y ∈ B x}

definition *UNION* :: *'a set ⇒ ('a ⇒ 'b set) ⇒ 'b set* **where**
UNION A B ≡ {y. ∃x∈A. y ∈ B x}

definition *Inter* :: *'a set set ⇒ 'a set* **where**
Inter S ≡ INTER S (λx. x)

definition *Union* :: *'a set set ⇒ 'a set* **where**
Union S ≡ UNION S (λx. x)

notation (*xsymbols*)
Inter (⋂- [*90*] *90*) **and**
Union (⋃- [*90*] *90*)

5.2 Additional concrete syntax

syntax
@SetCompr :: *'a => idts => bool => 'a set* ((*1*{- |/-./ -}))
@Collect :: *idt => 'a set => bool => 'a set* ((*1*{- :/ -./ -}))
@INTER1 :: *pttrns => 'b set => 'b set* ((*3INT* -./ -) [*0*, *10*] *10*)
@UNION1 :: *pttrns => 'b set => 'b set* ((*3UN* -./ -) [*0*, *10*] *10*)
@INTER :: *pttrn => 'a set => 'b set => 'b set* ((*3INT* -:-./ -) [*0*, *10*] *10*)
@UNION :: *pttrn => 'a set => 'b set => 'b set* ((*3UN* -:-./ -) [*0*, *10*] *10*)

syntax (*xsymbols*)
@Collect :: *idt => 'a set => bool => 'a set* ((*1*{- ∈/ -./ -}))
@INTER1 :: *pttrns => 'b set => 'b set* ((*3*⋂-./ -) [*0*, *10*] *10*)
@UNION1 :: *pttrns => 'b set => 'b set* ((*3*⋃-./ -) [*0*, *10*] *10*)
@INTER :: *pttrn => 'a set => 'b set => 'b set* ((*3*⋂-∈-./ -) [*0*, *10*] *10*)
@UNION :: *pttrn => 'a set => 'b set => 'b set* ((*3*⋃-∈-./ -) [*0*, *10*] *10*)

syntax (*latex* **output**)
@INTER1 :: *pttrns => 'b set => 'b set* ((*3*⋂(*00*-)/ -) [*0*, *10*] *10*)
@UNION1 :: *pttrns => 'b set => 'b set* ((*3*⋃(*00*-)/ -) [*0*, *10*] *10*)
@INTER :: *pttrn => 'a set => 'b set => 'b set* ((*3*⋂(*00*-∈-)/ -) [*0*, *10*] *10*)
@UNION :: *pttrn => 'a set => 'b set => 'b set* ((*3*⋃(*00*-∈-)/ -) [*0*, *10*] *10*)

translations
{x:A. P} => {x. x:A & P}
INT x y. B == INT x. INT y. B

INT x. B == *CONST INTER CONST UNIV* (%*x. B*)
INT x. B == *INT x*:*CONST UNIV. B*
INT x:*A. B* == *CONST INTER A* (%*x. B*)
UN x y. B == *UN x. UN y. B*
UN x. B == *CONST UNION CONST UNIV* (%*x. B*)
UN x. B == *UN x*:*CONST UNIV. B*
UN x:*A. B* == *CONST UNION A* (%*x. B*)

Note the difference between ordinary xsymbol syntax of indexed unions and intersections (e.g. $\bigcup a_1{\in}A_1.\ B$) and their LaTeX rendition: $\bigcup_{a_1 \in A_1} B$. The former does not make the index expression a subscript of the union/intersection symbol because this leads to problems with nested subscripts in Proof General.

abbreviation
subset :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
subset ≡ *less*

abbreviation
subset-eq :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
subset-eq ≡ *less-eq*

notation (**output**)
subset (*op* <) **and**
subset ((-/ < -) [*50*, *51*] *50*) **and**
subset-eq (*op* <=) **and**
subset-eq ((-/ <= -) [*50*, *51*] *50*)

notation (*xsymbols*)
subset (*op* ⊂) **and**
subset ((-/ ⊂ -) [*50*, *51*] *50*) **and**
subset-eq (*op* ⊆) **and**
subset-eq ((-/ ⊆ -) [*50*, *51*] *50*)

notation (*HTML* **output**)
subset (*op* ⊂) **and**
subset ((-/ ⊂ -) [*50*, *51*] *50*) **and**
subset-eq (*op* ⊆) **and**
subset-eq ((-/ ⊆ -) [*50*, *51*] *50*)

abbreviation (*input*)
supset :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
supset ≡ *greater*

abbreviation (*input*)
supset-eq :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
supset-eq ≡ *greater-eq*

notation (*xsymbols*)
supset (*op* ⊃) **and**

supset ((-/ ⊃ -) [*50*, *51*] *50*) **and**
supset-eq (*op* ⊇) **and**
supset-eq ((-/ ⊇ -) [*50*, *51*] *50*)

abbreviation
range :: ('*a* => '*b*) => '*b set* **where** — of function
range f == *f* ' *UNIV*

5.2.1 Bounded quantifiers

syntax (**output**)
-setlessAll :: [*idt*, '*a*, *bool*] => *bool* ((*3ALL* -<-./ -) [*0*, *0*, *10*] *10*)
-setlessEx :: [*idt*, '*a*, *bool*] => *bool* ((*3EX* -<-./ -) [*0*, *0*, *10*] *10*)
-setleAll :: [*idt*, '*a*, *bool*] => *bool* ((*3ALL* -<=-./ -) [*0*, *0*, *10*] *10*)
-setleEx :: [*idt*, '*a*, *bool*] => *bool* ((*3EX* -<=-./ -) [*0*, *0*, *10*] *10*)
-setleEx1 :: [*idt*, '*a*, *bool*] => *bool* ((*3EX*! -<=-./ -) [*0*, *0*, *10*] *10*)

syntax (*xsymbols*)
-setlessAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ -⊂-./ -) [*0*, *0*, *10*] *10*)
-setlessEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ -⊂-./ -) [*0*, *0*, *10*] *10*)
-setleAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ -⊆-./ -) [*0*, *0*, *10*] *10*)
-setleEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ -⊆-./ -) [*0*, *0*, *10*] *10*)
-setleEx1 :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃!-⊆-./ -) [*0*, *0*, *10*] *10*)

syntax (*HOL* **output**)
-setlessAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*! -<-./ -) [*0*, *0*, *10*] *10*)
-setlessEx :: [*idt*, '*a*, *bool*] => *bool* ((*3?* -<-./ -) [*0*, *0*, *10*] *10*)
-setleAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*! -<=-./ -) [*0*, *0*, *10*] *10*)
-setleEx :: [*idt*, '*a*, *bool*] => *bool* ((*3?* -<=-./ -) [*0*, *0*, *10*] *10*)
-setleEx1 :: [*idt*, '*a*, *bool*] => *bool* ((*3?*! -<=-./ -) [*0*, *0*, *10*] *10*)

syntax (*HTML* **output**)
-setlessAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ -⊂-./ -) [*0*, *0*, *10*] *10*)
-setlessEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ -⊂-./ -) [*0*, *0*, *10*] *10*)
-setleAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ -⊆-./ -) [*0*, *0*, *10*] *10*)
-setleEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ -⊆-./ -) [*0*, *0*, *10*] *10*)
-setleEx1 :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃!-⊆-./ -) [*0*, *0*, *10*] *10*)

translations
∀ *A*⊂*B*. *P* => *ALL A*. *A* ⊂ *B* --> *P*
∃ *A*⊂*B*. *P* => *EX A*. *A* ⊂ *B* & *P*
∀ *A*⊆*B*. *P* => *ALL A*. *A* ⊆ *B* --> *P*
∃ *A*⊆*B*. *P* => *EX A*. *A* ⊆ *B* & *P*
∃!*A*⊆*B*. *P* => *EX*! *A*. *A* ⊆ *B* & *P*

print-translation ⟪
let
val Type (*set-type*, -) = @{*typ* '*a set*};
val All-binder = *Syntax.binder-name* @{*const-syntax All*};

```
  val Ex-binder = Syntax.binder-name @{const-syntax Ex};
  val impl = @{const-syntax op -->};
  val conj = @{const-syntax op &};
  val sbset = @{const-syntax subset};
  val sbset-eq = @{const-syntax subset-eq};

  val trans =
   [((All-binder, impl, sbset), -setlessAll),
    ((All-binder, impl, sbset-eq), -setleAll),
    ((Ex-binder, conj, sbset), -setlessEx),
    ((Ex-binder, conj, sbset-eq), -setleEx)];

  fun mk v v' c n P =
    if v = v' andalso not (Term.exists-subterm (fn Free (x, -) => x = v | - =>
false) n)
    then Syntax.const c $ Syntax.mark-bound v' $ n $ P else raise Match;

  fun tr' q = (q,
    fn [Const (-bound, -) $ Free (v, Type (T, -)), Const (c, -) $ (Const (d, -) $
(Const (-bound, -) $ Free (v', -)) $ n) $ P] =>
        if T = (set-type) then case AList.lookup (op =) trans (q, c, d)
         of NONE => raise Match
          | SOME l => mk v v' l n P
        else raise Match
     | - => raise Match);
in
  [tr' All-binder, tr' Ex-binder]
end
⟫
```

Translate between $\{e \mid x1...xn.\ P\}$ and $\{u.\ EX\ x1..xn.\ u = e\ \&\ P\}$; $\{y.\ EX\ x1..xn.\ y = e\ \&\ P\}$ is only translated if $[0..n]$ *subset* $bvs(e)$.

parse-translation ⟪

```
  let
    val ex-tr = snd (mk-binder-tr (EX , Ex));

    fun nvars (Const (-idts, -) $ - $ idts) = nvars idts + 1
      | nvars - = 1;

    fun setcompr-tr [e, idts, b] =
      let
        val eq = Syntax.const op = $ Bound (nvars idts) $ e;
        val P = Syntax.const op & $ eq $ b;
        val exP = ex-tr [idts, P];
      in Syntax.const Collect $ Term.absdummy (dummyT, exP) end;

  in [(@SetCompr, setcompr-tr)] end;
⟫
```

```
print_translation ⟪
let
  fun btr' syn [A, Abs abs] =
    let val (x, t) = atomic_abs_tr' abs
    in Syntax.const syn $ x $ A $ t end
in
[(@{const_syntax Ball}, btr' Ball), (@{const_syntax Bex}, btr' Bex),
 (@{const_syntax UNION}, btr' @UNION),(@{const_syntax INTER}, btr' @INTER)]
end
⟫

print_translation ⟪
let
  val ex_tr' = snd (mk_binder_tr' (Ex, DUMMY));

  fun setcompr_tr' [Abs (abs as (_, _, P))] =
    let
      fun check (Const (Ex, _) $ Abs (_, _, P), n) = check (P, n + 1)
        | check (Const (op &, _) $ (Const (op =, _) $ Bound m $ e) $ P, n) =
            n > 0 andalso m = n andalso not (loose_bvar1 (P, n)) andalso
            ((0 upto (n - 1)) subset add_loose_bnos (e, 0, []))
        | check _ = false

      fun tr' (_ $ abs) =
        let val _ $ idts $ (_ $ (_ $ _ $ e) $ Q) = ex_tr' [abs]
        in Syntax.const @SetCompr $ e $ idts $ Q end;
    in if check (P, 0) then tr' P
       else let val (x as _ $ Free(xN,_), t) = atomic_abs_tr' abs
                val M = Syntax.const @Coll $ x $ t
            in case t of
                 Const(op &,_)
                   $ (Const(op :,_) $ (Const(_bound,_) $ Free(yN,_)) $ A)
                   $ P =>
                   if xN=yN then Syntax.const @Collect $ x $ A $ P else M
               | _ => M
            end
    end;
  in [(Collect, setcompr_tr')] end;
⟫
```

5.3 Rules and definitions

Isomorphisms between predicates and sets.

defs

```
  mem_def [code]: x : S == S x
  Collect_def [code]: Collect P == P
```

defs

Ball-def: *Ball A P* == *ALL x. x:A −−> P(x)*
Bex-def: *Bex A P* == *EX x. x:A & P(x)*
Bex1-def: *Bex1 A P* == *EX! x. x:A & P(x)*

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

definition
fun-diff-def: $A - B = (\%x.\ A\ x - B\ x)$

instance ..

end

instantiation *bool* :: *minus*
begin

definition
bool-diff-def: $A - B = (A\ \&\ \sim B)$

instance ..

end

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

definition
fun-Compl-def: $- A = (\%x.\ - A\ x)$

instance ..

end

instantiation *bool* :: *uminus*
begin

definition
bool-Compl-def: $- A = (\sim A)$

instance ..

end

defs
Pow-def: *Pow A* == $\{B.\ B <= A\}$
insert-def: *insert a B* == $\{x.\ x=a\}$ *Un B*
image-def: *f‘A* == $\{y.\ EX\ x:A.\ y = f(x)\}$

5.4 Lemmas and proof tool setup

5.4.1 Relating predicates and sets

lemma *mem-Collect-eq* [*iff*]: $(a : \{x.\ P(x)\}) = P(a)$
by (*simp add*: *Collect-def mem-def*)

lemma *Collect-mem-eq* [*simp*]: $\{x.\ x{:}A\} = A$
by (*simp add*: *Collect-def mem-def*)

lemma *CollectI*: $P(a) \Longrightarrow a : \{x.\ P(x)\}$
by *simp*

lemma *CollectD*: $a : \{x.\ P(x)\} \Longrightarrow P(a)$
by *simp*

lemma *Collect-cong*: $(!!x.\ P\ x = Q\ x) \Longrightarrow \{x.\ P(x)\} = \{x.\ Q(x)\}$
by *simp*

lemmas *CollectE* = *CollectD* [*elim-format*]

5.4.2 Bounded quantifiers

lemma *ballI* [*intro*!]: $(!!x.\ x{:}A \Longrightarrow P\ x) \Longrightarrow ALL\ x{:}A.\ P\ x$
by (*simp add*: *Ball-def*)

lemmas *strip* = *impI allI ballI*

lemma *bspec* [*dest?*]: $ALL\ x{:}A.\ P\ x \Longrightarrow x{:}A \Longrightarrow P\ x$
by (*simp add*: *Ball-def*)

lemma *ballE* [*elim*]: $ALL\ x{:}A.\ P\ x \Longrightarrow (P\ x \Longrightarrow Q) \Longrightarrow (x\ ^{\sim}{:}\ A \Longrightarrow Q) \Longrightarrow Q$
by (*unfold Ball-def*) *blast*

ML ⟪ *bind-thm* (*rev-ballE*, *permute-prems 1 1* @{*thm ballE*}) ⟫

This tactic takes assumptions $\forall x{\in}A.\ P\ x$ and $a \in A$; creates assumption $P\ a$.

ML ⟪
fun ball-tac i = etac @{*thm ballE*} *i THEN contr-tac* (*i + 1*)
⟫

Gives better instantiation for bound:

declaration ⟪ *fn - =>*
Classical.map-cs (*fn cs => cs addbefore* (*bspec*, *datac* @{*thm bspec*} *1*))
⟫

lemma *bexI* [*intro*]: $P\ x \Longrightarrow x{:}A \Longrightarrow EX\ x{:}A.\ P\ x$

— Normally the best argument order: $P\ x$ constrains the choice of $x \in A$.
by (*unfold Bex-def*) *blast*

lemma *rev-bexI* [*intro?*]: $x{:}A \Longrightarrow P\ x \Longrightarrow EX\ x{:}A.\ P\ x$
— The best argument order when there is only one $x \in A$.
by (*unfold Bex-def*) *blast*

lemma *bexCI*: $(ALL\ x{:}A.\ \sim P\ x \Longrightarrow P\ a) \Longrightarrow a{:}A \Longrightarrow EX\ x{:}A.\ P\ x$
by (*unfold Bex-def*) *blast*

lemma *bexE* [*elim*!]: $EX\ x{:}A.\ P\ x \Longrightarrow (!!x.\ x{:}A \Longrightarrow P\ x \Longrightarrow Q) \Longrightarrow Q$
by (*unfold Bex-def*) *blast*

lemma *ball-triv* [*simp*]: $(ALL\ x{:}A.\ P) = ((EX\ x.\ x{:}A) \longrightarrow P)$
— Trival rewrite rule.
by (*simp add*: *Ball-def*)

lemma *bex-triv* [*simp*]: $(EX\ x{:}A.\ P) = ((EX\ x.\ x{:}A) \mathbin{\&} P)$
— Dual form for existentials.
by (*simp add*: *Bex-def*)

lemma *bex-triv-one-point1* [*simp*]: $(EX\ x{:}A.\ x = a) = (a{:}A)$
by *blast*

lemma *bex-triv-one-point2* [*simp*]: $(EX\ x{:}A.\ a = x) = (a{:}A)$
by *blast*

lemma *bex-one-point1* [*simp*]: $(EX\ x{:}A.\ x = a \mathbin{\&} P\ x) = (a{:}A \mathbin{\&} P\ a)$
by *blast*

lemma *bex-one-point2* [*simp*]: $(EX\ x{:}A.\ a = x \mathbin{\&} P\ x) = (a{:}A \mathbin{\&} P\ a)$
by *blast*

lemma *ball-one-point1* [*simp*]: $(ALL\ x{:}A.\ x = a \longrightarrow P\ x) = (a{:}A \longrightarrow P\ a)$
by *blast*

lemma *ball-one-point2* [*simp*]: $(ALL\ x{:}A.\ a = x \longrightarrow P\ x) = (a{:}A \longrightarrow P\ a)$
by *blast*

ML ⟪
local
val unfold-bex-tac = unfold-tac @{*thms Bex-def*};
fun prove-bex-tac ss = unfold-bex-tac ss THEN Quantifier1.prove-one-point-ex-tac;
val rearrange-bex = Quantifier1.rearrange-bex prove-bex-tac;

val unfold-ball-tac = unfold-tac @{*thms Ball-def*};
fun prove-ball-tac ss = unfold-ball-tac ss THEN Quantifier1.prove-one-point-all-tac;
val rearrange-ball = Quantifier1.rearrange-ball prove-ball-tac;
in

```
  val defBEX-regroup = Simplifier.simproc (the-context ())
    defined BEX [EX x:A. P x & Q x] rearrange-bex;
  val defBALL-regroup = Simplifier.simproc (the-context ())
    defined BALL [ALL x:A. P x --> Q x] rearrange-ball;
end;

Addsimprocs [defBALL-regroup, defBEX-regroup];
⟫
```

5.4.3 Congruence rules

lemma *ball-cong*:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow P\ x = Q\ x) \Longrightarrow (ALL\ x{:}A.\ P\ x) = (ALL\ x{:}B.\ Q\ x)$
by (*simp add*: *Ball-def*)

lemma *strong-ball-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow_{simp} P\ x = Q\ x) \Longrightarrow (ALL\ x{:}A.\ P\ x) = (ALL\ x{:}B.\ Q\ x)$
by (*simp add*: *simp-implies-def Ball-def*)

lemma *bex-cong*:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow P\ x = Q\ x) \Longrightarrow (EX\ x{:}A.\ P\ x) = (EX\ x{:}B.\ Q\ x)$
by (*simp add*: *Bex-def cong*: *conj-cong*)

lemma *strong-bex-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow_{simp} P\ x = Q\ x) \Longrightarrow (EX\ x{:}A.\ P\ x) = (EX\ x{:}B.\ Q\ x)$
by (*simp add*: *simp-implies-def Bex-def cong*: *conj-cong*)

5.4.4 Subsets

lemma *subsetI* [*atp*,*intro*!]: $(!!x.\ x{:}A \Longrightarrow x{:}B) \Longrightarrow A \subseteq B$
by (*auto simp add*: *mem-def intro*: *predicate1I*)

Map the type $'a\ set \Rightarrow anything$ to just $'a$; for overloading constants whose first argument has type $'a\ set$.

lemma *subsetD* [*elim*, *intro?*]: $A \subseteq B \Longrightarrow c \in A \Longrightarrow c \in B$
— Rule in Modus Ponens style.
by (*unfold mem-def*) *blast*

lemma *rev-subsetD* [*intro?*]: $c \in A \Longrightarrow A \subseteq B \Longrightarrow c \in B$
— The same, with reversed premises for use with *erule* – cf *rev-mp*.
by (*rule subsetD*)

Converts $A \subseteq B$ to $x \in A \Longrightarrow x \in B$.

ML ⟪

fun impOfSubs th = th RSN (2, @{thm rev-subsetD})
⟫

lemma *subsetCE* [*elim*]: $A \subseteq B \Longrightarrow (c \notin A \Longrightarrow P) \Longrightarrow (c \in B \Longrightarrow P) \Longrightarrow P$
— Classical elimination rule.
by (*unfold mem-def*) *blast*

lemma *subset-eq*: $A \leq B = (\forall x{\in}A.\ x \in B)$ **by** *blast*

Takes assumptions $A \subseteq B$; $c \in A$ and creates the assumption $c \in B$.
ML ⟪
fun set-mp-tac i = etac @{thm subsetCE} i THEN mp-tac i
⟫

lemma *contra-subsetD*: $A \subseteq B \Longrightarrow c \notin B \Longrightarrow c \notin A$
by *blast*

lemma *subset-refl* [*simp*,*atp*]: $A \subseteq A$
by *fast*

lemma *subset-trans*: $A \subseteq B \Longrightarrow B \subseteq C \Longrightarrow A \subseteq C$
by *blast*

5.4.5 Equality

lemma *set-ext*: **assumes** *prem*: $(!!x.\ (x{:}A) = (x{:}B))$ **shows** $A = B$
apply (*rule prem* [*THEN ext*, *THEN arg-cong*, *THEN box-equals*])
apply (*rule Collect-mem-eq*)
apply (*rule Collect-mem-eq*)
done

lemma *expand-set-eq*: $(A = B) = (ALL\ x.\ (x{:}A) = (x{:}B))$
by(*auto intro*:*set-ext*)

lemma *subset-antisym* [*intro*!]: $A \subseteq B \Longrightarrow B \subseteq A \Longrightarrow A = B$
— Anti-symmetry of the subset relation.
by (*iprover intro*: *set-ext subsetD*)

Equality rules from ZF set theory – are they appropriate here?
lemma *equalityD1*: $A = B \Longrightarrow A \subseteq B$
by (*simp add*: *subset-refl*)

lemma *equalityD2*: $A = B \Longrightarrow B \subseteq A$
by (*simp add*: *subset-refl*)

Be careful when adding this to the claset as *subset-empty* is in the simpset: $A = \{\}$ goes to $\{\} \subseteq A$ and $A \subseteq \{\}$ and then back to $A = \{\}$!

lemma *equalityE*: $A = B \Longrightarrow (A \subseteq B \Longrightarrow B \subseteq A \Longrightarrow P) \Longrightarrow P$
by (*simp add*: *subset-refl*)

lemma *equalityCE* [*elim*]:
$A = B \Longrightarrow (c \in A \Longrightarrow c \in B \Longrightarrow P) \Longrightarrow (c \notin A \Longrightarrow c \notin B \Longrightarrow P) \Longrightarrow P$
by *blast*

lemma *eqset-imp-iff*: $A = B \Longrightarrow (x : A) = (x : B)$
by *simp*

lemma *eqelem-imp-iff*: $x = y \Longrightarrow (x : A) = (y : A)$
by *simp*

5.4.6 The universal set – UNIV

lemma *UNIV-I* [*simp*]: $x : UNIV$
by (*simp add*: *UNIV-def*)

declare *UNIV-I* [*intro*] — unsafe makes it less likely to cause problems

lemma *UNIV-witness* [*intro?*]: $EX\ x.\ x : UNIV$
by *simp*

lemma *subset-UNIV* [*simp*]: $A \subseteq UNIV$
by (*rule subsetI*) (*rule UNIV-I*)

Eta-contracting these two rules (to remove P) causes them to be ignored because of their interaction with congruence rules.

lemma *ball-UNIV* [*simp*]: $Ball\ UNIV\ P = All\ P$
by (*simp add*: *Ball-def*)

lemma *bex-UNIV* [*simp*]: $Bex\ UNIV\ P = Ex\ P$
by (*simp add*: *Bex-def*)

lemma *UNIV-eq-I*: $(\bigwedge x.\ x \in A) \Longrightarrow UNIV = A$
by *auto*

5.4.7 The empty set

lemma *empty-iff* [*simp*]: $(c : \{\}) = False$
by (*simp add*: *empty-def*)

lemma *emptyE* [*elim!*]: $a : \{\} \Longrightarrow P$
by *simp*

lemma *empty-subsetI* [*iff*]: $\{\} \subseteq A$
— One effect is to delete the ASSUMPTION $\{\} \subseteq A$
by *blast*

lemma *equals0I*: $(!!y.\ y \in A ==> False) ==> A = \{\}$
by *blast*

lemma *equals0D*: $A = \{\} ==> a \notin A$
— Use for reasoning about disjointness: $A \cap B = \{\}$
by *blast*

lemma *ball-empty* [*simp*]: *Ball* $\{\}$ P = *True*
by (*simp add*: *Ball-def*)

lemma *bex-empty* [*simp*]: *Bex* $\{\}$ P = *False*
by (*simp add*: *Bex-def*)

lemma *UNIV-not-empty* [*iff*]: *UNIV* $\tilde{}=$ $\{\}$
by (*blast elim*: *equalityE*)

5.4.8 The Powerset operator – Pow

lemma *Pow-iff* [*iff*]: $(A \in Pow\ B) = (A \subseteq B)$
by (*simp add*: *Pow-def*)

lemma *PowI*: $A \subseteq B ==> A \in Pow\ B$
by (*simp add*: *Pow-def*)

lemma *PowD*: $A \in Pow\ B ==> A \subseteq B$
by (*simp add*: *Pow-def*)

lemma *Pow-bottom*: $\{\} \in Pow\ B$
by *simp*

lemma *Pow-top*: $A \in Pow\ A$
by (*simp add*: *subset-refl*)

5.4.9 Set complement

lemma *Compl-iff* [*simp*]: $(c \in -A) = (c \notin A)$
by (*simp add*: *mem-def fun-Compl-def bool-Compl-def*)

lemma *ComplI* [*intro!*]: $(c \in A ==> False) ==> c \in -A$
by (*unfold mem-def fun-Compl-def bool-Compl-def*) *blast*

This form, with negated conclusion, works well with the Classical prover. Negated assumptions behave like formulae on the right side of the notional turnstile ...

lemma *ComplD* [*dest!*]: $c : -A ==> c\tilde{}:A$
by (*simp add*: *mem-def fun-Compl-def bool-Compl-def*)

lemmas *ComplE* = *ComplD* [*elim-format*]

lemma *Compl-eq*: $-A = \{x.\ \sim x : A\}$ **by** *blast*

5.4.10 Binary union – Un

lemma *Un-iff* [*simp*]: $(c : A\ Un\ B) = (c{:}A \mid c{:}B)$
by (*unfold Un-def*) *blast*

lemma *UnI1* [*elim?*]: $c{:}A \Longrightarrow c : A\ Un\ B$
by *simp*

lemma *UnI2* [*elim?*]: $c{:}B \Longrightarrow c : A\ Un\ B$
by *simp*

Classical introduction rule: no commitment to A vs B.

lemma *UnCI* [*intro*!]: $(c^{\sim}{:}B \Longrightarrow c{:}A) \Longrightarrow c : A\ Un\ B$
by *auto*

lemma *UnE* [*elim*!]: $c : A\ Un\ B \Longrightarrow (c{:}A \Longrightarrow P) \Longrightarrow (c{:}B \Longrightarrow P) \Longrightarrow P$
by (*unfold Un-def*) *blast*

5.4.11 Binary intersection – Int

lemma *Int-iff* [*simp*]: $(c : A\ Int\ B) = (c{:}A \,\&\, c{:}B)$
by (*unfold Int-def*) *blast*

lemma *IntI* [*intro*!]: $c{:}A \Longrightarrow c{:}B \Longrightarrow c : A\ Int\ B$
by *simp*

lemma *IntD1*: $c : A\ Int\ B \Longrightarrow c{:}A$
by *simp*

lemma *IntD2*: $c : A\ Int\ B \Longrightarrow c{:}B$
by *simp*

lemma *IntE* [*elim*!]: $c : A\ Int\ B \Longrightarrow (c{:}A \Longrightarrow c{:}B \Longrightarrow P) \Longrightarrow P$
by *simp*

5.4.12 Set difference

lemma *Diff-iff* [*simp*]: $(c : A - B) = (c{:}A \,\&\, c^{\sim}{:}B)$
by (*simp add*: *mem-def fun-diff-def bool-diff-def*)

lemma *DiffI* [*intro*!]: $c : A \Longrightarrow c\ ^{\sim}{:}\ B \Longrightarrow c : A - B$
by *simp*

lemma *DiffD1*: $c : A - B \Longrightarrow c : A$
by *simp*

lemma *DiffD2*: $c : A - B \Longrightarrow c : B \Longrightarrow P$
by *simp*

lemma *DiffE* [*elim*!]: $c : A - B \Longrightarrow (c{:}A \Longrightarrow c^{\sim}{:}B \Longrightarrow P) \Longrightarrow P$
by *simp*

lemma *set-diff-eq*: $A - B = \{x.\ x : A\ \&\ \sim x : B\}$ **by** *blast*

lemma *Compl-eq-Diff-UNIV*: $-A = (UNIV - A)$
by *blast*

5.4.13 Augmenting a set − insert

lemma *insert-iff* [*simp*]: $(a : insert\ b\ A) = (a = b \mid a{:}A)$
by (*unfold insert-def*) *blast*

lemma *insertI1*: $a : insert\ a\ B$
by *simp*

lemma *insertI2*: $a : B \Longrightarrow a : insert\ b\ B$
by *simp*

lemma *insertE* [*elim*!]: $a : insert\ b\ A \Longrightarrow (a = b \Longrightarrow P) \Longrightarrow (a{:}A \Longrightarrow P) \Longrightarrow P$
by (*unfold insert-def*) *blast*

lemma *insertCI* [*intro*!]: $(a^{\sim}{:}B \Longrightarrow a = b) \Longrightarrow a{:}\ insert\ b\ B$
— Classical introduction rule.
by *auto*

lemma *subset-insert-iff*: $(A \subseteq insert\ x\ B) = (if\ x{:}A\ then\ A - \{x\} \subseteq B\ else\ A \subseteq B)$
by *auto*

lemma *set-insert*:
assumes $x \in A$
obtains B **where** $A = insert\ x\ B$ **and** $x \notin B$
proof
from *assms* **show** $A = insert\ x\ (A - \{x\})$ **by** *blast*
next
show $x \notin A - \{x\}$ **by** *blast*
qed

lemma *insert-ident*: $x\ ^{\sim}{:}\ A \Longrightarrow x\ ^{\sim}{:}\ B \Longrightarrow (insert\ x\ A = insert\ x\ B) = (A = B)$
by *auto*

5.4.14 Singletons, using insert

lemma *singletonI* [*intro*!,*noatp*]: $a : \{a\}$
— Redundant? But unlike *insertCI*, it proves the subgoal immediately!
by (*rule insertI1*)

lemma *singletonD* [*dest*!,*noatp*]: $b : \{a\} \Longrightarrow b = a$
by *blast*

lemmas *singletonE* = *singletonD* [*elim-format*]

lemma *singleton-iff*: $(b : \{a\}) = (b = a)$
by *blast*

lemma *singleton-inject* [*dest*!]: $\{a\} = \{b\} \Longrightarrow a = b$
by *blast*

lemma *singleton-insert-inj-eq* [*iff*,*noatp*]:
$(\{b\} = \mathit{insert}\ a\ A) = (a = b \ \&\ A \subseteq \{b\})$
by *blast*

lemma *singleton-insert-inj-eq'* [*iff*,*noatp*]:
$(\mathit{insert}\ a\ A = \{b\}) = (a = b \ \&\ A \subseteq \{b\})$
by *blast*

lemma *subset-singletonD*: $A \subseteq \{x\} \Longrightarrow A = \{\} \mid A = \{x\}$
by *fast*

lemma *singleton-conv* [*simp*]: $\{x.\ x = a\} = \{a\}$
by *blast*

lemma *singleton-conv2* [*simp*]: $\{x.\ a = x\} = \{a\}$
by *blast*

lemma *diff-single-insert*: $A - \{x\} \subseteq B \Longrightarrow x \in A \Longrightarrow A \subseteq \mathit{insert}\ x\ B$
by *blast*

lemma *doubleton-eq-iff*: $(\{a,b\} = \{c,d\}) = (a{=}c \ \&\ b{=}d \mid a{=}d \ \&\ b{=}c)$
by (*blast elim*: *equalityE*)

5.4.15 Unions of families

UN $x{:}A.\ B\ x$ is $\bigcup B$ ‘ A.

declare *UNION-def* [*noatp*]

lemma *UN-iff* [*simp*]: $(b{:}\ (\mathit{UN}\ x{:}A.\ B\ x)) = (\mathit{EX}\ x{:}A.\ b{:}\ B\ x)$
by (*unfold UNION-def*) *blast*

lemma *UN-I* [*intro*]: $a{:}A \Longrightarrow b{:}\ B\ a \Longrightarrow b{:}\ (\mathit{UN}\ x{:}A.\ B\ x)$

— The order of the premises presupposes that A is rigid; b may be flexible.
by *auto*

lemma *UN-E* [*elim*!]: $b : (UN\ x{:}A.\ B\ x) \Longrightarrow (!!x.\ x{:}A \Longrightarrow b{:}\ B\ x \Longrightarrow R) \Longrightarrow R$
by (*unfold UNION-def*) *blast*

lemma *UN-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow C\ x = D\ x) \Longrightarrow (UN\ x{:}A.\ C\ x) = (UN\ x{:}B.\ D\ x)$
by (*simp add*: *UNION-def*)

lemma *strong-UN-cong*:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow_{simp} C\ x = D\ x) \Longrightarrow (UN\ x{:}A.\ C\ x) = (UN\ x{:}B.\ D\ x)$
by (*simp add*: *UNION-def simp-implies-def*)

5.4.16 Intersections of families

$INT\ x{:}A.\ B\ x$ is $\bigcap B\ `\ A$.

lemma *INT-iff* [*simp*]: $(b{:}\ (INT\ x{:}A.\ B\ x)) = (ALL\ x{:}A.\ b{:}\ B\ x)$
by (*unfold INTER-def*) *blast*

lemma *INT-I* [*intro*!]: $(!!x.\ x{:}A \Longrightarrow b{:}\ B\ x) \Longrightarrow b : (INT\ x{:}A.\ B\ x)$
by (*unfold INTER-def*) *blast*

lemma *INT-D* [*elim*]: $b : (INT\ x{:}A.\ B\ x) \Longrightarrow a{:}A \Longrightarrow b{:}\ B\ a$
by *auto*

lemma *INT-E* [*elim*]: $b : (INT\ x{:}A.\ B\ x) \Longrightarrow (b{:}\ B\ a \Longrightarrow R) \Longrightarrow (a^{\sim}{:}A \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination – by the Excluded Middle on $a \in A$.
by (*unfold INTER-def*) *blast*

lemma *INT-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow C\ x = D\ x) \Longrightarrow (INT\ x{:}A.\ C\ x) = (INT\ x{:}B.\ D\ x)$
by (*simp add*: *INTER-def*)

5.4.17 Union

lemma *Union-iff* [*simp*,*noatp*]: $(A : Union\ C) = (EX\ X{:}C.\ A{:}X)$
by (*unfold Union-def*) *blast*

lemma *UnionI* [*intro*]: $X{:}C \Longrightarrow A{:}X \Longrightarrow A : Union\ C$
— The order of the premises presupposes that C is rigid; A may be flexible.
by *auto*

lemma *UnionE* [*elim*!]: $A : Union\ C \Longrightarrow (!!X.\ A{:}X \Longrightarrow X{:}C \Longrightarrow R) \Longrightarrow R$
by (*unfold Union-def*) *blast*

5.4.18 Inter

lemma *Inter-iff* [*simp*,*noatp*]: $(A : Inter\ C) = (ALL\ X{:}C.\ A{:}X)$
by (*unfold Inter-def*) *blast*

lemma *InterI* [*intro*!]: $(!!X.\ X{:}C \Longrightarrow A{:}X) \Longrightarrow A : Inter\ C$
by (*simp add*: *Inter-def*)

A "destruct" rule – every X in C contains A as an element, but $A \in X$ can hold when $X \in C$ does not! This rule is analogous to *spec*.

lemma *InterD* [*elim*]: $A : Inter\ C \Longrightarrow X{:}C \Longrightarrow A{:}X$
by *auto*

lemma *InterE* [*elim*]: $A : Inter\ C \Longrightarrow (X^{\sim}{:}C \Longrightarrow R) \Longrightarrow (A{:}X \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination rule – does not require proving $X \in C$.
by (*unfold Inter-def*) *blast*

Image of a set under a function. Frequently b does not have the syntactic form of $f\ x$.

declare *image-def* [*noatp*]

lemma *image-eqI* [*simp*, *intro*]: $b = f\ x \Longrightarrow x{:}A \Longrightarrow b : f`A$
by (*unfold image-def*) *blast*

lemma *imageI*: $x : A \Longrightarrow f\ x : f\ `\ A$
by (*rule image-eqI*) (*rule refl*)

lemma *rev-image-eqI*: $x{:}A \Longrightarrow b = f\ x \Longrightarrow b : f`A$
— This version's more effective when we already have the required x.
by (*unfold image-def*) *blast*

lemma *imageE* [*elim*!]:
$b : (\%x.\ f\ x)`A \Longrightarrow (!!x.\ b = f\ x \Longrightarrow x{:}A \Longrightarrow P) \Longrightarrow P$
— The eta-expansion gives variable-name preservation.
by (*unfold image-def*) *blast*

lemma *image-Un*: $f`(A\ Un\ B) = f`A\ Un\ f`B$
by *blast*

lemma *image-eq-UN*: $f`A = (UN\ x{:}A.\ \{f\ x\})$
by *blast*

lemma *image-iff*: $(z : f`A) = (EX\ x{:}A.\ z = f\ x)$

by *blast*

lemma *image-subset-iff*: $(f`A \subseteq B) = (\forall x{\in}A.\ f\ x \in B)$
— This rewrite rule would confuse users if made default.
by *blast*

lemma *subset-image-iff*: $(B \subseteq f`A) = (EX\ AA.\ AA \subseteq A\ \&\ B = f`AA)$
apply *safe*
prefer *2* **apply** *fast*
apply (*rule-tac* $x = \{a.\ a : A\ \&\ f\ a : B\}$ **in** *exI*, *fast*)
done

lemma *image-subsetI*: $(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f`A \subseteq B$
— Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.
by *blast*

Range of a function – just a translation for image!

lemma *range-eqI*: $b = f\ x \Longrightarrow b \in range\ f$
by *simp*

lemma *rangeI*: $f\ x \in range\ f$
by *simp*

lemma *rangeE* [*elim?*]: $b \in range\ (\lambda x.\ f\ x) \Longrightarrow (!!x.\ b = f\ x \Longrightarrow P) \Longrightarrow P$
by *blast*

5.4.19 Set reasoning tools

Rewrite rules for boolean case-splitting: faster than *split-if* [*split*].

lemma *split-if-eq1*: $((if\ Q\ then\ x\ else\ y) = b) = ((Q \dashrightarrow x = b)\ \&\ (\sim Q \dashrightarrow y = b))$
by (*rule split-if*)

lemma *split-if-eq2*: $(a = (if\ Q\ then\ x\ else\ y)) = ((Q \dashrightarrow a = x)\ \&\ (\sim Q \dashrightarrow a = y))$
by (*rule split-if*)

Split ifs on either side of the membership relation. Not for [*simp*] – can cause goals to blow up!

lemma *split-if-mem1*: $((if\ Q\ then\ x\ else\ y) : b) = ((Q \dashrightarrow x : b)\ \&\ (\sim Q \dashrightarrow y : b))$
by (*rule split-if*)

lemma *split-if-mem2*: $(a : (if\ Q\ then\ x\ else\ y)) = ((Q \dashrightarrow a : x)\ \&\ (\sim Q \dashrightarrow a : y))$
by (*rule split-if* [**where** $P{=}\%S.\ a : S$])

lemmas *split-ifs = if-bool-eq-conj split-if-eq1 split-if-eq2 split-if-mem1 split-if-mem2*

ML ⟪
val mksimps-pairs = [(@{const-name Ball}, @{thms bspec})] @ mksimps-pairs;
⟫
declaration ⟪ *fn - =>*
Simplifier.map-ss (fn ss => ss setmksimps (mksimps mksimps-pairs))
⟫

5.4.20 The "proper subset" relation

lemma *psubsetI* [*intro*!,*noatp*]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$
by (*unfold less-le*) *blast*

lemma *psubsetE* [*elim*!,*noatp*]:
$[|A \subset B;\ [|A \subseteq B; \sim (B \subseteq A)|] \Longrightarrow R|] \Longrightarrow R$
by (*unfold less-le*) *blast*

lemma *psubset-insert-iff*:
$(A \subset \mathit{insert}\ x\ B) = (\mathit{if}\ x \in B\ \mathit{then}\ A \subset B\ \mathit{else}\ \mathit{if}\ x \in A\ \mathit{then}\ A - \{x\} \subset B\ \mathit{else}\ A \subseteq B)$
by (*auto simp add*: *less-le subset-insert-iff*)

lemma *psubset-eq*: $(A \subset B) = (A \subseteq B \ \&\ A \neq B)$
by (*simp only*: *less-le*)

lemma *psubset-imp-subset*: $A \subset B \Longrightarrow A \subseteq B$
by (*simp add*: *psubset-eq*)

lemma *psubset-trans*: $[|\ A \subset B;\ B \subset C\ |] \Longrightarrow A \subset C$
apply (*unfold less-le*)
apply (*auto dest*: *subset-antisym*)
done

lemma *psubsetD*: $[|\ A \subset B;\ c \in A\ |] \Longrightarrow c \in B$
apply (*unfold less-le*)
apply (*auto dest*: *subsetD*)
done

lemma *psubset-subset-trans*: $A \subset B \Longrightarrow B \subseteq C \Longrightarrow A \subset C$
by (*auto simp add*: *psubset-eq*)

lemma *subset-psubset-trans*: $A \subseteq B \Longrightarrow B \subset C \Longrightarrow A \subset C$
by (*auto simp add*: *psubset-eq*)

lemma *psubset-imp-ex-mem*: $A \subset B \Longrightarrow \exists\, b.\ b \in (B - A)$
by (*unfold less-le*) *blast*

lemma *atomize-ball*:
(!!x. $x \in A$ ==> $P\ x$) == *Trueprop* ($\forall x{\in}A.\ P\ x$)
by (*simp only*: *Ball-def atomize-all atomize-imp*)

lemmas [*symmetric*, *rulify*] = *atomize-ball*
and [*symmetric*, *defn*] = *atomize-ball*

5.5 Further set-theory lemmas

5.5.1 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq$ *insert a B*
by (*rule subsetI*) (*erule insertI2*)

lemma *subset-insertI2*: $A \subseteq B \Longrightarrow A \subseteq$ *insert b B*
by *blast*

lemma *subset-insert*: $x \notin A$ ==> ($A \subseteq$ *insert x B*) = ($A \subseteq B$)
by *blast*

Big Union – least upper bound of a set.

lemma *Union-upper*: $B \in A$ ==> $B \subseteq$ *Union A*
by (*iprover intro*: *subsetI UnionI*)

lemma *Union-least*: (!!X. $X \in A$ ==> $X \subseteq C$) ==> *Union A* $\subseteq C$
by (*iprover intro*: *subsetI elim*: *UnionE dest*: *subsetD*)

General union.

lemma *UN-upper*: $a \in A$ ==> $B\ a \subseteq (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *UN-least*: (!!x. $x \in A$ ==> $B\ x \subseteq C$) ==> $(\bigcup x{\in}A.\ B\ x) \subseteq C$
by (*iprover intro*: *subsetI elim*: *UN-E dest*: *subsetD*)

Big Intersection – greatest lower bound of a set.

lemma *Inter-lower*: $B \in A$ ==> *Inter A* $\subseteq B$
by *blast*

lemma *Inter-subset*:
[| !!X. $X \in A$ ==> $X \subseteq B$; $A \sim= \{\}$ |] ==> $\bigcap A \subseteq B$
by *blast*

lemma *Inter-greatest*: (!!X. $X \in A$ ==> $C \subseteq X$) ==> $C \subseteq$ *Inter A*
by (*iprover intro*: *InterI subsetI dest*: *subsetD*)

lemma *INT-lower*: $a \in A \Longrightarrow (\bigcap x{\in}A.\ B\ x) \subseteq B\ a$
by *blast*

lemma *INT-greatest*: $(!!x.\ x \in A \Longrightarrow C \subseteq B\ x) \Longrightarrow C \subseteq (\bigcap x{\in}A.\ B\ x)$
by (*iprover intro*: *INT-I subsetI dest*: *subsetD*)

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
by *blast*

lemma *Un-upper2*: $B \subseteq A \cup B$
by *blast*

lemma *Un-least*: $A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow A \cup B \subseteq C$
by *blast*

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
by *blast*

lemma *Int-lower2*: $A \cap B \subseteq B$
by *blast*

lemma *Int-greatest*: $C \subseteq A \Longrightarrow C \subseteq B \Longrightarrow C \subseteq A \cap B$
by *blast*

Set difference.

lemma *Diff-subset*: $A - B \subseteq A$
by *blast*

lemma *Diff-subset-conv*: $(A - B \subseteq C) = (A \subseteq B \cup C)$
by *blast*

5.5.2 Equalities involving union, intersection, inclusion, etc.

{}.

lemma *Collect-const* [*simp*]: $\{s.\ P\} = (\mathit{if}\ P\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{\})$
— supersedes *Collect-False-empty*
by *auto*

lemma *subset-empty* [*simp*]: $(A \subseteq \{\}) = (A = \{\})$
by *blast*

lemma *not-psubset-empty* [*iff*]: $\neg\ (A < \{\})$
by (*unfold less-le*) *blast*

lemma *Collect-empty-eq* [*simp*]: $(\mathit{Collect}\ P = \{\}) = (\forall x.\ \neg\ P\ x)$

by *blast*

lemma *empty-Collect-eq* [*simp*]: $(\{\} = \mathit{Collect}\ P) = (\forall x.\ \neg\ P\ x)$
by *blast*

lemma *Collect-neg-eq*: $\{x.\ \neg\ P\ x\} = -\ \{x.\ P\ x\}$
by *blast*

lemma *Collect-disj-eq*: $\{x.\ P\ x \mid Q\ x\} = \{x.\ P\ x\} \cup \{x.\ Q\ x\}$
by *blast*

lemma *Collect-imp-eq*: $\{x.\ P\ x \longrightarrow Q\ x\} = -\{x.\ P\ x\} \cup \{x.\ Q\ x\}$
by *blast*

lemma *Collect-conj-eq*: $\{x.\ P\ x\ \&\ Q\ x\} = \{x.\ P\ x\} \cap \{x.\ Q\ x\}$
by *blast*

lemma *Collect-all-eq*: $\{x.\ \forall y.\ P\ x\ y\} = (\bigcap y.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-ball-eq*: $\{x.\ \forall y{\in}A.\ P\ x\ y\} = (\bigcap y{\in}A.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-ex-eq* [*noatp*]: $\{x.\ \exists y.\ P\ x\ y\} = (\bigcup y.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-bex-eq* [*noatp*]: $\{x.\ \exists y{\in}A.\ P\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ P\ x\ y\})$
by *blast*

insert.

lemma *insert-is-Un*: *insert a A* = $\{a\}$ *Un A*
— NOT SUITABLE FOR REWRITING since $\{a\}$ == *insert a* $\{\}$
by *blast*

lemma *insert-not-empty* [*simp*]: *insert a A* $\neq \{\}$
by *blast*

lemmas *empty-not-insert* = *insert-not-empty* [*symmetric*, *standard*]
declare *empty-not-insert* [*simp*]

lemma *insert-absorb*: $a \in A$ ==> *insert a A* = *A*
— [*simp*] causes recursive calls when there are nested inserts
— with *quadratic* running time
by *blast*

lemma *insert-absorb2* [*simp*]: *insert x* (*insert x A*) = *insert x A*
by *blast*

lemma *insert-commute*: *insert x* (*insert y A*) = *insert y* (*insert x A*)

by *blast*

lemma *insert-subset* [*simp*]: (*insert x A* ⊆ *B*) = (*x* ∈ *B* & *A* ⊆ *B*)
by *blast*

lemma *mk-disjoint-insert*: *a* ∈ *A* ==> ∃ *B*. *A* = *insert a B* & *a* ∉ *B*
— use new *B* rather than *A* − {*a*} to avoid infinite unfolding
apply (*rule-tac x* = *A* − {*a*} **in** *exI*, *blast*)
done

lemma *insert-Collect*: *insert a* (*Collect P*) = {*u*. *u* ≠ *a* −−> *P u*}
by *auto*

lemma *UN-insert-distrib*: *u* ∈ *A* ==> (⋃ *x*∈*A*. *insert a* (*B x*)) = *insert a* (⋃ *x*∈*A*. *B x*)
by *blast*

lemma *insert-inter-insert*[*simp*]: *insert a A* ∩ *insert a B* = *insert a* (*A* ∩ *B*)
by *blast*

lemma *insert-disjoint* [*simp*,*noatp*]:
(*insert a A* ∩ *B* = {}) = (*a* ∉ *B* ∧ *A* ∩ *B* = {})
({} = *insert a A* ∩ *B*) = (*a* ∉ *B* ∧ {} = *A* ∩ *B*)
by *auto*

lemma *disjoint-insert* [*simp*,*noatp*]:
(*B* ∩ *insert a A* = {}) = (*a* ∉ *B* ∧ *B* ∩ *A* = {})
({} = *A* ∩ *insert b B*) = (*b* ∉ *A* ∧ {} = *A* ∩ *B*)
by *auto*

image.

lemma *image-empty* [*simp*]: *f*‘{} = {}
by *blast*

lemma *image-insert* [*simp*]: *f* ‘ *insert a B* = *insert* (*f a*) (*f*‘*B*)
by *blast*

lemma *image-constant*: *x* ∈ *A* ==> (λ*x*. *c*) ‘ *A* = {*c*}
by *auto*

lemma *image-constant-conv*: (%*x*. *c*) ‘ *A* = (*if A* = {} *then* {} *else* {*c*})
by *auto*

lemma *image-image*: *f* ‘ (*g* ‘ *A*) = (λ*x*. *f* (*g x*)) ‘ *A*
by *blast*

lemma *insert-image* [*simp*]: *x* ∈ *A* ==> *insert* (*f x*) (*f*‘*A*) = *f*‘*A*
by *blast*

lemma *image-is-empty* [*iff*]: $(f`A = \{\}) = (A = \{\})$
by *blast*

lemma *image-Collect* [*noatp*]: $f ` \{x.\ P\ x\} = \{f\ x \mid x.\ P\ x\}$
— NOT suitable as a default simprule: the RHS isn't simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.
by *blast*

lemma *if-image-distrib* [*simp*]:
$(\lambda x.\ if\ P\ x\ then\ f\ x\ else\ g\ x) ` S$
$= (f ` (S \cap \{x.\ P\ x\})) \cup (g ` (S \cap \{x.\ \neg\ P\ x\}))$
by (*auto simp add*: *image-def*)

lemma *image-cong*: $M = N \Longrightarrow (!!x.\ x \in N \Longrightarrow f\ x = g\ x) \Longrightarrow f`M = g`N$
by (*simp add*: *image-def*)

range.

lemma *full-SetCompr-eq* [*noatp*]: $\{u.\ \exists x.\ u = f\ x\} = range\ f$
by *auto*

lemma *range-composition*: $range\ (\lambda x.\ f\ (g\ x)) = f`range\ g$
by (*subst image-image*, *simp*)

Int

lemma *Int-absorb* [*simp*]: $A \cap A = A$
by *blast*

lemma *Int-left-absorb*: $A \cap (A \cap B) = A \cap B$
by *blast*

lemma *Int-commute*: $A \cap B = B \cap A$
by *blast*

lemma *Int-left-commute*: $A \cap (B \cap C) = B \cap (A \cap C)$
by *blast*

lemma *Int-assoc*: $(A \cap B) \cap C = A \cap (B \cap C)$
by *blast*

lemmas *Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute*
— Intersection is an AC-operator

lemma *Int-absorb1*: $B \subseteq A \Longrightarrow A \cap B = B$
by *blast*

lemma *Int-absorb2*: $A \subseteq B \Longrightarrow A \cap B = A$

by *blast*

lemma *Int-empty-left* [*simp*]: $\{\} \cap B = \{\}$
by *blast*

lemma *Int-empty-right* [*simp*]: $A \cap \{\} = \{\}$
by *blast*

lemma *disjoint-eq-subset-Compl*: $(A \cap B = \{\}) = (A \subseteq -B)$
by *blast*

lemma *disjoint-iff-not-equal*: $(A \cap B = \{\}) = (\forall x{\in}A.\ \forall y{\in}B.\ x \neq y)$
by *blast*

lemma *Int-UNIV-left* [*simp*]: $UNIV \cap B = B$
by *blast*

lemma *Int-UNIV-right* [*simp*]: $A \cap UNIV = A$
by *blast*

lemma *Int-eq-Inter*: $A \cap B = \bigcap\{A, B\}$
by *blast*

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by *blast*

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
by *blast*

lemma *Int-UNIV* [*simp*,*noatp*]: $(A \cap B = UNIV) = (A = UNIV\ \&\ B = UNIV)$
by *blast*

lemma *Int-subset-iff* [*simp*]: $(C \subseteq A \cap B) = (C \subseteq A\ \&\ C \subseteq B)$
by *blast*

lemma *Int-Collect*: $(x \in A \cap \{x.\ P\ x\}) = (x \in A\ \&\ P\ x)$
by *blast*

Un.

lemma *Un-absorb* [*simp*]: $A \cup A = A$
by *blast*

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
by *blast*

lemma *Un-commute*: $A \cup B = B \cup A$
by *blast*

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$

by *blast*

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
by *blast*

lemmas *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*
— Union is an AC-operator

lemma *Un-absorb1*: $A \subseteq B \Longrightarrow A \cup B = B$
by *blast*

lemma *Un-absorb2*: $B \subseteq A \Longrightarrow A \cup B = A$
by *blast*

lemma *Un-empty-left* [*simp*]: $\{\} \cup B = B$
by *blast*

lemma *Un-empty-right* [*simp*]: $A \cup \{\} = A$
by *blast*

lemma *Un-UNIV-left* [*simp*]: $UNIV \cup B = UNIV$
by *blast*

lemma *Un-UNIV-right* [*simp*]: $A \cup UNIV = UNIV$
by *blast*

lemma *Un-eq-Union*: $A \cup B = \bigcup \{A, B\}$
by *blast*

lemma *Un-insert-left* [*simp*]: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
by *blast*

lemma *Un-insert-right* [*simp*]: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
by *blast*

lemma *Int-insert-left*:
$(insert\ a\ B)\ Int\ C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$
by *auto*

lemma *Int-insert-right*:
$A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
by *auto*

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
by *blast*

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
by *blast*

lemma *Un-Int-crazy*:
$(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
by *blast*

lemma *subset-Un-eq*: $(A \subseteq B) = (A \cup B = B)$
by *blast*

lemma *Un-empty* [*iff*]: $(A \cup B = \{\}) = (A = \{\} \ \& \ B = \{\})$
by *blast*

lemma *Un-subset-iff* [*simp*]: $(A \cup B \subseteq C) = (A \subseteq C \ \& \ B \subseteq C)$
by *blast*

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
by *blast*

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
by *blast*

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap -A = \{\}$
by *blast*

lemma *Compl-disjoint2* [*simp*]: $-A \cap A = \{\}$
by *blast*

lemma *Compl-partition*: $A \cup -A = UNIV$
by *blast*

lemma *Compl-partition2*: $-A \cup A = UNIV$
by *blast*

lemma *double-complement* [*simp*]: $-(-A) = (A::'a\ set)$
by *blast*

lemma *Compl-Un* [*simp*]: $-(A \cup B) = (-A) \cap (-B)$
by *blast*

lemma *Compl-Int* [*simp*]: $-(A \cap B) = (-A) \cup (-B)$
by *blast*

lemma *Compl-UN* [*simp*]: $-(\bigcup x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ -B\ x)$
by *blast*

lemma *Compl-INT* [*simp*]: $-(\bigcap x{\in}A.\ B\ x) = (\bigcup x{\in}A.\ -B\ x)$
by *blast*

lemma *subset-Compl-self-eq*: $(A \subseteq -A) = (A = \{\})$
by *blast*

lemma *Un-Int-assoc-eq*: $((A \cap B) \cup C = A \cap (B \cup C)) = (C \subseteq A)$
— Halmos, Naive Set Theory, page 16.
by *blast*

lemma *Compl-UNIV-eq* [*simp*]: $-UNIV = \{\}$
by *blast*

lemma *Compl-empty-eq* [*simp*]: $-\{\} = UNIV$
by *blast*

lemma *Compl-subset-Compl-iff* [*iff*]: $(-A \subseteq -B) = (B \subseteq A)$
by *blast*

lemma *Compl-eq-Compl-iff* [*iff*]: $(-A = -B) = (A = (B::'a\ set))$
by *blast*

Union.

lemma *Union-empty* [*simp*]: $Union(\{\}) = \{\}$
by *blast*

lemma *Union-UNIV* [*simp*]: $Union\ UNIV = UNIV$
by *blast*

lemma *Union-insert* [*simp*]: $Union\ (insert\ a\ B) = a \cup \bigcup B$
by *blast*

lemma *Union-Un-distrib* [*simp*]: $\bigcup(A\ Un\ B) = \bigcup A \cup \bigcup B$
by *blast*

lemma *Union-Int-subset*: $\bigcup(A \cap B) \subseteq \bigcup A \cap \bigcup B$
by *blast*

lemma *Union-empty-conv* [*simp*,*noatp*]: $(\bigcup A = \{\}) = (\forall x{\in}A.\ x = \{\})$
by *blast*

lemma *empty-Union-conv* [*simp*,*noatp*]: $(\{\} = \bigcup A) = (\forall x{\in}A.\ x = \{\})$
by *blast*

lemma *Union-disjoint*: $(\bigcup C \cap A = \{\}) = (\forall B{\in}C.\ B \cap A = \{\})$
by *blast*

Inter.

lemma *Inter-empty* [*simp*]: $\bigcap\{\} = UNIV$
by *blast*

lemma *Inter-UNIV* [*simp*]: $\bigcap UNIV = \{\}$
by *blast*

lemma *Inter-insert* [*simp*]: $\bigcap(\mathit{insert}\ a\ B) = a \cap \bigcap B$
by *blast*

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap(A \cap B)$
by *blast*

lemma *Inter-Un-distrib*: $\bigcap(A \cup B) = \bigcap A \cap \bigcap B$
by *blast*

lemma *Inter-UNIV-conv* [*simp*,*noatp*]:
$(\bigcap A = \mathit{UNIV}) = (\forall x{\in}A.\ x = \mathit{UNIV})$
$(\mathit{UNIV} = \bigcap A) = (\forall x{\in}A.\ x = \mathit{UNIV})$
by *blast+*

UN and *INT*.

Basic identities:

lemma *UN-empty* [*simp*,*noatp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
by *blast*

lemma *UN-empty2* [*simp*]: $(\bigcup x{\in}A.\ \{\}) = \{\}$
by *blast*

lemma *UN-singleton* [*simp*]: $(\bigcup x{\in}A.\ \{x\}) = A$
by *blast*

lemma *UN-absorb*: $k \in I \Longrightarrow A\ k \cup (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ A\ i)$
by *auto*

lemma *INT-empty* [*simp*]: $(\bigcap x{\in}\{\}.\ B\ x) = \mathit{UNIV}$
by *blast*

lemma *INT-absorb*: $k \in I \Longrightarrow A\ k \cap (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ A\ i)$
by *blast*

lemma *UN-insert* [*simp*]: $(\bigcup x{\in}\mathit{insert}\ a\ A.\ B\ x) = B\ a \cup \mathit{UNION}\ A\ B$
by *blast*

lemma *UN-Un*[*simp*]: $(\bigcup i \in A \cup B.\ M\ i) = (\bigcup i{\in}A.\ M\ i) \cup (\bigcup i{\in}B.\ M\ i)$
by *blast*

lemma *UN-UN-flatten*: $(\bigcup x \in (\bigcup y{\in}A.\ B\ y).\ C\ x) = (\bigcup y{\in}A.\ \bigcup x{\in}B\ y.\ C\ x)$
by *blast*

lemma *UN-subset-iff*: $((\bigcup i{\in}I.\ A\ i) \subseteq B) = (\forall i{\in}I.\ A\ i \subseteq B)$
by *blast*

lemma *INT-subset-iff*: $(B \subseteq (\bigcap i{\in}I.\ A\ i)) = (\forall i{\in}I.\ B \subseteq A\ i)$
by *blast*

lemma *INT-insert* [*simp*]: $(\bigcap x \in insert\ a\ A.\ B\ x) = B\ a \cap INTER\ A\ B$
by *blast*

lemma *INT-Un*: $(\bigcap i \in A \cup B.\ M\ i) = (\bigcap i \in A.\ M\ i) \cap (\bigcap i{\in}B.\ M\ i)$
by *blast*

lemma *INT-insert-distrib*:
$u \in A \Longrightarrow (\bigcap x{\in}A.\ insert\ a\ (B\ x)) = insert\ a\ (\bigcap x{\in}A.\ B\ x)$
by *blast*

lemma *Union-image-eq* [*simp*]: $\bigcup(B`A) = (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *image-Union*: $f\ `\ \bigcup S = (\bigcup x{\in}S.\ f\ `\ x)$
by *blast*

lemma *Inter-image-eq* [*simp*]: $\bigcap(B`A) = (\bigcap x{\in}A.\ B\ x)$
by *blast*

lemma *UN-constant* [*simp*]: $(\bigcup y{\in}A.\ c) = (if\ A = \{\}\ then\ \{\}\ else\ c)$
by *auto*

lemma *INT-constant* [*simp*]: $(\bigcap y{\in}A.\ c) = (if\ A = \{\}\ then\ UNIV\ else\ c)$
by *auto*

lemma *UN-eq*: $(\bigcup x{\in}A.\ B\ x) = \bigcup(\{Y.\ \exists x{\in}A.\ Y = B\ x\})$
by *blast*

lemma *INT-eq*: $(\bigcap x{\in}A.\ B\ x) = \bigcap(\{Y.\ \exists x{\in}A.\ Y = B\ x\})$
— Look: it has an *existential* quantifier
by *blast*

lemma *UNION-empty-conv*[*simp*]:
$(\{\} = (UN\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = \{\})$
$((UN\ x{:}A.\ B\ x) = \{\}) = (\forall x{\in}A.\ B\ x = \{\})$
by *blast+*

lemma *INTER-UNIV-conv*[*simp*]:
$(UNIV = (INT\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = UNIV)$
$((INT\ x{:}A.\ B\ x) = UNIV) = (\forall x{\in}A.\ B\ x = UNIV)$
by *blast+*

Distributive laws:

lemma *Int-Union*: $A \cap \bigcup B = (\bigcup C{\in}B.\ A \cap C)$
by *blast*

lemma *Int-Union2*: $\bigcup B \cap A = (\bigcup C{\in}B.\ C \cap A)$
by *blast*

lemma *Un-Union-image*: $(\bigcup x{\in}C.\ A\ x \cup B\ x) = \bigcup(A`C) \cup \bigcup(B`C)$
— Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
— Union of a family of unions
by *blast*

lemma *UN-Un-distrib*: $(\bigcup i{\in}I.\ A\ i \cup B\ i) = (\bigcup i{\in}I.\ A\ i) \cup (\bigcup i{\in}I.\ B\ i)$
— Equivalent version
by *blast*

lemma *Un-Inter*: $A \cup \bigcap B = (\bigcap C{\in}B.\ A \cup C)$
by *blast*

lemma *Int-Inter-image*: $(\bigcap x{\in}C.\ A\ x \cap B\ x) = \bigcap(A`C) \cap \bigcap(B`C)$
by *blast*

lemma *INT-Int-distrib*: $(\bigcap i{\in}I.\ A\ i \cap B\ i) = (\bigcap i{\in}I.\ A\ i) \cap (\bigcap i{\in}I.\ B\ i)$
— Equivalent version
by *blast*

lemma *Int-UN-distrib*: $B \cap (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ B \cap A\ i)$
— Halmos, Naive Set Theory, page 35.
by *blast*

lemma *Un-INT-distrib*: $B \cup (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ B \cup A\ i)$
by *blast*

lemma *Int-UN-distrib2*: $(\bigcup i{\in}I.\ A\ i) \cap (\bigcup j{\in}J.\ B\ j) = (\bigcup i{\in}I.\ \bigcup j{\in}J.\ A\ i \cap B\ j)$
by *blast*

lemma *Un-INT-distrib2*: $(\bigcap i{\in}I.\ A\ i) \cup (\bigcap j{\in}J.\ B\ j) = (\bigcap i{\in}I.\ \bigcap j{\in}J.\ A\ i \cup B\ j)$
by *blast*

Bounded quantifiers.

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

lemma *ball-Un*: $(\forall x \in A \cup B.\ P\ x) = ((\forall x{\in}A.\ P\ x)\ \&\ (\forall x{\in}B.\ P\ x))$
by *blast*

lemma *bex-Un*: $(\exists x \in A \cup B.\ P\ x) = ((\exists x{\in}A.\ P\ x)\ |\ (\exists x{\in}B.\ P\ x))$
by *blast*

lemma *ball-UN*: $(\forall z \in \mathit{UNION}\ A\ B.\ P\ z) = (\forall x{\in}A.\ \forall z \in B\ x.\ P\ z)$
by *blast*

lemma *bex-UN*: $(\exists z \in \mathit{UNION}\ A\ B.\ P\ z) = (\exists x{\in}A.\ \exists z{\in}B\ x.\ P\ z)$

by *blast*

Set difference.

lemma *Diff-eq*: $A - B = A \cap (-B)$
by *blast*

lemma *Diff-eq-empty-iff* [*simp*,*noatp*]: $(A - B = \{\}) = (A \subseteq B)$
by *blast*

lemma *Diff-cancel* [*simp*]: $A - A = \{\}$
by *blast*

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - (B::'a\ set)$
by *blast*

lemma *Diff-triv*: $A \cap B = \{\} \Longrightarrow A - B = A$
by (*blast elim*: *equalityE*)

lemma *empty-Diff* [*simp*]: $\{\} - A = \{\}$
by *blast*

lemma *Diff-empty* [*simp*]: $A - \{\} = A$
by *blast*

lemma *Diff-UNIV* [*simp*]: $A - UNIV = \{\}$
by *blast*

lemma *Diff-insert0* [*simp*,*noatp*]: $x \notin A \Longrightarrow A - insert\ x\ B = A - B$
by *blast*

lemma *Diff-insert*: $A - insert\ a\ B = A - B - \{a\}$
— NOT SUITABLE FOR REWRITING since $\{a\} == insert\ a\ 0$
by *blast*

lemma *Diff-insert2*: $A - insert\ a\ B = A - \{a\} - B$
— NOT SUITABLE FOR REWRITING since $\{a\} == insert\ a\ 0$
by *blast*

lemma *insert-Diff-if*: $insert\ x\ A - B = (if\ x \in B\ then\ A - B\ else\ insert\ x\ (A - B))$
by *auto*

lemma *insert-Diff1* [*simp*]: $x \in B \Longrightarrow insert\ x\ A - B = A - B$
by *blast*

lemma *insert-Diff-single*[*simp*]: $insert\ a\ (A - \{a\}) = insert\ a\ A$
by *blast*

lemma *insert-Diff*: $a \in A \Longrightarrow insert\ a\ (A - \{a\}) = A$

by *blast*

lemma *Diff-insert-absorb*: $x \notin A \Longrightarrow (insert\ x\ A) - \{x\} = A$
by *auto*

lemma *Diff-disjoint* [*simp*]: $A \cap (B - A) = \{\}$
by *blast*

lemma *Diff-partition*: $A \subseteq B \Longrightarrow A \cup (B - A) = B$
by *blast*

lemma *double-diff*: $A \subseteq B \Longrightarrow B \subseteq C \Longrightarrow B - (C - A) = A$
by *blast*

lemma *Un-Diff-cancel* [*simp*]: $A \cup (B - A) = A \cup B$
by *blast*

lemma *Un-Diff-cancel2* [*simp*]: $(B - A) \cup A = B \cup A$
by *blast*

lemma *Diff-Un*: $A - (B \cup C) = (A - B) \cap (A - C)$
by *blast*

lemma *Diff-Int*: $A - (B \cap C) = (A - B) \cup (A - C)$
by *blast*

lemma *Un-Diff*: $(A \cup B) - C = (A - C) \cup (B - C)$
by *blast*

lemma *Int-Diff*: $(A \cap B) - C = A \cap (B - C)$
by *blast*

lemma *Diff-Int-distrib*: $C \cap (A - B) = (C \cap A) - (C \cap B)$
by *blast*

lemma *Diff-Int-distrib2*: $(A - B) \cap C = (A \cap C) - (B \cap C)$
by *blast*

lemma *Diff-Compl* [*simp*]: $A - (-B) = A \cap B$
by *auto*

lemma *Compl-Diff-eq* [*simp*]: $-(A - B) = -A \cup B$
by *blast*

Quantification over type *bool*.

lemma *bool-induct*: $P\ True \Longrightarrow P\ False \Longrightarrow P\ x$
by (*cases x*) *auto*

lemma *all-bool-eq*: $(\forall b.\ P\ b) \longleftrightarrow P\ True \wedge P\ False$

by (*auto intro*: *bool-induct*)

lemma *bool-contrapos*: $P\ x \Longrightarrow \neg\ P\ False \Longrightarrow P\ True$
by (*cases x*) *auto*

lemma *ex-bool-eq*: $(\exists\, b.\ P\ b) \longleftrightarrow P\ True \vee P\ False$
by (*auto intro*: *bool-contrapos*)

lemma *Un-eq-UN*: $A \cup B = (\bigcup b.\ if\ b\ then\ A\ else\ B)$
by (*auto simp add*: *split-if-mem2*)

lemma *UN-bool-eq*: $(\bigcup b{::}bool.\ A\ b) = (A\ True \cup A\ False)$
by (*auto intro*: *bool-contrapos*)

lemma *INT-bool-eq*: $(\bigcap b{::}bool.\ A\ b) = (A\ True \cap A\ False)$
by (*auto intro*: *bool-induct*)

Pow

lemma *Pow-empty* [*simp*]: $Pow\ \{\} = \{\{\}\}$
by (*auto simp add*: *Pow-def*)

lemma *Pow-insert*: $Pow\ (insert\ a\ A) = Pow\ A \cup (insert\ a\ `\ Pow\ A)$
by (*blast intro*: *image-eqI* [**where** *?x* $= u - \{a\}$, *standard*])

lemma *Pow-Compl*: $Pow\ (-\ A) = \{-B \mid B.\ A \in Pow\ B\}$
by (*blast intro*: *exI* [**where** *?x* $= -\ u$, *standard*])

lemma *Pow-UNIV* [*simp*]: $Pow\ UNIV = UNIV$
by *blast*

lemma *Un-Pow-subset*: $Pow\ A \cup Pow\ B \subseteq Pow\ (A \cup B)$
by *blast*

lemma *UN-Pow-subset*: $(\bigcup x{\in}A.\ Pow\ (B\ x)) \subseteq Pow\ (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *subset-Pow-Union*: $A \subseteq Pow\ (\bigcup A)$
by *blast*

lemma *Union-Pow-eq* [*simp*]: $\bigcup(Pow\ A) = A$
by *blast*

lemma *Pow-Int-eq* [*simp*]: $Pow\ (A \cap B) = Pow\ A \cap Pow\ B$
by *blast*

lemma *Pow-INT-eq*: $Pow\ (\bigcap x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ Pow\ (B\ x))$
by *blast*

Miscellany.

lemma *set-eq-subset*: $(A = B) = (A \subseteq B \ \& \ B \subseteq A)$
by *blast*

lemma *subset-iff*: $(A \subseteq B) = (\forall t.\ t \in A \longrightarrow t \in B)$
by *blast*

lemma *subset-iff-psubset-eq*: $(A \subseteq B) = ((A \subset B) \mid (A = B))$
by (*unfold less-le*) *blast*

lemma *all-not-in-conv* [*simp*]: $(\forall x.\ x \notin A) = (A = \{\})$
by *blast*

lemma *ex-in-conv*: $(\exists x.\ x \in A) = (A \neq \{\})$
by *blast*

lemma *distinct-lemma*: $f\ x \neq f\ y \Longrightarrow x \neq y$
by *iprover*

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma *UN-simps* [*simp*]:
$!!a\ B\ C.\ (UN\ x{:}C.\ insert\ a\ (B\ x)) = (if\ C{=}\{\}\ then\ \{\}\ else\ insert\ a\ (UN\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Un\ B) = ((if\ C{=}\{\}\ then\ \{\}\ else\ (UN\ x{:}C.\ A\ x)\ Un\ B))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ Un\ B\ x) = ((if\ C{=}\{\}\ then\ \{\}\ else\ A\ Un\ (UN\ x{:}C.\ B\ x)))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Int\ B) = ((UN\ x{:}C.\ A\ x)\ Int\ B)$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ Int\ B\ x) = (A\ Int\ (UN\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x - B) = ((UN\ x{:}C.\ A\ x) - B)$
$!!A\ B\ C.\ (UN\ x{:}C.\ A - B\ x) = (A - (INT\ x{:}C.\ B\ x))$
$!!A\ B.\ (UN\ x{:}\ Union\ A.\ B\ x) = (UN\ y{:}A.\ UN\ x{:}y.\ B\ x)$
$!!A\ B\ C.\ (UN\ z{:}\ UNION\ A\ B.\ C\ z) = (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z)$
$!!A\ B\ f.\ (UN\ x{:}f`A.\ B\ x) = (UN\ a{:}A.\ B\ (f\ a))$
by *auto*

lemma *INT-simps* [*simp*]:
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Int\ B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x)\ Int\ B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ Int\ B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A\ Int\ (INT\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x - B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x) - B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A - B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A - (UN\ x{:}C.\ B\ x))$
$!!a\ B\ C.\ (INT\ x{:}C.\ insert\ a\ (B\ x)) = insert\ a\ (INT\ x{:}C.\ B\ x)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Un\ B) = ((INT\ x{:}C.\ A\ x)\ Un\ B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ Un\ B\ x) = (A\ Un\ (INT\ x{:}C.\ B\ x))$
$!!A\ B.\ (INT\ x{:}\ Union\ A.\ B\ x) = (INT\ y{:}A.\ INT\ x{:}y.\ B\ x)$
$!!A\ B\ C.\ (INT\ z{:}\ UNION\ A\ B.\ C\ z) = (INT\ x{:}A.\ INT\ z{:}\ B(x).\ C\ z)$

!!$A\ B\ f.\ (INT\ x{:}f`A.\ B\ x) \quad = (INT\ a{:}A.\ B\ (f\ a))$
by *auto*

lemma *ball-simps* [*simp*,*noatp*]:
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x \mid Q) = ((ALL\ x{:}A.\ P\ x) \mid Q)$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P \mid Q\ x) = (P \mid (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P \dashrightarrow Q\ x) = (P \dashrightarrow (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x \dashrightarrow Q) = ((EX\ x{:}A.\ P\ x) \dashrightarrow Q)$
!!$P.\ (ALL\ x{:}\{\}.\ P\ x) = True$
!!$P.\ (ALL\ x{:}UNIV.\ P\ x) = (ALL\ x.\ P\ x)$
!!$a\ B\ P.\ (ALL\ x{:}insert\ a\ B.\ P\ x) = (P\ a\ \&\ (ALL\ x{:}B.\ P\ x))$
!!$A\ P.\ (ALL\ x{:}Union\ A.\ P\ x) = (ALL\ y{:}A.\ ALL\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (ALL\ x{:}\ UNION\ A\ B.\ P\ x) = (ALL\ a{:}A.\ ALL\ x{:}\ B\ a.\ P\ x)$
!!$P\ Q.\ (ALL\ x{:}Collect\ Q.\ P\ x) = (ALL\ x.\ Q\ x \dashrightarrow P\ x)$
!!$A\ P\ f.\ (ALL\ x{:}f`A.\ P\ x) = (ALL\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(ALL\ x{:}A.\ P\ x)) = (EX\ x{:}A.\ \sim P\ x)$
by *auto*

lemma *bex-simps* [*simp*,*noatp*]:
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ x\ \&\ Q) = ((EX\ x{:}A.\ P\ x)\ \&\ Q)$
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x{:}A.\ Q\ x))$
!!$P.\ (EX\ x{:}\{\}.\ P\ x) = False$
!!$P.\ (EX\ x{:}UNIV.\ P\ x) = (EX\ x.\ P\ x)$
!!$a\ B\ P.\ (EX\ x{:}insert\ a\ B.\ P\ x) = (P(a) \mid (EX\ x{:}B.\ P\ x))$
!!$A\ P.\ (EX\ x{:}Union\ A.\ P\ x) = (EX\ y{:}A.\ EX\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (EX\ x{:}\ UNION\ A\ B.\ P\ x) = (EX\ a{:}A.\ EX\ x{:}B\ a.\ P\ x)$
!!$P\ Q.\ (EX\ x{:}Collect\ Q.\ P\ x) = (EX\ x.\ Q\ x\ \&\ P\ x)$
!!$A\ P\ f.\ (EX\ x{:}f`A.\ P\ x) = (EX\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(EX\ x{:}A.\ P\ x)) = (ALL\ x{:}A.\ \sim P\ x)$
by *auto*

lemma *ball-conj-distrib*:
$(ALL\ x{:}A.\ P\ x\ \&\ Q\ x) = ((ALL\ x{:}A.\ P\ x)\ \&\ (ALL\ x{:}A.\ Q\ x))$
by *blast*

lemma *bex-disj-distrib*:
$(EX\ x{:}A.\ P\ x \mid Q\ x) = ((EX\ x{:}A.\ P\ x) \mid (EX\ x{:}A.\ Q\ x))$
by *blast*

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:
!!$a\ B\ C.\ insert\ a\ (UN\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ \{a\}\ else\ (UN\ x{:}C.\ insert\ a\ (B\ x)))$
!!$A\ B\ C.\ (UN\ x{:}C.\ A\ x)\ Un\ B \quad = (if\ C{=}\{\}\ then\ B\ else\ (UN\ x{:}C.\ A\ x\ Un\ B))$
!!$A\ B\ C.\ A\ Un\ (UN\ x{:}C.\ B\ x) \quad = (if\ C{=}\{\}\ then\ A\ else\ (UN\ x{:}C.\ A\ Un\ B\ x))$
!!$A\ B\ C.\ ((UN\ x{:}C.\ A\ x)\ Int\ B) = (UN\ x{:}C.\ A\ x\ Int\ B)$
!!$A\ B\ C.\ (A\ Int\ (UN\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A\ Int\ B\ x)$
!!$A\ B\ C.\ ((UN\ x{:}C.\ A\ x) - B) = (UN\ x{:}C.\ A\ x - B)$
!!$A\ B\ C.\ (A - (INT\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A - B\ x)$

!!$A\ B.\ (UN\ y{:}A.\ UN\ x{:}y.\ B\ x) = (UN\ x{:}\ Union\ A.\ B\ x)$
!!$A\ B\ C.\ (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z) = (UN\ z{:}\ UNION\ A\ B.\ C\ z)$
!!$A\ B\ f.\ (UN\ a{:}A.\ B\ (f\ a)) = (UN\ x{:}f`A.\ B\ x)$
by *auto*

lemma *INT-extend-simps*:
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x)\ Int\ B = (if\ C{=}\{\}\ then\ B\ else\ (INT\ x{:}C.\ A\ x\ Int\ B))$
!!$A\ B\ C.\ A\ Int\ (INT\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (INT\ x{:}C.\ A\ Int\ B\ x))$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x) - B\ \ = (if\ C{=}\{\}\ then\ UNIV{-}B\ else\ (INT\ x{:}C.\ A\ x - B))$
!!$A\ B\ C.\ A - (UN\ x{:}C.\ B\ x)\ \ = (if\ C{=}\{\}\ then\ A\ else\ (INT\ x{:}C.\ A - B\ x))$
!!$a\ B\ C.\ insert\ a\ (INT\ x{:}C.\ B\ x) = (INT\ x{:}C.\ insert\ a\ (B\ x))$
!!$A\ B\ C.\ ((INT\ x{:}C.\ A\ x)\ Un\ B)\ = (INT\ x{:}C.\ A\ x\ Un\ B)$
!!$A\ B\ C.\ A\ Un\ (INT\ x{:}C.\ B\ x)\ = (INT\ x{:}C.\ A\ Un\ B\ x)$
!!$A\ B.\ (INT\ y{:}A.\ INT\ x{:}y.\ B\ x) = (INT\ x{:}\ Union\ A.\ B\ x)$
!!$A\ B\ C.\ (INT\ x{:}A.\ INT\ z{:}\ B(x).\ C\ z) = (INT\ z{:}\ UNION\ A\ B.\ C\ z)$
!!$A\ B\ f.\ (INT\ a{:}A.\ B\ (f\ a))\ \ = (INT\ x{:}f`A.\ B\ x)$
by *auto*

5.5.3 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \Longrightarrow f`A \subseteq f`B$
by *blast*

lemma *Pow-mono*: $A \subseteq B \Longrightarrow Pow\ A \subseteq Pow\ B$
by *blast*

lemma *Union-mono*: $A \subseteq B \Longrightarrow \bigcup A \subseteq \bigcup B$
by *blast*

lemma *Inter-anti-mono*: $B \subseteq A \Longrightarrow \bigcap A \subseteq \bigcap B$
by *blast*

lemma *UN-mono*:
$A \subseteq B \Longrightarrow (!!x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow$
$(\bigcup x{\in}A.\ f\ x) \subseteq (\bigcup x{\in}B.\ g\ x)$
by (*blast dest*: *subsetD*)

lemma *INT-anti-mono*:
$B \subseteq A \Longrightarrow (!!x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow$
$(\bigcap x{\in}A.\ f\ x) \subseteq (\bigcap x{\in}A.\ g\ x)$
— The last inclusion is POSITIVE!
by (*blast dest*: *subsetD*)

lemma *insert-mono*: $C \subseteq D \Longrightarrow insert\ a\ C \subseteq insert\ a\ D$
by *blast*

lemma *Un-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cup B \subseteq C \cup D$
by *blast*

lemma *Int-mono*: $A \subseteq C ==> B \subseteq D ==> A \cap B \subseteq C \cap D$
by *blast*

lemma *Diff-mono*: $A \subseteq C ==> D \subseteq B ==> A - B \subseteq C - D$
by *blast*

lemma *Compl-anti-mono*: $A \subseteq B ==> -B \subseteq -A$
by *blast*

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B ==> x \in A --> x \in B$
apply (*rule impI*)
apply (*erule subsetD*, *assumption*)
done

lemma *conj-mono*: $P1 --> Q1 ==> P2 --> Q2 ==> (P1 \& P2) --> (Q1 \& Q2)$
by *iprover*

lemma *disj-mono*: $P1 --> Q1 ==> P2 --> Q2 ==> (P1 \mid P2) --> (Q1 \mid Q2)$
by *iprover*

lemma *imp-mono*: $Q1 --> P1 ==> P2 --> Q2 ==> (P1 --> P2) --> (Q1 --> Q2)$
by *iprover*

lemma *imp-refl*: $P --> P$ **..**

lemma *ex-mono*: $(!!x.\ P\ x --> Q\ x) ==> (EX\ x.\ P\ x) --> (EX\ x.\ Q\ x)$
by *iprover*

lemma *all-mono*: $(!!x.\ P\ x --> Q\ x) ==> (ALL\ x.\ P\ x) --> (ALL\ x.\ Q\ x)$
by *iprover*

lemma *Collect-mono*: $(!!x.\ P\ x --> Q\ x) ==> Collect\ P \subseteq Collect\ Q$
by *blast*

lemma *Int-Collect-mono*:
$A \subseteq B ==> (!!x.\ x \in A ==> P\ x --> Q\ x) ==> A \cap Collect\ P \subseteq B \cap Collect\ Q$
by *blast*

lemmas *basic-monos* =
subset-refl imp-refl disj-mono conj-mono
ex-mono Collect-mono in-mono

lemma *eq-to-mono*: $a = b ==> c = d ==> b --> d ==> a --> c$

by *iprover*

lemma *eq-to-mono2*: $a = b \Longrightarrow c = d \Longrightarrow \sim b \longrightarrow \sim d \Longrightarrow \sim a \longrightarrow \sim c$
by *iprover*

5.6 Inverse image of a function

constdefs
vimage :: $('a \Rightarrow 'b) \Rightarrow 'b\ set \Rightarrow 'a\ set$ (**infixr** $-`$ *90*)
[*code del*]: $f\ -`\ B == \{x.\ f\ x : B\}$

5.6.1 Basic rules

lemma *vimage-eq* [*simp*]: $(a : f\ -`\ B) = (f\ a : B)$
by (*unfold vimage-def*) *blast*

lemma *vimage-singleton-eq*: $(a : f\ -`\ \{b\}) = (f\ a = b)$
by *simp*

lemma *vimageI* [*intro*]: $f\ a = b \Longrightarrow b{:}B \Longrightarrow a : f\ -`\ B$
by (*unfold vimage-def*) *blast*

lemma *vimageI2*: $f\ a : A \Longrightarrow a : f\ -`\ A$
by (*unfold vimage-def*) *fast*

lemma *vimageE* [*elim!*]: $a{:}\ f\ -`\ B \Longrightarrow (!!x.\ f\ a = x \Longrightarrow x{:}B \Longrightarrow P) \Longrightarrow P$
by (*unfold vimage-def*) *blast*

lemma *vimageD*: $a : f\ -`\ A \Longrightarrow f\ a : A$
by (*unfold vimage-def*) *fast*

5.6.2 Equations

lemma *vimage-empty* [*simp*]: $f\ -`\ \{\} = \{\}$
by *blast*

lemma *vimage-Compl*: $f\ -`\ (-A) = -(f\ -`\ A)$
by *blast*

lemma *vimage-Un* [*simp*]: $f\ -`\ (A\ Un\ B) = (f\ -`\ A)\ Un\ (f\ -`\ B)$
by *blast*

lemma *vimage-Int* [*simp*]: $f\ -`\ (A\ Int\ B) = (f\ -`\ A)\ Int\ (f\ -`\ B)$
by *fast*

lemma *vimage-Union*: $f\ -`\ (Union\ A) = (UN\ X{:}A.\ f\ -`\ X)$
by *blast*

lemma *vimage-UN*: $f-`(UN\ x{:}A.\ B\ x) = (UN\ x{:}A.\ f\ -`\ B\ x)$

by *blast*

lemma *vimage-INT*: $f-`(INT\ x{:}A.\ B\ x) = (INT\ x{:}A.\ f\ -`\ B\ x)$
by *blast*

lemma *vimage-Collect-eq* [*simp*]: $f\ -`\ Collect\ P = \{y.\ P\ (f\ y)\}$
by *blast*

lemma *vimage-Collect*: $(!!x.\ P\ (f\ x) = Q\ x) ==> f\ -`\ (Collect\ P) = Collect\ Q$
by *blast*

lemma *vimage-insert*: $f-`(insert\ a\ B) = (f-`\{a\})\ Un\ (f-`B)$
— NOT suitable for rewriting because of the recurrence of $\{a\}$.
by *blast*

lemma *vimage-Diff*: $f\ -`\ (A\ -\ B) = (f\ -`\ A)\ -\ (f\ -`\ B)$
by *blast*

lemma *vimage-UNIV* [*simp*]: $f\ -`\ UNIV = UNIV$
by *blast*

lemma *vimage-eq-UN*: $f-`B = (UN\ y{:}\ B.\ f-`\{y\})$
— NOT suitable for rewriting
by *blast*

lemma *vimage-mono*: $A \subseteq B ==> f\ -`\ A \subseteq f\ -`\ B$
— monotonicity
by *blast*

lemma *vimage-image-eq* [*noatp*]: $f\ -`\ (f\ `\ A) = \{y.\ EX\ x{:}A.\ f\ x = f\ y\}$
by (*blast intro*: *sym*)

lemma *image-vimage-subset*: $f\ `\ (f\ -`\ A) <= A$
by *blast*

lemma *image-vimage-eq* [*simp*]: $f\ `\ (f\ -`\ A) = A\ Int\ range\ f$
by *blast*

lemma *image-Int-subset*: $f`(A\ Int\ B) <= f`A\ Int\ f`B$
by *blast*

lemma *image-diff-subset*: $f`A\ -\ f`B <= f`(A\ -\ B)$
by *blast*

lemma *image-UN*: $(f\ `\ (UNION\ A\ B)) = (UN\ x{:}A.(f\ `\ (B\ x)))$
by *blast*

5.7 Getting the Contents of a Singleton Set

definition *contents* :: *'a set* ⇒ *'a* **where**
  [*code del*]: *contents X* = (*THE x*. *X* = {*x*})

lemma *contents-eq* [*simp*]: *contents* {*x*} = *x*
  by (*simp add*: *contents-def*)

5.8 Transitivity rules for calculational reasoning

lemma *set-rev-mp*: *x*:*A* ==> *A* ⊆ *B* ==> *x*:*B*
  by (*rule subsetD*)

lemma *set-mp*: *A* ⊆ *B* ==> *x*:*A* ==> *x*:*B*
  by (*rule subsetD*)

lemmas *basic-trans-rules* [*trans*] =
  order-trans-rules set-rev-mp set-mp

5.9 Least value operator

lemma *Least-mono*:
  mono (*f*::*'a*::*order* => *'b*::*order*) ==> *EX x*:*S*. *ALL y*:*S*. *x* <= *y*
    ==> (*LEAST y*. *y* : *f* ‘ *S*) = *f* (*LEAST x*. *x* : *S*)
    — Courtesy of Stephan Merz
  apply *clarify*
  apply (*erule-tac P* = %*x*. *x* : *S* **in** *LeastI2-order*, *fast*)
  apply (*rule LeastI2-order*)
  apply (*auto elim*: *monoD intro*!: *order-antisym*)
  done

5.10 Rudimentary code generation

lemma *empty-code* [*code*]: {} *x* ⟷ *False*
  unfolding *empty-def Collect-def* **..**

lemma *UNIV-code* [*code*]: *UNIV x* ⟷ *True*
  unfolding *UNIV-def Collect-def* **..**

lemma *insert-code* [*code*]: *insert y A x* ⟷ *y* = *x* ∨ *A x*
  unfolding *insert-def Collect-def mem-def Un-def* **by** *auto*

lemma *inter-code* [*code*]: (*A* ∩ *B*) *x* ⟷ *A x* ∧ *B x*
  unfolding *Int-def Collect-def mem-def* **..**

lemma *union-code* [*code*]: (*A* ∪ *B*) *x* ⟷ *A x* ∨ *B x*
  unfolding *Un-def Collect-def mem-def* **..**

lemma *vimage-code* [*code*]: (*f* −‘ *A*) *x* = *A* (*f x*)
  unfolding *vimage-def Collect-def mem-def* **..**

5.11 Complete lattices

notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*)

class *complete-lattice* = *lattice* + *bot* + *top* +
fixes *Inf* :: *'a set* $\Rightarrow$ *'a* ($\bigsqcap$- [*900*] *900*)
and *Sup* :: *'a set* $\Rightarrow$ *'a* ($\bigsqcup$- [*900*] *900*)
assumes *Inf-lower*: $x \in A \Longrightarrow \bigsqcap A \sqsubseteq x$
and *Inf-greatest*: $(\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x) \Longrightarrow z \sqsubseteq \bigsqcap A$
assumes *Sup-upper*: $x \in A \Longrightarrow x \sqsubseteq \bigsqcup A$
and *Sup-least*: $(\bigwedge x.\ x \in A \Longrightarrow x \sqsubseteq z) \Longrightarrow \bigsqcup A \sqsubseteq z$
begin

lemma *Inf-Sup*: $\bigsqcap A = \bigsqcup \{b.\ \forall a \in A.\ b \leq a\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Sup-Inf*: $\bigsqcup A = \bigsqcap \{b.\ \forall a \in A.\ a \leq b\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Inf-Univ*: $\bigsqcap UNIV = \bigsqcup \{\}$
unfolding *Sup-Inf* **by** *auto*

lemma *Sup-Univ*: $\bigsqcup UNIV = \bigsqcap \{\}$
unfolding *Inf-Sup* **by** *auto*

lemma *Inf-insert*: $\bigsqcap insert\ a\ A = a \sqcap \bigsqcap A$
by (*auto intro*: *antisym Inf-greatest Inf-lower*)

lemma *Sup-insert*: $\bigsqcup insert\ a\ A = a \sqcup \bigsqcup A$
by (*auto intro*: *antisym Sup-least Sup-upper*)

lemma *Inf-singleton* [*simp*]:
$\bigsqcap \{a\} = a$
by (*auto intro*: *antisym Inf-lower Inf-greatest*)

lemma *Sup-singleton* [*simp*]:
$\bigsqcup \{a\} = a$
by (*auto intro*: *antisym Sup-upper Sup-least*)

lemma *Inf-insert-simp*:
$\bigsqcap insert\ a\ A = (if\ A = \{\}\ then\ a\ else\ a \sqcap \bigsqcap A)$
by (*cases* $A = \{\}$) (*simp-all*, *simp add*: *Inf-insert*)

lemma *Sup-insert-simp*:
$\bigsqcup insert\ a\ A = (if\ A = \{\}\ then\ a\ else\ a \sqcup \bigsqcup A)$
by (*cases* $A = \{\}$) (*simp-all*, *simp add*: *Sup-insert*)

lemma *Inf-binary*:
$\bigsqcap\{a, b\} = a \sqcap b$
by (*simp add*: *Inf-insert-simp*)

lemma *Sup-binary*:
$\bigsqcup\{a, b\} = a \sqcup b$
by (*simp add*: *Sup-insert-simp*)

lemma *bot-def*:
$bot = \bigsqcup\{\}$
by (*auto intro*: *antisym Sup-least*)

lemma *top-def*:
$top = \bigsqcap\{\}$
by (*auto intro*: *antisym Inf-greatest*)

lemma *sup-bot* [*simp*]:
$x \sqcup bot = x$
using *bot-least* [*of x*] **by** (*simp add*: *le-iff-sup sup-commute*)

lemma *inf-top* [*simp*]:
$x \sqcap top = x$
using *top-greatest* [*of x*] **by** (*simp add*: *le-iff-inf inf-commute*)

definition *SUPR* :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a* **where**
SUPR A f == $\bigsqcup$ (*f* ‘ *A*)

definition *INFI* :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a* **where**
INFI A f == $\bigsqcap$ (*f* ‘ *A*)

end

syntax
-SUP1 :: *pttrns* => *'b* => *'b* (*3SUP -./ -*) [*0*, *10*] *10*)
-SUP :: *pttrn* => *'a set* => *'b* => *'b* (*3SUP -:-./ -*) [*0*, *10*] *10*)
-INF1 :: *pttrns* => *'b* => *'b* (*3INF -./ -*) [*0*, *10*] *10*)
-INF :: *pttrn* => *'a set* => *'b* => *'b* (*3INF -:-./ -*) [*0*, *10*] *10*)

translations
SUP x y. B == *SUP x. SUP y. B*
SUP x. B == *CONST SUPR CONST UNIV* (%*x. B*)
SUP x. B == *SUP x:CONST UNIV. B*
SUP x:A. B == *CONST SUPR A* (%*x. B*)
INF x y. B == *INF x. INF y. B*
INF x. B == *CONST INFI CONST UNIV* (%*x. B*)
INF x. B == *INF x:CONST UNIV. B*
INF x:A. B == *CONST INFI A* (%*x. B*)

print-translation ⟪
let
  fun btr' syn (*A* :: *Abs abs* :: *ts*) =
    let val (*x*,*t*) = *atomic-abs-tr' abs*
    in list-comb (*Syntax.const syn* $ *x* $ *A* $ *t*, *ts*) *end*
  val const-syntax-name = *Sign.const-syntax-name* @{*theory*} *o fst o dest-Const*
in
[(*const-syntax-name* @{*term SUPR*}, *btr' -SUP*),(*const-syntax-name* @{*term INFI*}, *btr' -INF*)]
end
⟫

context *complete-lattice*
begin

lemma *le-SUPI*: *i* : *A* ⟹ *M i* ≤ (*SUP i*:*A*. *M i*)
  by (*auto simp add*: *SUPR-def intro*: *Sup-upper*)

lemma *SUP-leI*: (⋀*i*. *i* : *A* ⟹ *M i* ≤ *u*) ⟹ (*SUP i*:*A*. *M i*) ≤ *u*
  by (*auto simp add*: *SUPR-def intro*: *Sup-least*)

lemma *INF-leI*: *i* : *A* ⟹ (*INF i*:*A*. *M i*) ≤ *M i*
  by (*auto simp add*: *INFI-def intro*: *Inf-lower*)

lemma *le-INFI*: (⋀*i*. *i* : *A* ⟹ *u* ≤ *M i*) ⟹ *u* ≤ (*INF i*:*A*. *M i*)
  by (*auto simp add*: *INFI-def intro*: *Inf-greatest*)

lemma *SUP-const*[*simp*]: *A* ≠ {} ⟹ (*SUP i*:*A*. *M*) = *M*
  by (*auto intro*: *antisym SUP-leI le-SUPI*)

lemma *INF-const*[*simp*]: *A* ≠ {} ⟹ (*INF i*:*A*. *M*) = *M*
  by (*auto intro*: *antisym INF-leI le-INFI*)

end

5.12 Bool as complete lattice

instantiation *bool* :: *complete-lattice*
begin

definition
  Inf-bool-def: ⨅ *A* ⟷ (∀ *x*∈*A*. *x*)

definition
  Sup-bool-def: ⨆ *A* ⟷ (∃ *x*∈*A*. *x*)

instance
  by *intro-classes* (*auto simp add*: *Inf-bool-def Sup-bool-def le-bool-def*)

end

lemma *Inf-empty-bool* [*simp*]:
$\bigsqcap \{\}$
unfolding *Inf-bool-def* **by** *auto*

lemma *not-Sup-empty-bool* [*simp*]:
$\neg \bigsqcup \{\}$
unfolding *Sup-bool-def* **by** *auto*

5.13 Fun as complete lattice

instantiation *fun* :: (*type*, *complete-lattice*) *complete-lattice*
begin

definition
Inf-fun-def [*code del*]: $\bigsqcap A = (\lambda x.\ \bigsqcap \{y.\ \exists f{\in}A.\ y = f\ x\})$

definition
Sup-fun-def [*code del*]: $\bigsqcup A = (\lambda x.\ \bigsqcup \{y.\ \exists f{\in}A.\ y = f\ x\})$

instance
by *intro-classes*
(*auto simp add*: *Inf-fun-def Sup-fun-def le-fun-def*
intro: *Inf-lower Sup-upper Inf-greatest Sup-least*)

end

lemma *Inf-empty-fun*:
$\bigsqcap \{\} = (\lambda\text{-}.\ \bigsqcap \{\})$
by *rule* (*auto simp add*: *Inf-fun-def*)

lemma *Sup-empty-fun*:
$\bigsqcup \{\} = (\lambda\text{-}.\ \bigsqcup \{\})$
by *rule* (*auto simp add*: *Sup-fun-def*)

5.14 Set as lattice

lemma *inf-set-eq*: $A \sqcap B = A \cap B$
apply (*rule subset-antisym*)
apply (*rule Int-greatest*)
apply (*rule inf-le1*)
apply (*rule inf-le2*)
apply (*rule inf-greatest*)
apply (*rule Int-lower1*)
apply (*rule Int-lower2*)
done

lemma *sup-set-eq*: $A \sqcup B = A \cup B$

apply (*rule subset-antisym*)
apply (*rule sup-least*)
apply (*rule Un-upper1*)
apply (*rule Un-upper2*)
apply (*rule Un-least*)
apply (*rule sup-ge1*)
apply (*rule sup-ge2*)
done

lemma *mono-Int*: *mono f* $\Longrightarrow$ *f* (*A* $\cap$ *B*) $\subseteq$ *f A* $\cap$ *f B*
apply (*fold inf-set-eq sup-set-eq*)
apply (*erule mono-inf*)
done

lemma *mono-Un*: *mono f* $\Longrightarrow$ *f A* $\cup$ *f B* $\subseteq$ *f* (*A* $\cup$ *B*)
apply (*fold inf-set-eq sup-set-eq*)
apply (*erule mono-sup*)
done

lemma *Inf-set-eq*: $\bigsqcap S = \bigcap S$
apply (*rule subset-antisym*)
apply (*rule Inter-greatest*)
apply (*erule Inf-lower*)
apply (*rule Inf-greatest*)
apply (*erule Inter-lower*)
done

lemma *Sup-set-eq*: $\bigsqcup S = \bigcup S$
apply (*rule subset-antisym*)
apply (*rule Sup-least*)
apply (*erule Union-upper*)
apply (*rule Union-least*)
apply (*erule Sup-upper*)
done

lemma *top-set-eq*: *top* = *UNIV*
by (*iprover intro*!: *subset-antisym subset-UNIV top-greatest*)

lemma *bot-set-eq*: *bot* = {}
by (*iprover intro*!: *subset-antisym empty-subsetI bot-least*)

no-notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
Inf ($\bigsqcap$ - [*900*] *900*) **and**
Sup ($\bigsqcup$ - [*900*] *900*)

5.15 Misc theorem and ML bindings

lemmas *equalityI = subset-antisym*
lemmas *mem-simps =*
insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff
mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [*simp*] above.

ML ⟪
val Ball-def = @{thm Ball-def}
val Bex-def = @{thm Bex-def}
val CollectD = @{thm CollectD}
val CollectE = @{thm CollectE}
val CollectI = @{thm CollectI}
val Collect-conj-eq = @{thm Collect-conj-eq}
val Collect-mem-eq = @{thm Collect-mem-eq}
val IntD1 = @{thm IntD1}
val IntD2 = @{thm IntD2}
val IntE = @{thm IntE}
val IntI = @{thm IntI}
val Int-Collect = @{thm Int-Collect}
val UNIV-I = @{thm UNIV-I}
val UNIV-witness = @{thm UNIV-witness}
val UnE = @{thm UnE}
val UnI1 = @{thm UnI1}
val UnI2 = @{thm UnI2}
val ballE = @{thm ballE}
val ballI = @{thm ballI}
val bexCI = @{thm bexCI}
val bexE = @{thm bexE}
val bexI = @{thm bexI}
val bex-triv = @{thm bex-triv}
val bspec = @{thm bspec}
val contra-subsetD = @{thm contra-subsetD}
val distinct-lemma = @{thm distinct-lemma}
val eq-to-mono = @{thm eq-to-mono}
val eq-to-mono2 = @{thm eq-to-mono2}
val equalityCE = @{thm equalityCE}
val equalityD1 = @{thm equalityD1}
val equalityD2 = @{thm equalityD2}
val equalityE = @{thm equalityE}
val equalityI = @{thm equalityI}
val imageE = @{thm imageE}
val imageI = @{thm imageI}
val image-Un = @{thm image-Un}
val image-insert = @{thm image-insert}
val insert-commute = @{thm insert-commute}
val insert-iff = @{thm insert-iff}
val mem-Collect-eq = @{thm mem-Collect-eq}
val rangeE = @{thm rangeE}

val rangeI = @{thm rangeI}
val range-eqI = @{thm range-eqI}
val subsetCE = @{thm subsetCE}
val subsetD = @{thm subsetD}
val subsetI = @{thm subsetI}
val subset-refl = @{thm subset-refl}
val subset-trans = @{thm subset-trans}
val vimageD = @{thm vimageD}
val vimageE = @{thm vimageE}
val vimageI = @{thm vimageI}
val vimageI2 = @{thm vimageI2}
val vimage-Collect = @{thm vimage-Collect}
val vimage-Int = @{thm vimage-Int}
val vimage-Un = @{thm vimage-Un}
⟩⟩

end

6 Typedef: HOL type definitions

theory *Typedef*
imports *Set*
uses
(*Tools/typedef-package.ML*)
(*Tools/typecopy-package.ML*)
(*Tools/typedef-codegen.ML*)
begin

ML ⟨⟨
structure HOL = struct val thy = theory HOL end;
⟩⟩ — belongs to theory HOL

locale *type-definition* =
fixes *Rep* **and** *Abs* **and** *A*
assumes *Rep*: *Rep x* ∈ *A*
and *Rep-inverse*: *Abs* (*Rep x*) = *x*
and *Abs-inverse*: *y* ∈ *A* ==> *Rep* (*Abs y*) = *y*
— This will be axiomatized for each typedef!
begin

lemma *Rep-inject*:
(*Rep x* = *Rep y*) = (*x* = *y*)
proof
assume *Rep x* = *Rep y*
then have *Abs* (*Rep x*) = *Abs* (*Rep y*) **by** (*simp only*:)
moreover have *Abs* (*Rep x*) = *x* **by** (*rule Rep-inverse*)
moreover have *Abs* (*Rep y*) = *y* **by** (*rule Rep-inverse*)
ultimately show *x* = *y* **by** *simp*

next
assume $x = y$
thus $Rep\ x = Rep\ y$ **by** (*simp only*:)
qed

lemma *Abs-inject*:
assumes x: $x \in A$ **and** y: $y \in A$
shows $(Abs\ x = Abs\ y) = (x = y)$
proof
assume $Abs\ x = Abs\ y$
then have $Rep\ (Abs\ x) = Rep\ (Abs\ y)$ **by** (*simp only*:)
moreover from x **have** $Rep\ (Abs\ x) = x$ **by** (*rule Abs-inverse*)
moreover from y **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
ultimately show $x = y$ **by** *simp*
next
assume $x = y$
thus $Abs\ x = Abs\ y$ **by** (*simp only*:)
qed

lemma *Rep-cases* [*cases set*]:
assumes y: $y \in A$
and *hyp*: $!!x.\ y = Rep\ x \Longrightarrow P$
shows P
proof (*rule hyp*)
from y **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
thus $y = Rep\ (Abs\ y)$ **..**
qed

lemma *Abs-cases* [*cases type*]:
assumes r: $!!y.\ x = Abs\ y \Longrightarrow y \in A \Longrightarrow P$
shows P
proof (*rule r*)
have $Abs\ (Rep\ x) = x$ **by** (*rule Rep-inverse*)
thus $x = Abs\ (Rep\ x)$ **..**
show $Rep\ x \in A$ **by** (*rule Rep*)
qed

lemma *Rep-induct* [*induct set*]:
assumes y: $y \in A$
and *hyp*: $!!x.\ P\ (Rep\ x)$
shows $P\ y$
proof −
have $P\ (Rep\ (Abs\ y))$ **by** (*rule hyp*)
moreover from y **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
ultimately show $P\ y$ **by** *simp*
qed

lemma *Abs-induct* [*induct type*]:
assumes r: $!!y.\ y \in A \Longrightarrow P\ (Abs\ y)$

```
  shows P x
proof −
  have Rep x ∈ A by (rule Rep)
  then have P (Abs (Rep x)) by (rule r)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  ultimately show P x by simp
qed

lemma Rep-range: range Rep = A
proof
  show range Rep <= A using Rep by (auto simp add: image-def)
  show A <= range Rep
  proof
    fix x assume x : A
    hence x = Rep (Abs x) by (rule Abs-inverse [symmetric])
    thus x : range Rep by (rule range-eqI)
  qed
qed

lemma Abs-image: Abs ‘ A = UNIV
proof
  show Abs ‘ A <= UNIV by (rule subset-UNIV)
next
  show UNIV <= Abs ‘ A
  proof
    fix x
    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

end

use Tools/typedef-package.ML setup TypedefPackage.setup
use Tools/typecopy-package.ML setup TypecopyPackage.setup
use Tools/typedef-codegen.ML setup TypedefCodegen.setup

end
```

7 Fun: Notions about functions

```
theory Fun
imports Set
begin
```

As a simplification rule, it replaces all function equalities by first-order equalities.

lemma *expand-fun-eq*: $f = g \longleftrightarrow (\forall x.\ f\ x = g\ x)$
apply (*rule iffI*)
apply (*simp* (*no-asm-simp*))
apply (*rule ext*)
apply (*simp* (*no-asm-simp*))
done

lemma *apply-inverse*:
$f\ x = u \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow g\ (f\ x) = x) \Longrightarrow P\ x \Longrightarrow x = g\ u$
by *auto*

7.1 The Identity Function *id*

definition
$id :: 'a \Rightarrow 'a$
where
$id = (\lambda x.\ x)$

lemma *id-apply* [*simp*]: $id\ x = x$
by (*simp add*: *id-def*)

lemma *image-ident* [*simp*]: $(\%x.\ x)\ `\ Y = Y$
by *blast*

lemma *image-id* [*simp*]: $id\ `\ Y = Y$
by (*simp add*: *id-def*)

lemma *vimage-ident* [*simp*]: $(\%x.\ x) -` Y = Y$
by *blast*

lemma *vimage-id* [*simp*]: $id -` A = A$
by (*simp add*: *id-def*)

7.2 The Composition Operator $f \circ g$

definition
$comp :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** *o* *55*)
where
$f\ o\ g = (\lambda x.\ f\ (g\ x))$

notation (*xsymbols*)
comp (**infixl** $\circ$ *55*)

notation (*HTML* **output**)
comp (**infixl** $\circ$ *55*)

compatibility

lemmas *o-def* = *comp-def*

lemma *o-apply* [*simp*]: $(f\ o\ g)\ x = f\ (g\ x)$
by (*simp add*: *comp-def*)

lemma *o-assoc*: $f\ o\ (g\ o\ h) = f\ o\ g\ o\ h$
by (*simp add*: *comp-def*)

lemma *id-o* [*simp*]: $id\ o\ g = g$
by (*simp add*: *comp-def*)

lemma *o-id* [*simp*]: $f\ o\ id = f$
by (*simp add*: *comp-def*)

lemma *image-compose*: $(f\ o\ g)\ `\ r = f`(g`r)$
by (*simp add*: *comp-def*, *blast*)

lemma *UN-o*: $UNION\ A\ (g\ o\ f) = UNION\ (f`A)\ g$
by (*unfold comp-def*, *blast*)

7.3 The Forward Composition Operator *fcomp*

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

lemma *fcomp-apply*: $(f\ o>\ g)\ x = g\ (f\ x)$
by (*simp add*: *fcomp-def*)

lemma *fcomp-assoc*: $(f\ o>\ g)\ o>\ h = f\ o>\ (g\ o>\ h)$
by (*simp add*: *fcomp-def*)

lemma *id-fcomp* [*simp*]: $id\ o>\ g = g$
by (*simp add*: *fcomp-def*)

lemma *fcomp-id* [*simp*]: $f\ o>\ id = f$
by (*simp add*: *fcomp-def*)

no-notation *fcomp* (**infixl** $o>\ 60$)

7.4 Injectivity and Surjectivity

constdefs
$inj\text{-}on :: ['a => 'b, 'a\ set] => bool$ — injective
$inj\text{-}on\ f\ A == !\ x{:}A.\ !\ y{:}A.\ f(x){=}f(y)\ --> x{=}y$

A common special case: functions injective over the entire domain type.

abbreviation
$inj\ f == inj\text{-}on\ f\ UNIV$

definition
bij-betw :: ('*a* => '*b*) => '*a set* => '*b set* => *bool* **where** — bijective
[*code del*]: *bij-betw f A B* ⟷ *inj-on f A* & *f* ' *A* = *B*

constdefs
surj :: ('*a* => '*b*) => *bool*
surj f == ! *y*. ? *x*. *y*=*f*(*x*)

bij :: ('*a* => '*b*) => *bool*
bij f == *inj f* & *surj f*

lemma *injI*:
assumes ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
shows *inj f*
using *assms* **unfolding** *inj-on-def* **by** *auto*

For Proofs in *Tools/datatype-rep-proofs*

lemma *datatype-injI*:
(!! *x*. *ALL y*. *f*(*x*) = *f*(*y*) −−> *x*=*y*) ==> *inj*(*f*)
by (*simp add*: *inj-on-def*)

theorem *range-ex1-eq*: *inj f* ⟹ *b* : *range f* = (*EX*! *x*. *b* = *f x*)
by (*unfold inj-on-def*, *blast*)

lemma *injD*: [| *inj*(*f*); *f*(*x*) = *f*(*y*) |] ==> *x*=*y*
by (*simp add*: *inj-on-def*)

lemma *inj-eq*: *inj*(*f*) ==> (*f*(*x*) = *f*(*y*)) = (*x*=*y*)
by (*force simp add*: *inj-on-def*)

lemma *inj-on-id*[*simp*]: *inj-on id A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-id2*[*simp*]: *inj-on* (%*x*. *x*) *A*
by (*simp add*: *inj-on-def*)

lemma *surj-id*[*simp*]: *surj id*
by (*simp add*: *surj-def*)

lemma *bij-id*[*simp*]: *bij id*
by (*simp add*: *bij-def inj-on-id surj-id*)

lemma *inj-onI*:
(!! *x y*. [| *x*:*A*; *y*:*A*; *f*(*x*) = *f*(*y*) |] ==> *x*=*y*) ==> *inj-on f A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-inverseI*: (!!*x*. *x*:*A* ==> *g*(*f*(*x*)) = *x*) ==> *inj-on f A*
by (*auto dest*: *arg-cong* [*of concl*: *g*] *simp add*: *inj-on-def*)

lemma *inj-onD*: [| *inj-on f A*; $f(x)=f(y)$; x:*A*; y:*A* |] ==> $x=y$
by (*unfold inj-on-def*, *blast*)

lemma *inj-on-iff*: [| *inj-on f A*; x:*A*; y:*A* |] ==> $(f(x)=f(y)) = (x=y)$
by (*blast dest*!: *inj-onD*)

lemma *comp-inj-on*:
[| *inj-on f A*; *inj-on g* (*f*'*A*) |] ==> *inj-on* (*g o f*) *A*
by (*simp add*: *comp-def inj-on-def*)

lemma *inj-on-imageI*: *inj-on* (*g o f*) *A* ⟹ *inj-on g* (*f* ' *A*)
apply(*simp add*:*inj-on-def image-def*)
apply *blast*
done

lemma *inj-on-image-iff*: ⟦ *ALL* x:*A*. *ALL* y:*A*. $(g(f\,x) = g(f\,y)) = (g\,x = g\,y)$;
inj-on f A ⟧ ⟹ *inj-on g* (*f* ' *A*) = *inj-on g A*
apply(*unfold inj-on-def*)
apply *blast*
done

lemma *inj-on-contraD*: [| *inj-on f A*; ~x=y; x:*A*; y:*A* |] ==> ~ $f(x)=f(y)$
by (*unfold inj-on-def*, *blast*)

lemma *inj-singleton*: *inj* (%*s*. {*s*})
by (*simp add*: *inj-on-def*)

lemma *inj-on-empty*[*iff*]: *inj-on f* {}
by(*simp add*: *inj-on-def*)

lemma *subset-inj-on*: [| *inj-on f B*; *A* <= *B* |] ==> *inj-on f A*
by (*unfold inj-on-def*, *blast*)

lemma *inj-on-Un*:
inj-on f (*A Un B*) =
(*inj-on f A* & *inj-on f B* & *f*'(*A*−*B*) *Int f*'(*B*−*A*) = {})
apply(*unfold inj-on-def*)
apply (*blast intro*:*sym*)
done

lemma *inj-on-insert*[*iff*]:
inj-on f (*insert a A*) = (*inj-on f A* & *f a* ~: *f*'(*A*−{*a*}))
apply(*unfold inj-on-def*)
apply (*blast intro*:*sym*)
done

lemma *inj-on-diff*: *inj-on f A* ==> *inj-on f* (*A*−*B*)
apply(*unfold inj-on-def*)

apply (*blast*)
done

lemma *surjI*: (!! *x*. *g*(*f x*) = *x*) ==> *surj g*
apply (*simp add*: *surj-def*)
apply (*blast intro*: *sym*)
done

lemma *surj-range*: *surj f* ==> *range f* = *UNIV*
by (*auto simp add*: *surj-def*)

lemma *surjD*: *surj f* ==> *EX x*. *y* = *f x*
by (*simp add*: *surj-def*)

lemma *surjE*: *surj f* ==> (!!*x*. *y* = *f x* ==> *C*) ==> *C*
by (*simp add*: *surj-def*, *blast*)

lemma *comp-surj*: [| *surj f*; *surj g* |] ==> *surj* (*g o f*)
apply (*simp add*: *comp-def surj-def*, *clarify*)
apply (*drule-tac x* = *y* **in** *spec*, *clarify*)
apply (*drule-tac x* = *x* **in** *spec*, *blast*)
done

lemma *bijI*: [| *inj f*; *surj f* |] ==> *bij f*
by (*simp add*: *bij-def*)

lemma *bij-is-inj*: *bij f* ==> *inj f*
by (*simp add*: *bij-def*)

lemma *bij-is-surj*: *bij f* ==> *surj f*
by (*simp add*: *bij-def*)

lemma *bij-betw-imp-inj-on*: *bij-betw f A B* ⟹ *inj-on f A*
by (*simp add*: *bij-betw-def*)

lemma *bij-betw-inv*: **assumes** *bij-betw f A B* **shows** *EX g*. *bij-betw g B A*
proof −
have *i*: *inj-on f A* **and** *s*: *f* ‘ *A* = *B*
using *assms* **by**(*auto simp*:*bij-betw-def*)
let *?P* = %*b a*. *a*:*A* ∧ *f a* = *b* **let** *?g* = %*b*. *The* (*?P b*)
{ fix *a b* **assume** *P*: *?P b a*
hence *ex1*: ∃ *a*. *?P b a* **using** *s* **unfolding** *image-def* **by** *blast*
hence *uex1*: ∃!*a*. *?P b a* **by**(*blast dest*:*inj-onD*[*OF i*])
hence *?g b* = *a* **using** *the1-equality*[*OF uex1*, *OF P*] *P* **by** *simp*
} note *g* = *this*
have *inj-on ?g B*
proof(*rule inj-onI*)
fix *x y* **assume** *x*:*B y*:*B ?g x* = *?g y*
from *s* ‹*x*:*B*› **obtain** *a1* **where** *a1*: *?P x a1* **unfolding** *image-def* **by** *blast*

```
    from s ‹y:B› obtain a2 where a2: ?P y a2 unfolding image-def by blast
    from g[OF a1] a1 g[OF a2] a2 ‹?g x = ?g y› show x=y by simp
  qed
  moreover have ?g ‘ B = A
  proof(auto simp:image-def)
    fix b assume b:B
    with s obtain a where P: ?P b a unfolding image-def by blast
    thus ?g b ∈ A using g[OF P] by auto
  next
    fix a assume a:A
    then obtain b where P: ?P b a using s unfolding image-def by blast
    then have b:B using s unfolding image-def by blast
    with g[OF P] show ∃ b∈B. a = ?g b by blast
  qed
  ultimately show ?thesis by(auto simp:bij-betw-def)
qed

lemma surj-image-vimage-eq: surj f ==> f ‘ (f −‘ A) = A
by (simp add: surj-range)

lemma inj-vimage-image-eq: inj f ==> f −‘ (f ‘ A) = A
by (simp add: inj-on-def, blast)

lemma vimage-subsetD: surj f ==> f −‘ B <= A ==> B <= f ‘ A
apply (unfold surj-def)
apply (blast intro: sym)
done

lemma vimage-subsetI: inj f ==> B <= f ‘ A ==> f −‘ B <= A
by (unfold inj-on-def, blast)

lemma vimage-subset-eq: bij f ==> (f −‘ B <= A) = (B <= f ‘ A)
apply (unfold bij-def)
apply (blast del: subsetI intro: vimage-subsetI vimage-subsetD)
done

lemma inj-on-image-Int:
   [| inj-on f C;  A<=C;  B<=C |] ==> f‘(A Int B) = f‘A Int f‘B
apply (simp add: inj-on-def, blast)
done

lemma inj-on-image-set-diff:
   [| inj-on f C;  A<=C;  B<=C |] ==> f‘(A−B) = f‘A − f‘B
apply (simp add: inj-on-def, blast)
done

lemma image-Int: inj f ==> f‘(A Int B) = f‘A Int f‘B
by (simp add: inj-on-def, blast)
```

lemma *image-set-diff*: *inj f ==> f‘(A−B) = f‘A − f‘B*
by (*simp add*: *inj-on-def*, *blast*)

lemma *inj-image-mem-iff*: *inj f ==> (f a : f‘A) = (a : A)*
by (*blast dest*: *injD*)

lemma *inj-image-subset-iff*: *inj f ==> (f‘A <= f‘B) = (A<=B)*
by (*simp add*: *inj-on-def*, *blast*)

lemma *inj-image-eq-iff*: *inj f ==> (f‘A = f‘B) = (A = B)*
by (*blast dest*: *injD*)

lemma *image-INT*:
[| *inj-on f C*; *ALL x:A. B x <= C*; *j:A* |]
==> *f ‘ (INTER A B) = (INT x:A. f ‘ B x)*
apply (*simp add*: *inj-on-def*, *blast*)
done

lemma *bij-image-INT*: *bij f ==> f ‘ (INTER A B) = (INT x:A. f ‘ B x)*
apply (*simp add*: *bij-def*)
apply (*simp add*: *inj-on-def surj-def*, *blast*)
done

lemma *surj-Compl-image-subset*: *surj f ==> −(f‘A) <= f‘(−A)*
by (*auto simp add*: *surj-def*)

lemma *inj-image-Compl-subset*: *inj f ==> f‘(−A) <= −(f‘A)*
by (*auto simp add*: *inj-on-def*)

lemma *bij-image-Compl-eq*: *bij f ==> f‘(−A) = −(f‘A)*
apply (*simp add*: *bij-def*)
apply (*rule equalityI*)
apply (*simp-all* (*no-asm-simp*) *add*: *inj-image-Compl-subset surj-Compl-image-subset*)
done

7.5 Function Updating

constdefs
fun-upd :: (*'a => 'b*) => *'a => 'b => ('a => 'b*)
fun-upd f a b == % x. if x=a then b else f x

nonterminals
updbinds updbind
syntax
-*updbind* :: [*'a*, *'a*] => *updbind* ((*2*- :=/ -))
:: *updbind* => *updbinds* (-)
-*updbinds*:: [*updbind*, *updbinds*] => *updbinds* (-,/ -)

-Update :: ['*a*, *updbinds*] => '*a* (-/'((-)') [*1000*,*0*] *900*)

translations
-Update f (*-updbinds b bs*) == *-Update* (*-Update f b*) *bs*
f(*x*:=*y*) == *fun-upd f x y*

lemma *fun-upd-idem-iff*: (*f*(*x*:=*y*) = *f*) = (*f x* = *y*)
apply (*simp add*: *fun-upd-def*, *safe*)
apply (*erule subst*)
apply (*rule-tac* [*2*] *ext*, *auto*)
done

lemmas *fun-upd-idem* = *fun-upd-idem-iff* [*THEN iffD2*, *standard*]

lemmas *fun-upd-triv* = *refl* [*THEN fun-upd-idem*]
declare *fun-upd-triv* [*iff*]

lemma *fun-upd-apply* [*simp*]: (*f*(*x*:=*y*))*z* = (*if z*=*x then y else f z*)
by (*simp add*: *fun-upd-def*)

lemma *fun-upd-same*: (*f*(*x*:=*y*)) *x* = *y*
by *simp*

lemma *fun-upd-other*: *z*~=*x* ==> (*f*(*x*:=*y*)) *z* = *f z*
by *simp*

lemma *fun-upd-upd* [*simp*]: *f*(*x*:=*y*,*x*:=*z*) = *f*(*x*:=*z*)
by (*simp add*: *expand-fun-eq*)

lemma *fun-upd-twist*: *a* ~= *c* ==> (*m*(*a*:=*b*))(*c*:=*d*) = (*m*(*c*:=*d*))(*a*:=*b*)
by (*rule ext*, *auto*)

lemma *inj-on-fun-updI*: ⟦ *inj-on f A*; *y* ∉ *f*‘*A* ⟧ ⟹ *inj-on* (*f*(*x*:=*y*)) *A*
by(*fastsimp simp*:*inj-on-def image-def*)

lemma *fun-upd-image*:
f(*x*:=*y*) ‘ *A* = (*if x* ∈ *A then insert y* (*f* ‘ (*A*−{*x*})) *else f* ‘ *A*)
by *auto*

7.6 *override-on*

definition
override-on :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a set* ⇒ '*a* ⇒ '*b*
where

override-on f g A = (λa. if a ∈ A then g a else f a)

lemma *override-on-emptyset*[*simp*]: *override-on f g {} = f*
by(*simp add*:*override-on-def*)

lemma *override-on-apply-notin*[*simp*]: *a ~: A ==> (override-on f g A) a = f a*
by(*simp add*:*override-on-def*)

lemma *override-on-apply-in*[*simp*]: *a : A ==> (override-on f g A) a = g a*
by(*simp add*:*override-on-def*)

7.7 *swap*

definition
swap :: 'a ⇒ 'a ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
where
swap a b f = f (a := f b, b:= f a)

lemma *swap-self*: *swap a a f = f*
by (*simp add*: *swap-def*)

lemma *swap-commute*: *swap a b f = swap b a f*
by (*rule ext*, *simp add*: *fun-upd-def swap-def*)

lemma *swap-nilpotent* [*simp*]: *swap a b (swap a b f) = f*
by (*rule ext*, *simp add*: *fun-upd-def swap-def*)

lemma *inj-on-imp-inj-on-swap*:
[|inj-on f A; a ∈ A; b ∈ A|] ==> inj-on (swap a b f) A
by (*simp add*: *inj-on-def swap-def*, *blast*)

lemma *inj-on-swap-iff* [*simp*]:
assumes *A*: *a ∈ A b ∈ A* **shows** *inj-on (swap a b f) A = inj-on f A*
proof
assume *inj-on (swap a b f) A*
with *A* **have** *inj-on (swap a b (swap a b f)) A*
by (*iprover intro*: *inj-on-imp-inj-on-swap*)
thus *inj-on f A* **by** *simp*
next
assume *inj-on f A*
with *A* **show** *inj-on (swap a b f) A* **by**(*iprover intro*: *inj-on-imp-inj-on-swap*)
qed

lemma *surj-imp-surj-swap*: *surj f ==> surj (swap a b f)*
apply (*simp add*: *surj-def swap-def*, *clarify*)
apply (*case-tac y = f b*, *blast*)
apply (*case-tac y = f a*, *auto*)
done

lemma *surj-swap-iff* [*simp*]: *surj* (*swap a b f*) = *surj f*
proof
assume *surj* (*swap a b f*)
hence *surj* (*swap a b* (*swap a b f*)) **by** (*rule surj-imp-surj-swap*)
thus *surj f* **by** *simp*
next
assume *surj f*
thus *surj* (*swap a b f*) **by** (*rule surj-imp-surj-swap*)
qed

lemma *bij-swap-iff*: *bij* (*swap a b f*) = *bij f*
by (*simp add*: *bij-def*)

hide (**open**) *const swap*

7.8 Proof tool setup

simplifies terms of the form f(...,x:=y,...,x:=z,...) to f(...,x:=z,...)

simproc-setup *fun-upd2* (*f*(*v* := *w*, *x* := *y*)) = ⟪ *fn* - =>
let
fun gen-fun-upd NONE T - - = *NONE*
| *gen-fun-upd* (*SOME f*) *T x y* = *SOME* (*Const* (@{*const-name fun-upd*}, *T*)
$ *f* $ *x* $ *y*)
fun dest-fun-T1 (*Type* (-, *T* :: *Ts*)) = *T*
fun find-double (*t as Const* (@{*const-name fun-upd*},*T*) $ *f* $ *x* $ *y*) =
let
fun find (*Const* (@{*const-name fun-upd*},*T*) $ *g* $ *v* $ *w*) =
if v aconv x then SOME g else gen-fun-upd (*find g*) *T v w*
| *find t* = *NONE*
in (*dest-fun-T1 T*, *gen-fun-upd* (*find f*) *T x y*) *end*

fun proc ss ct =
let
val ctxt = *Simplifier.the-context ss*
val t = *Thm.term-of ct*
in
case find-double t of
(*T*, *NONE*) => *NONE*
| (*T*, *SOME rhs*) =>
SOME (*Goal.prove ctxt* [] [] (*Logic.mk-equals* (*t*, *rhs*))
(*fn* - =>
rtac eq-reflection 1 THEN
rtac ext 1 THEN
simp-tac (*Simplifier.inherit-context ss* @{*simpset*}) *1*))
end
in proc end
⟫

7.9 Code generator setup

types-code
fun ((- ->/ -))
attach (*term-of*) ⟪
fun term-of-fun-type - aT - bT - = Free (<*function*>, *aT* --> *bT*);
⟫
attach (*test*) ⟪
fun gen-fun-type aF aT bG bT i =
let
val tab = ref [];
fun mk-upd (*x*, (-, *y*)) *t = Const* (*Fun.fun-upd*,
(*aT* --> *bT*) --> *aT* --> *bT* --> *aT* --> *bT*) $ *t* $ *aF x* $ *y* ()
in
(*fn x* =>
case AList.lookup op = (!*tab*) *x of*
NONE =>
let val p as (*y*, -) = *bG i*
in (*tab* := (*x*, *p*) :: !*tab*; *y*) *end*
| *SOME* (*y*, -) => *y*,
fn () => *Basics.fold mk-upd* (!*tab*) (*Const* (*HOL.undefined*, *aT* --> *bT*)))
end;
⟫

code-const *op* ∘
(*SML* **infixl** *5 o*)
(*Haskell* **infixr** *9* .)

code-const *id*
(*Haskell id*)

end

8 Sum-Type: The Disjoint Sum of Two Types

theory *Sum-Type*
imports *Typedef Fun*
begin

The representations of the two injections

constdefs
Inl-Rep :: ['*a*, '*a*, '*b*, *bool*] => *bool*
Inl-Rep == (%*a*. %*x y p*. *x*=*a* & *p*)

Inr-Rep :: ['*b*, '*a*, '*b*, *bool*] => *bool*
Inr-Rep == (%*b*. %*x y p*. *y*=*b* & ~*p*)

global

typedef (*Sum*)
(*'a*, *'b*) + (**infixr** + *10*)
= {*f*. (*? a*. *f* = *Inl-Rep*(*a*::*'a*)) | (*? b*. *f* = *Inr-Rep*(*b*::*'b*))}
by *auto*

local

abstract constants and syntax

constdefs
Inl :: *'a* => *'a* + *'b*
Inl == (%*a*. *Abs-Sum*(*Inl-Rep*(*a*)))

Inr :: *'b* => *'a* + *'b*
Inr == (%*b*. *Abs-Sum*(*Inr-Rep*(*b*)))

Plus :: [*'a set*, *'b set*] => (*'a* + *'b*) *set* (**infixr** <+> *65*)
A <+> *B* == (*Inl*‘*A*) *Un* (*Inr*‘*B*)
— disjoint sum for sets; the operator + is overloaded with wrong type!

Part :: [*'a set*, *'b* => *'a*] => *'a set*
Part A h == *A Int* {*x*. *? z*. *x* = *h*(*z*)}
— for selecting out the components of a mutually recursive definition

lemma *Inl-RepI*: *Inl-Rep*(*a*) : *Sum*
by (*auto simp add*: *Sum-def*)

lemma *Inr-RepI*: *Inr-Rep*(*b*) : *Sum*
by (*auto simp add*: *Sum-def*)

lemma *inj-on-Abs-Sum*: *inj-on Abs-Sum Sum*
apply (*rule inj-on-inverseI*)
apply (*erule Abs-Sum-inverse*)
done

8.1 Freeness Properties for *Inl* and *Inr*

Distinctness

lemma *Inl-Rep-not-Inr-Rep*: *Inl-Rep*(*a*) ~= *Inr-Rep*(*b*)
by (*auto simp add*: *Inl-Rep-def Inr-Rep-def expand-fun-eq*)

lemma *Inl-not-Inr* [*iff*]: *Inl*(*a*) ~= *Inr*(*b*)
apply (*simp add*: *Inl-def Inr-def*)

apply (*rule inj-on-Abs-Sum* [*THEN inj-on-contraD*])
apply (*rule Inl-Rep-not-Inr-Rep*)
apply (*rule Inl-RepI*)
apply (*rule Inr-RepI*)
done

lemmas *Inr-not-Inl* = *Inl-not-Inr* [*THEN not-sym*, *standard*]
declare *Inr-not-Inl* [*iff*]

lemmas *Inl-neq-Inr* = *Inl-not-Inr* [*THEN notE*, *standard*]
lemmas *Inr-neq-Inl* = *sym* [*THEN Inl-neq-Inr*, *standard*]

Injectiveness

lemma *Inl-Rep-inject*: *Inl-Rep*(*a*) = *Inl-Rep*(*c*) ==> *a*=*c*
by (*auto simp add*: *Inl-Rep-def expand-fun-eq*)

lemma *Inr-Rep-inject*: *Inr-Rep*(*b*) = *Inr-Rep*(*d*) ==> *b*=*d*
by (*auto simp add*: *Inr-Rep-def expand-fun-eq*)

lemma *inj-Inl* [*simp*]: *inj-on Inl A*
apply (*simp add*: *Inl-def*)
apply (*rule inj-onI*)
apply (*erule inj-on-Abs-Sum* [*THEN inj-onD*, *THEN Inl-Rep-inject*])
apply (*rule Inl-RepI*)
apply (*rule Inl-RepI*)
done

lemmas *Inl-inject* = *inj-Inl* [*THEN injD*, *standard*]

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
apply (*simp add*: *Inr-def*)
apply (*rule inj-onI*)
apply (*erule inj-on-Abs-Sum* [*THEN inj-onD*, *THEN Inr-Rep-inject*])
apply (*rule Inr-RepI*)
apply (*rule Inr-RepI*)
done

lemmas *Inr-inject* = *inj-Inr* [*THEN injD*, *standard*]

lemma *Inl-eq* [*iff*]: (*Inl*(*x*)=*Inl*(*y*)) = (*x*=*y*)
by (*blast dest*!: *Inl-inject*)

lemma *Inr-eq* [*iff*]: (*Inr*(*x*)=*Inr*(*y*)) = (*x*=*y*)
by (*blast dest*!: *Inr-inject*)

8.2 The Disjoint Sum of Sets

lemma *InlI* [*intro*!]: *a* : *A* ==> *Inl*(*a*) : *A* <+> *B*
by (*simp add*: *Plus-def*)

lemma *InrI* [*intro*!]: $b : B \Longrightarrow Inr(b) : A <+> B$
by (*simp add*: *Plus-def*)

lemma *PlusE* [*elim*!]:
$[| u: A <+> B;$
$!!x.\ [| x:A;\ u=Inl(x) |] \Longrightarrow P;$
$!!y.\ [| y:B;\ u=Inr(y) |] \Longrightarrow P$
$|] \Longrightarrow P$
by (*auto simp add*: *Plus-def*)

Exhaustion rule for sums, a degenerate form of induction

lemma *sumE*:
$[| !!x::'a.\ s = Inl(x) \Longrightarrow P;\ !!y::'b.\ s = Inr(y) \Longrightarrow P$
$|] \Longrightarrow P$
apply (*rule Abs-Sum-cases* [*of s*])
apply (*auto simp add*: *Sum-def Inl-def Inr-def*)
done

lemma *UNIV-Plus-UNIV* [*simp*]: $UNIV <+> UNIV = UNIV$
apply (*rule set-ext*)
apply(*rename-tac s*)
apply(*rule-tac s=s* **in** *sumE*)
apply *auto*
done

8.3 The *Part* Primitive

lemma *Part-eqI* [*intro*]: $[| a : A;\ a=h(b) |] \Longrightarrow a : Part\ A\ h$
by (*auto simp add*: *Part-def*)

lemmas *PartI* = *Part-eqI* [*OF - refl*, *standard*]

lemma *PartE* [*elim*!]: $[| a : Part\ A\ h;\ !!z.\ [| a : A;\ a=h(z) |] \Longrightarrow P |] \Longrightarrow P$
by (*auto simp add*: *Part-def*)

lemma *Part-subset*: $Part\ A\ h <= A$
by (*auto simp add*: *Part-def*)

lemma *Part-mono*: $A<=B \Longrightarrow Part\ A\ h <= Part\ B\ h$
by *blast*

lemmas *basic-monos* = *basic-monos Part-mono*

lemma *PartD1*: $a : Part\ A\ h \Longrightarrow a : A$

by (*simp add*: *Part-def*)

lemma *Part-id*: *Part A* (%*x*. *x*) = *A*
by *blast*

lemma *Part-Int*: *Part* (*A Int B*) *h* = (*Part A h*) *Int* (*Part B h*)
by *blast*

lemma *Part-Collect*: *Part* (*A Int* {*x*. *P x*}) *h* = (*Part A h*) *Int* {*x*. *P x*}
by *blast*

ML
⟪
val Inl-RepI = *thm Inl-RepI*;
val Inr-RepI = *thm Inr-RepI*;
val inj-on-Abs-Sum = *thm inj-on-Abs-Sum*;
val Inl-Rep-not-Inr-Rep = *thm Inl-Rep-not-Inr-Rep*;
val Inl-not-Inr = *thm Inl-not-Inr*;
val Inr-not-Inl = *thm Inr-not-Inl*;
val Inl-neq-Inr = *thm Inl-neq-Inr*;
val Inr-neq-Inl = *thm Inr-neq-Inl*;
val Inl-Rep-inject = *thm Inl-Rep-inject*;
val Inr-Rep-inject = *thm Inr-Rep-inject*;
val inj-Inl = *thm inj-Inl*;
val Inl-inject = *thm Inl-inject*;
val inj-Inr = *thm inj-Inr*;
val Inr-inject = *thm Inr-inject*;
val Inl-eq = *thm Inl-eq*;
val Inr-eq = *thm Inr-eq*;
val InlI = *thm InlI*;
val InrI = *thm InrI*;
val PlusE = *thm PlusE*;
val sumE = *thm sumE*;
val Part-eqI = *thm Part-eqI*;
val PartI = *thm PartI*;
val PartE = *thm PartE*;
val Part-subset = *thm Part-subset*;
val Part-mono = *thm Part-mono*;
val PartD1 = *thm PartD1*;
val Part-id = *thm Part-id*;
val Part-Int = *thm Part-Int*;
val Part-Collect = *thm Part-Collect*;

val basic-monos = *thms basic-monos*;
⟫

end

9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*
imports *Lattices Sum-Type*
uses
(*Tools/inductive-package.ML*)
Tools/dseq.ML
(*Tools/inductive-codegen.ML*)
(*Tools/datatype-aux.ML*)
(*Tools/datatype-prop.ML*)
(*Tools/datatype-rep-proofs.ML*)
(*Tools/datatype-abs-proofs.ML*)
(*Tools/datatype-case.ML*)
(*Tools/datatype-package.ML*)
(*Tools/old-primrec-package.ML*)
(*Tools/primrec-package.ML*)
(*Tools/datatype-codegen.ML*)
begin

9.1 Least and greatest fixed points

context *complete-lattice*
begin

definition
lfp :: $('a \Rightarrow 'a) \Rightarrow 'a$ **where**
lfp f = *Inf* $\{u.\ f\ u \leq u\}$ — least fixed point

definition
gfp :: $('a \Rightarrow 'a) \Rightarrow 'a$ **where**
gfp f = *Sup* $\{u.\ u \leq f\ u\}$ — greatest fixed point

9.2 Proof of Knaster-Tarski Theorem using *lfp*

lfp f is the least upper bound of the set $\{u.\ f\ u \leq u\}$

lemma *lfp-lowerbound*: *f A* $\leq$ *A* ==> *lfp f* $\leq$ *A*
by (*auto simp add*: *lfp-def intro*: *Inf-lower*)

lemma *lfp-greatest*: (!!*u*. *f u* $\leq$ *u* ==> *A* $\leq$ *u*) ==> *A* $\leq$ *lfp f*
by (*auto simp add*: *lfp-def intro*: *Inf-greatest*)

end

lemma *lfp-lemma2*: *mono f* ==> *f* (*lfp f*) $\leq$ *lfp f*
by (*iprover intro*: *lfp-greatest order-trans monoD lfp-lowerbound*)

lemma *lfp-lemma3*: *mono f* ==> *lfp f* $\leq$ *f* (*lfp f*)
by (*iprover intro*: *lfp-lemma2 monoD lfp-lowerbound*)

lemma *lfp-unfold*: *mono f ==> lfp f = f (lfp f)*
by (*iprover intro*: *order-antisym lfp-lemma2 lfp-lemma3*)

lemma *lfp-const*: *lfp* ($\lambda x.\ t$) = *t*
by (*rule lfp-unfold*) (*simp add*:*mono-def*)

9.3 General induction rules for least fixed points

theorem *lfp-induct*:
assumes *mono*: *mono f* **and** *ind*: *f* (*inf* (*lfp f*) *P*) <= *P*
shows *lfp f* <= *P*
proof −
have *inf* (*lfp f*) *P* <= *lfp f* **by** (*rule inf-le1*)
with *mono* **have** *f* (*inf* (*lfp f*) *P*) <= *f* (*lfp f*) **..**
also from *mono* **have** *f* (*lfp f*) = *lfp f* **by** (*rule lfp-unfold* [*symmetric*])
finally have *f* (*inf* (*lfp f*) *P*) <= *lfp f* **.**
from *this* **and** *ind* **have** *f* (*inf* (*lfp f*) *P*) <= *inf* (*lfp f*) *P* **by** (*rule le-infI*)
hence *lfp f* <= *inf* (*lfp f*) *P* **by** (*rule lfp-lowerbound*)
also have *inf* (*lfp f*) *P* <= *P* **by** (*rule inf-le2*)
finally show *?thesis* **.**
qed

lemma *lfp-induct-set*:
assumes *lfp*: *a*: *lfp*(*f*)
and *mono*: *mono*(*f*)
and *indhyp*: !!*x*. [| *x*: *f*(*lfp*(*f*) *Int* {*x*. *P*(*x*)}) |] ==> *P*(*x*)
shows *P*(*a*)
by (*rule lfp-induct* [*THEN subsetD*, *THEN CollectD*, *OF mono - lfp*])
(*auto simp*: *inf-set-eq intro*: *indhyp*)

lemma *lfp-ordinal-induct*:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'a*
assumes *mono*: *mono f*
and *P-f*: ⋀*S*. *P S* ⟹ *P* (*f S*)
and *P-Union*: ⋀*M*. ∀ *S*∈*M*. *P S* ⟹ *P* (*Sup M*)
shows *P* (*lfp f*)
proof −
let *?M* = {*S*. *S* ≤ *lfp f* ∧ *P S*}
have *P* (*Sup ?M*) **using** *P-Union* **by** *simp*
also have *Sup ?M* = *lfp f*
proof (*rule antisym*)
show *Sup ?M* ≤ *lfp f* **by** (*blast intro*: *Sup-least*)
hence *f* (*Sup ?M*) ≤ *f* (*lfp f*) **by** (*rule mono* [*THEN monoD*])
hence *f* (*Sup ?M*) ≤ *lfp f* **using** *mono* [*THEN lfp-unfold*] **by** *simp*
hence *f* (*Sup ?M*) ∈ *?M* **using** *P-f P-Union* **by** *simp*
hence *f* (*Sup ?M*) ≤ *Sup ?M* **by** (*rule Sup-upper*)
thus *lfp f* ≤ *Sup ?M* **by** (*rule lfp-lowerbound*)
qed

finally show *?thesis* **.**
qed

lemma *lfp-ordinal-induct-set*:
assumes *mono*: *mono f*
and *P-f*: !!*S*. *P S* ==> *P*(*f S*)
and *P-Union*: !!*M*. !*S*:*M*. *P S* ==> *P*(*Union M*)
shows *P*(*lfp f*)
using *assms* **unfolding** *Sup-set-eq* [*symmetric*]
by (*rule lfp-ordinal-induct* [**where** *P*=*P*])

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

lemma *def-lfp-unfold*: [| *h*==*lfp*(*f*); *mono*(*f*) |] ==> *h* = *f*(*h*)
by (*auto intro*!: *lfp-unfold*)

lemma *def-lfp-induct*:
[| *A* == *lfp*(*f*); *mono*(*f*);
f (*inf A P*) ≤ *P*
|] ==> *A* ≤ *P*
by (*blast intro*: *lfp-induct*)

lemma *def-lfp-induct-set*:
[| *A* == *lfp*(*f*); *mono*(*f*); *a*:*A*;
!!*x*. [| *x*: *f*(*A Int* {*x*. *P*(*x*)}) |] ==> *P*(*x*)
|] ==> *P*(*a*)
by (*blast intro*: *lfp-induct-set*)

lemma *lfp-mono*: (!!*Z*. *f Z* ≤ *g Z*) ==> *lfp f* ≤ *lfp g*
by (*rule lfp-lowerbound* [*THEN lfp-greatest*], *blast intro*: *order-trans*)

9.4 Proof of Knaster-Tarski Theorem using *gfp*

gfp f is the greatest lower bound of the set {*u*. *u* ≤ *f u*}

lemma *gfp-upperbound*: *X* ≤ *f X* ==> *X* ≤ *gfp f*
by (*auto simp add*: *gfp-def intro*: *Sup-upper*)

lemma *gfp-least*: (!!*u*. *u* ≤ *f u* ==> *u* ≤ *X*) ==> *gfp f* ≤ *X*
by (*auto simp add*: *gfp-def intro*: *Sup-least*)

lemma *gfp-lemma2*: *mono f* ==> *gfp f* ≤ *f* (*gfp f*)
by (*iprover intro*: *gfp-least order-trans monoD gfp-upperbound*)

lemma *gfp-lemma3*: *mono f* ==> *f* (*gfp f*) ≤ *gfp f*
by (*iprover intro*: *gfp-lemma2 monoD gfp-upperbound*)

lemma *gfp-unfold*: *mono f* ==> *gfp f* = *f* (*gfp f*)
by (*iprover intro*: *order-antisym gfp-lemma2 gfp-lemma3*)

9.5 Coinduction rules for greatest fixed points

weak version

lemma *weak-coinduct*: [| *a*: *X*; *X* ⊆ *f*(*X*) |] ==> *a* : *gfp*(*f*)
by (*rule gfp-upperbound* [*THEN subsetD*], *auto*)

lemma *weak-coinduct-image*: !!*X*. [| *a* : *X*; *g*‘*X* ⊆ *f* (*g*‘*X*) |] ==> *g a* : *gfp f*
apply (*erule gfp-upperbound* [*THEN subsetD*])
apply (*erule imageI*)
done

lemma *coinduct-lemma*:
[| *X* ≤ *f* (*sup X* (*gfp f*)); *mono f* |] ==> *sup X* (*gfp f*) ≤ *f* (*sup X* (*gfp f*))
apply (*frule gfp-lemma2*)
apply (*drule mono-sup*)
apply (*rule le-supI*)
apply *assumption*
apply (*rule order-trans*)
apply (*rule order-trans*)
apply *assumption*
apply (*rule sup-ge2*)
apply *assumption*
done

strong version, thanks to Coen and Frost

lemma *coinduct-set*: [| *mono*(*f*); *a*: *X*; *X* ⊆ *f*(*X Un gfp*(*f*)) |] ==> *a* : *gfp*(*f*)
by (*blast intro*: *weak-coinduct* [*OF - coinduct-lemma*, *simplified sup-set-eq*])

lemma *coinduct*: [| *mono*(*f*); *X* ≤ *f* (*sup X* (*gfp f*)) |] ==> *X* ≤ *gfp*(*f*)
apply (*rule order-trans*)
apply (*rule sup-ge1*)
apply (*erule gfp-upperbound* [*OF coinduct-lemma*])
apply *assumption*
done

lemma *gfp-fun-UnI2*: [| *mono*(*f*); *a*: *gfp*(*f*) |] ==> *a*: *f*(*X Un gfp*(*f*))
by (*blast dest*: *gfp-lemma2 mono-Un*)

9.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition *X* ⊆ *f X* to one expressed using both *lfp* and *gfp*

lemma *coinduct3-mono-lemma*: *mono*(*f*) ==> *mono*(%*x*. *f*(*x*) *Un X Un B*)
by (*iprover intro*: *subset-refl monoI Un-mono monoD*)

lemma *coinduct3-lemma*:
[| *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*))); *mono*(*f*) |]
==> *lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*)) ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*)))
apply (*rule subset-trans*)

apply (*erule coinduct3-mono-lemma* [*THEN lfp-lemma3*])
apply (*rule Un-least* [*THEN Un-least*])
apply (*rule subset-refl*, *assumption*)
apply (*rule gfp-unfold* [*THEN equalityD1*, *THEN subset-trans*], *assumption*)
apply (*rule monoD* [**where** *f*=*f*], *assumption*)
apply (*subst coinduct3-mono-lemma* [*THEN lfp-unfold*], *auto*)
done

lemma *coinduct3*:
[| *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*))) |] ==> *a* : *gfp*(*f*)
apply (*rule coinduct3-lemma* [*THEN* [*2*] *weak-coinduct*])
apply (*rule coinduct3-mono-lemma* [*THEN lfp-unfold*, *THEN ssubst*], *auto*)
done

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding

lemma *def-gfp-unfold*: [| *A*==*gfp*(*f*); *mono*(*f*) |] ==> *A* = *f*(*A*)
by (*auto intro*!: *gfp-unfold*)

lemma *def-coinduct*:
[| *A*==*gfp*(*f*); *mono*(*f*); *X* ≤ *f*(*sup X A*) |] ==> *X* ≤ *A*
by (*iprover intro*!: *coinduct*)

lemma *def-coinduct-set*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*X Un A*) |] ==> *a*: *A*
by (*auto intro*!: *coinduct-set*)

lemma *def-Collect-coinduct*:
[| *A* == *gfp*(%*w*. *Collect*(*P*(*w*))); *mono*(%*w*. *Collect*(*P*(*w*)));
a: *X*; !!*z*. *z*: *X* ==> *P* (*X Un A*) *z* |] ==>
a : *A*
apply (*erule def-coinduct-set*, *auto*)
done

lemma *def-coinduct3*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un A*)) |] ==> *a*: *A*
by (*auto intro*!: *coinduct3*)

Monotonicity of *gfp*!

lemma *gfp-mono*: (!!*Z*. *f Z* ≤ *g Z*) ==> *gfp f* ≤ *gfp g*
by (*rule gfp-upperbound* [*THEN gfp-least*], *blast intro*: *order-trans*)

9.7 Inductive predicates and sets

Inversion of injective functions.

constdefs
myinv :: ('*a* => '*b*) => ('*b* => '*a*)
myinv (*f* :: '*a* => '*b*) == λ*y*. *THE x*. *f x* = *y*

lemma *myinv-f-f*: *inj f* ==> *myinv f* (*f x*) = *x*
proof −
assume *inj f*
hence (*THE x'*. *f x'* = *f x*) = (*THE x'*. *x'* = *x*)
by (*simp only*: *inj-eq*)
also have ... = *x* **by** (*rule the-eq-trivial*)
finally show *?thesis* **by** (*unfold myinv-def*)
qed

lemma *f-myinv-f*: *inj f* ==> *y* ∈ *range f* ==> *f* (*myinv f y*) = *y*
proof (*unfold myinv-def*)
assume *inj*: *inj f*
assume *y* ∈ *range f*
then obtain *x* **where** *y* = *f x* **..**
hence *x*: *f x* = *y* **..**
thus *f* (*THE x*. *f x* = *y*) = *y*
proof (*rule theI*)
fix *x'* **assume** *f x'* = *y*
with *x* **have** *f x'* = *f x* **by** *simp*
with *inj* **show** *x'* = *x* **by** (*rule injD*)
qed
qed

hide *const myinv*

Package setup.

theorems *basic-monos* =
subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
Collect-mono in-mono vimage-mono
imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
not-all not-ex
Ball-def Bex-def
induct-rulify-fallback

ML ⟪
val def-lfp-unfold = @{*thm def-lfp-unfold*}
val def-gfp-unfold = @{*thm def-gfp-unfold*}
val def-lfp-induct = @{*thm def-lfp-induct*}
val def-coinduct = @{*thm def-coinduct*}
val inf-bool-eq = @{*thm inf-bool-eq*} *RS* @{*thm eq-reflection*}
val inf-fun-eq = @{*thm inf-fun-eq*} *RS* @{*thm eq-reflection*}
val sup-bool-eq = @{*thm sup-bool-eq*} *RS* @{*thm eq-reflection*}
val sup-fun-eq = @{*thm sup-fun-eq*} *RS* @{*thm eq-reflection*}
val le-boolI = @{*thm le-boolI*}
val le-boolI' = @{*thm le-boolI'*}
val le-funI = @{*thm le-funI*}
val le-boolE = @{*thm le-boolE*}
val le-funE = @{*thm le-funE*}

val le-boolD = @{thm le-boolD}
val le-funD = @{thm le-funD}
val le-bool-def = @{thm le-bool-def} RS @{thm eq-reflection}
val le-fun-def = @{thm le-fun-def} RS @{thm eq-reflection}
⟩⟩

use *Tools/inductive-package.ML*
setup *InductivePackage.setup*

theorems [*mono*] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
not-all not-ex
Ball-def Bex-def
induct-rulify-fallback

9.8 Inductive datatypes and primitive recursion

Package setup.

use *Tools/datatype-aux.ML*
use *Tools/datatype-prop.ML*
use *Tools/datatype-rep-proofs.ML*
use *Tools/datatype-abs-proofs.ML*
use *Tools/datatype-case.ML*
use *Tools/datatype-package.ML*
setup *DatatypePackage.setup*
use *Tools/old-primrec-package.ML*
use *Tools/primrec-package.ML*

use *Tools/datatype-codegen.ML*
setup *DatatypeCodegen.setup*

use *Tools/inductive-codegen.ML*
setup *InductiveCodegen.setup*

Lambda-abstractions with pattern matching:

syntax
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((%-) *10*)
syntax (*xsymbols*)
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((λ-) *10*)

parse-translation (**advanced**) ⟪⟪
let
fun fun-tr ctxt [*cs*] =
let
val x = Free (Name.variant (Term.add-free-names cs []*) x, dummyT)*;
val ft = DatatypeCase.case-tr true DatatypePackage.datatype-of-constr
ctxt [*x*, *cs*]
in lambda x ft end

in [(*-lam-pats-syntax*, *fun-tr*)] *end*
⟫

end

10 OrderedGroup: Ordered Groups

theory *OrderedGroup*
imports *Lattices*
uses ~~/*src*/*Provers*/*Arith*/*abel-cancel.ML*
begin

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

ML⟪
structure AlgebraSimps =
  NamedThmsFun(*val name* = *algebra-simps*
    val description = *algebra simplification rules*);
⟫

setup *AlgebraSimps.setup*

The rewrites accumulated in *algebra-simps* deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by *field-simps*.

10.1 Semigroups and Monoids

class *semigroup-add* = *plus* +
  assumes *add-assoc*[*algebra-simps*]: $(a + b) + c = a + (b + c)$

class *ab-semigroup-add* = *semigroup-add* +

assumes *add-commute*[*algebra-simps*]: $a + b = b + a$
begin

lemma *add-left-commute*[*algebra-simps*]: $a + (b + c) = b + (a + c)$
by (*rule mk-left-commute* [*of plus*, *OF add-assoc add-commute*])

theorems *add-ac* = *add-assoc add-commute add-left-commute*

end

theorems *add-ac* = *add-assoc add-commute add-left-commute*

class *semigroup-mult* = *times* +
assumes *mult-assoc*[*algebra-simps*]: $(a * b) * c = a * (b * c)$

class *ab-semigroup-mult* = *semigroup-mult* +
assumes *mult-commute*[*algebra-simps*]: $a * b = b * a$
begin

lemma *mult-left-commute*[*algebra-simps*]: $a * (b * c) = b * (a * c)$
by (*rule mk-left-commute* [*of times*, *OF mult-assoc mult-commute*])

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

end

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

class *ab-semigroup-idem-mult* = *ab-semigroup-mult* +
assumes *mult-idem*[*simp*]: $x * x = x$
begin

lemma *mult-left-idem*[*simp*]: $x * (x * y) = x * y$
unfolding *mult-assoc* [*symmetric*, *of x*] *mult-idem* **..**

end

class *monoid-add* = *zero* + *semigroup-add* +
assumes *add-0-left* [*simp*]: $0 + a = a$
and *add-0-right* [*simp*]: $a + 0 = a$

lemma *zero-reorient*: $0 = x \longleftrightarrow x = 0$
by (*rule eq-commute*)

class *comm-monoid-add* = *zero* + *ab-semigroup-add* +
assumes *add-0*: $0 + a = a$
begin

subclass *monoid-add*

proof qed (*insert add-0*, *simp-all add*: *add-commute*)

end

class *monoid-mult* = *one* + *semigroup-mult* +
assumes *mult-1-left* [*simp*]: *1* ∗ *a* = *a*
assumes *mult-1-right* [*simp*]: *a* ∗ *1* = *a*

lemma *one-reorient*: *1* = *x* ⟷ *x* = *1*
by (*rule eq-commute*)

class *comm-monoid-mult* = *one* + *ab-semigroup-mult* +
assumes *mult-1*: *1* ∗ *a* = *a*
begin

subclass *monoid-mult*
proof qed (*insert mult-1*, *simp-all add*: *mult-commute*)

end

class *cancel-semigroup-add* = *semigroup-add* +
assumes *add-left-imp-eq*: *a* + *b* = *a* + *c* ⟹ *b* = *c*
assumes *add-right-imp-eq*: *b* + *a* = *c* + *a* ⟹ *b* = *c*
begin

lemma *add-left-cancel* [*simp*]:
a + *b* = *a* + *c* ⟷ *b* = *c*
by (*blast dest*: *add-left-imp-eq*)

lemma *add-right-cancel* [*simp*]:
b + *a* = *c* + *a* ⟷ *b* = *c*
by (*blast dest*: *add-right-imp-eq*)

end

class *cancel-ab-semigroup-add* = *ab-semigroup-add* +
assumes *add-imp-eq*: *a* + *b* = *a* + *c* ⟹ *b* = *c*
begin

subclass *cancel-semigroup-add*
proof
fix *a b c* :: *'a*
assume *a* + *b* = *a* + *c*
then show *b* = *c* **by** (*rule add-imp-eq*)
next
fix *a b c* :: *'a*
assume *b* + *a* = *c* + *a*
then have *a* + *b* = *a* + *c* **by** (*simp only*: *add-commute*)
then show *b* = *c* **by** (*rule add-imp-eq*)

qed

end

class *cancel-comm-monoid-add* = *cancel-ab-semigroup-add* + *comm-monoid-add*

10.2 Groups

class *group-add* = *minus* + *uminus* + *monoid-add* +
  assumes *left-minus* [*simp*]: − *a* + *a* = *0*
  assumes *diff-minus*: *a* − *b* = *a* + (− *b*)
begin

lemma *minus-add-cancel*: − *a* + (*a* + *b*) = *b*
by (*simp add*: *add-assoc*[*symmetric*])

lemma *minus-zero* [*simp*]: − *0* = *0*
proof −
  have − *0* = − *0* + (*0* + *0*) **by** (*simp only*: *add-0-right*)
  also have ... = *0* **by** (*rule minus-add-cancel*)
  finally show *?thesis* **.**
qed

lemma *minus-minus* [*simp*]: − (− *a*) = *a*
proof −
  have − (− *a*) = − (− *a*) + (− *a* + *a*) **by** *simp*
  also have ... = *a* **by** (*rule minus-add-cancel*)
  finally show *?thesis* **.**
qed

lemma *right-minus* [*simp*]: *a* + − *a* = *0*
proof −
  have *a* + − *a* = − (− *a*) + − *a* **by** *simp*
  also have ... = *0* **by** (*rule left-minus*)
  finally show *?thesis* **.**
qed

lemma *right-minus-eq*: *a* − *b* = *0* ⟷ *a* = *b*
proof
  assume *a* − *b* = *0*
  have *a* = (*a* − *b*) + *b* **by** (*simp add*:*diff-minus add-assoc*)
  also have ... = *b* **using** ‹*a* − *b* = *0*› **by** *simp*
  finally show *a* = *b* **.**
next
  assume *a* = *b* **thus** *a* − *b* = *0* **by** (*simp add*: *diff-minus*)
qed

lemma *equals-zero-I*:
  assumes *a* + *b* = *0* **shows** − *a* = *b*

proof −
have − *a* = − *a* + (*a* + *b*) **using** *assms* **by** *simp*
also have ... = *b* **by** (*simp add*: *add-assoc*[*symmetric*])
finally show *?thesis* **.**
qed

lemma *diff-self* [*simp*]: *a* − *a* = *0*
by (*simp add*: *diff-minus*)

lemma *diff-0* [*simp*]: *0* − *a* = − *a*
by (*simp add*: *diff-minus*)

lemma *diff-0-right* [*simp*]: *a* − *0* = *a*
by (*simp add*: *diff-minus*)

lemma *diff-minus-eq-add* [*simp*]: *a* − − *b* = *a* + *b*
by (*simp add*: *diff-minus*)

lemma *neg-equal-iff-equal* [*simp*]:
− *a* = − *b* ⟷ *a* = *b*
proof
assume − *a* = − *b*
hence − (− *a*) = − (− *b*) **by** *simp*
thus *a* = *b* **by** *simp*
next
assume *a* = *b*
thus − *a* = − *b* **by** *simp*
qed

lemma *neg-equal-0-iff-equal* [*simp*]:
− *a* = *0* ⟷ *a* = *0*
by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

lemma *neg-0-equal-iff-equal* [*simp*]:
0 = − *a* ⟷ *0* = *a*
by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*:
a = − *b* ⟷ *b* = − *a*
proof −
have − (− *a*) = − *b* ⟷ − *a* = *b* **by** (*rule neg-equal-iff-equal*)
thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

lemma *minus-equation-iff*:
− *a* = *b* ⟷ − *b* = *a*
proof −
have − *a* = − (− *b*) ⟷ *a* = −*b* **by** (*rule neg-equal-iff-equal*)

thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

lemma *diff-add-cancel*: $a - b + b = a$
by (*simp add*: *diff-minus add-assoc*)

lemma *add-diff-cancel*: $a + b - b = a$
by (*simp add*: *diff-minus add-assoc*)

declare *diff-minus*[*symmetric*, *algebra-simps*]

lemma *eq-neg-iff-add-eq-0*: $a = - b \longleftrightarrow a + b = 0$
proof
assume $a = - b$ **then show** $a + b = 0$ **by** *simp*
next
assume $a + b = 0$
moreover have $a + (b + - b) = (a + b) + - b$
by (*simp only*: *add-assoc*)
ultimately show $a = - b$ **by** *simp*
qed

end

class *ab-group-add* = *minus* + *uminus* + *comm-monoid-add* +
assumes *ab-left-minus*: $- a + a = 0$
assumes *ab-diff-minus*: $a - b = a + (- b)$
begin

subclass *group-add*
proof qed (*simp-all add*: *ab-left-minus ab-diff-minus*)

subclass *cancel-comm-monoid-add*
proof
fix $a\ b\ c :: 'a$
assume $a + b = a + c$
then have $- a + a + b = - a + a + c$
unfolding *add-assoc* **by** *simp*
then show $b = c$ **by** *simp*
qed

lemma *uminus-add-conv-diff*[*algebra-simps*]:
$- a + b = b - a$
by (*simp add*:*diff-minus add-commute*)

lemma *minus-add-distrib* [*simp*]:
$- (a + b) = - a + - b$
by (*rule equals-zero-I*) (*simp add*: *add-ac*)

lemma *minus-diff-eq* [*simp*]:

$-(a - b) = b - a$
by (*simp add*: *diff-minus add-commute*)

lemma *add-diff-eq*[*algebra-simps*]: $a + (b - c) = (a + b) - c$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-add-eq*[*algebra-simps*]: $(a - b) + c = (a + c) - b$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-eq-eq*[*algebra-simps*]: $a - b = c \longleftrightarrow a = c + b$
by (*auto simp add*: *diff-minus add-assoc*)

lemma *eq-diff-eq*[*algebra-simps*]: $a = c - b \longleftrightarrow a + b = c$
by (*auto simp add*: *diff-minus add-assoc*)

lemma *diff-diff-eq*[*algebra-simps*]: $(a - b) - c = a - (b + c)$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-diff-eq2*[*algebra-simps*]: $a - (b - c) = (a + c) - b$
by (*simp add*: *diff-minus add-ac*)

lemma *eq-iff-diff-eq-0*: $a = b \longleftrightarrow a - b = 0$
by (*simp add*: *algebra-simps*)

lemma *diff-eq-0-iff-eq* [*simp*, *noatp*]: $a - b = 0 \longleftrightarrow a = b$
by (*simp add*: *algebra-simps*)

end

10.3 (Partially) Ordered Groups

class *pordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +
  assumes *add-left-mono*: $a \leq b \Longrightarrow c + a \leq c + b$
begin

lemma *add-right-mono*:
  $a \leq b \Longrightarrow a + c \leq b + c$
by (*simp add*: *add-commute* [*of* - *c*] *add-left-mono*)

non-strict, in both arguments

lemma *add-mono*:
  $a \leq b \Longrightarrow c \leq d \Longrightarrow a + c \leq b + d$
  apply (*erule add-right-mono* [*THEN order-trans*])
  apply (*simp add*: *add-commute add-left-mono*)
  done

end

class *pordered-cancel-ab-semigroup-add* =

pordered-ab-semigroup-add + *cancel-ab-semigroup-add*
begin

lemma *add-strict-left-mono*:
$a < b \Longrightarrow c + a < c + b$
by (*auto simp add*: *less-le add-left-mono*)

lemma *add-strict-right-mono*:
$a < b \Longrightarrow a + c < b + c$
by (*simp add*: *add-commute* [*of* - *c*] *add-strict-left-mono*)

Strict monotonicity in both arguments

lemma *add-strict-mono*:
$a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$
apply (*erule add-strict-right-mono* [*THEN less-trans*])
apply (*erule add-strict-left-mono*)
done

lemma *add-less-le-mono*:
$a < b \Longrightarrow c \leq d \Longrightarrow a + c < b + d$
apply (*erule add-strict-right-mono* [*THEN less-le-trans*])
apply (*erule add-left-mono*)
done

lemma *add-le-less-mono*:
$a \leq b \Longrightarrow c < d \Longrightarrow a + c < b + d$
apply (*erule add-right-mono* [*THEN le-less-trans*])
apply (*erule add-strict-left-mono*)
done

end

class *pordered-ab-semigroup-add-imp-le* =
pordered-cancel-ab-semigroup-add +
assumes *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
begin

lemma *add-less-imp-less-left*:
assumes *less*: $c + a < c + b$ **shows** $a < b$
proof −
from *less* **have** *le*: $c + a <= c + b$ **by** (*simp add*: *order-le-less*)
have $a <= b$
apply (*insert le*)
apply (*drule add-le-imp-le-left*)
by (*insert le*, *drule add-le-imp-le-left*, *assumption*)
moreover have $a \neq b$
proof (*rule ccontr*)
assume $\sim(a \neq b)$
then have $a = b$ **by** *simp*

```
    then have c + a = c + b by simp
    with less show Falseby simp
  qed
  ultimately show a < b by (simp add: order-le-less)
qed

lemma add-less-imp-less-right:
  a + c < b + c ⟹ a < b
apply (rule add-less-imp-less-left [of c])
apply (simp add: add-commute)
done

lemma add-less-cancel-left [simp]:
  c + a < c + b ⟷ a < b
by (blast intro: add-less-imp-less-left add-strict-left-mono)

lemma add-less-cancel-right [simp]:
  a + c < b + c ⟷ a < b
by (blast intro: add-less-imp-less-right add-strict-right-mono)

lemma add-le-cancel-left [simp]:
  c + a ≤ c + b ⟷ a ≤ b
by (auto, drule add-le-imp-le-left, simp-all add: add-left-mono)

lemma add-le-cancel-right [simp]:
  a + c ≤ b + c ⟷ a ≤ b
by (simp add: add-commute [of a c] add-commute [of b c])

lemma add-le-imp-le-right:
  a + c ≤ b + c ⟹ a ≤ b
by simp

lemma max-add-distrib-left:
  max x y + z = max (x + z) (y + z)
  unfolding max-def by auto

lemma min-add-distrib-left:
  min x y + z = min (x + z) (y + z)
  unfolding min-def by auto

end
```

10.4 Support for reasoning about signs

```
class pordered-comm-monoid-add =
  pordered-cancel-ab-semigroup-add + comm-monoid-add
begin

lemma add-pos-nonneg:
```

assumes $0 < a$ **and** $0 \leq b$ **shows** $0 < a + b$
proof −
have $0 + 0 < a + b$
using *assms* **by** (*rule add-less-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-pos-pos*:
assumes $0 < a$ **and** $0 < b$ **shows** $0 < a + b$
by (*rule add-pos-nonneg*) (*insert assms*, *auto*)

lemma *add-nonneg-pos*:
assumes $0 \leq a$ **and** $0 < b$ **shows** $0 < a + b$
proof −
have $0 + 0 < a + b$
using *assms* **by** (*rule add-le-less-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonneg-nonneg*:
assumes $0 \leq a$ **and** $0 \leq b$ **shows** $0 \leq a + b$
proof −
have $0 + 0 \leq a + b$
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-neg-nonpos*:
assumes $a < 0$ **and** $b \leq 0$ **shows** $a + b < 0$
proof −
have $a + b < 0 + 0$
using *assms* **by** (*rule add-less-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-neg-neg*:
assumes $a < 0$ **and** $b < 0$ **shows** $a + b < 0$
by (*rule add-neg-nonpos*) (*insert assms*, *auto*)

lemma *add-nonpos-neg*:
assumes $a \leq 0$ **and** $b < 0$ **shows** $a + b < 0$
proof −
have $a + b < 0 + 0$
using *assms* **by** (*rule add-le-less-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonpos-nonpos*:
assumes $a \leq 0$ **and** $b \leq 0$ **shows** $a + b \leq 0$

proof −
have $a + b \leq 0 + 0$
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemmas *add-sign-intros* =
add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

lemma *add-nonneg-eq-0-iff*:
assumes x: $0 \leq x$ **and** y: $0 \leq y$
shows $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
proof (*intro iffI conjI*)
have $x = x + 0$ **by** *simp*
also have $x + 0 \leq x + y$ **using** y **by** (*rule add-left-mono*)
also assume $x + y = 0$
also have $0 \leq x$ **using** x **.**
finally show $x = 0$ **.**
next
have $y = 0 + y$ **by** *simp*
also have $0 + y \leq x + y$ **using** x **by** (*rule add-right-mono*)
also assume $x + y = 0$
also have $0 \leq y$ **using** y **.**
finally show $y = 0$ **.**
next
assume $x = 0 \wedge y = 0$
then show $x + y = 0$ **by** *simp*
qed

end

class *pordered-ab-group-add* =
ab-group-add + *pordered-ab-semigroup-add*
begin

subclass *pordered-cancel-ab-semigroup-add* **..**

subclass *pordered-ab-semigroup-add-imp-le*
proof
fix a b c :: $'a$
assume $c + a \leq c + b$
hence $(-c) + (c + a) \leq (-c) + (c + b)$ **by** (*rule add-left-mono*)
hence $((-c) + c) + a \leq ((-c) + c) + b$ **by** (*simp only*: *add-assoc*)
thus $a \leq b$ **by** *simp*
qed

subclass *pordered-comm-monoid-add* **..**

lemma *max-diff-distrib-left*:
  shows $max\ x\ y - z = max\ (x - z)\ (y - z)$
by (*simp add*: *diff-minus*, *rule max-add-distrib-left*)

lemma *min-diff-distrib-left*:
  shows $min\ x\ y - z = min\ (x - z)\ (y - z)$
by (*simp add*: *diff-minus*, *rule min-add-distrib-left*)

lemma *le-imp-neg-le*:
  assumes $a \leq b$ **shows** $-b \leq -a$
proof −
  have $-a+a \leq -a+b$ **using** ‹$a \leq b$› **by** (*rule add-left-mono*)
  hence $0 \leq -a+b$ **by** *simp*
  hence $0 + (-b) \leq (-a + b) + (-b)$ **by** (*rule add-right-mono*)
  thus *?thesis* **by** (*simp add*: *add-assoc*)
qed

lemma *neg-le-iff-le* [*simp*]: $- b \leq - a \longleftrightarrow a \leq b$
proof
  assume $- b \leq - a$
  hence $- (- a) \leq - (- b)$ **by** (*rule le-imp-neg-le*)
  thus $a \leq b$ **by** *simp*
next
  assume $a \leq b$
  thus $-b \leq -a$ **by** (*rule le-imp-neg-le*)
qed

lemma *neg-le-0-iff-le* [*simp*]: $- a \leq 0 \longleftrightarrow 0 \leq a$
by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-0-le-iff-le* [*simp*]: $0 \leq - a \longleftrightarrow a \leq 0$
by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-less-iff-less* [*simp*]: $- b < - a \longleftrightarrow a < b$
by (*force simp add*: *less-le*)

lemma *neg-less-0-iff-less* [*simp*]: $- a < 0 \longleftrightarrow 0 < a$
by (*subst neg-less-iff-less* [*symmetric*], *simp*)

lemma *neg-0-less-iff-less* [*simp*]: $0 < - a \longleftrightarrow a < 0$
by (*subst neg-less-iff-less* [*symmetric*], *simp*)

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < - b \longleftrightarrow b < - a$
proof −
  have $(- (-a) < - b) = (b < - a)$ **by** (*rule neg-less-iff-less*)
  thus *?thesis* **by** *simp*
qed

lemma *minus-less-iff*: $- a < b \longleftrightarrow - b < a$
proof −
have $(- a < - (-b)) = (- b < a)$ **by** (*rule neg-less-iff-less*)
thus *?thesis* **by** *simp*
qed

lemma *le-minus-iff*: $a \leq - b \longleftrightarrow b \leq - a$
proof −
have *mm*: !! a (b::'a). $(-(-a)) < -b \Longrightarrow -(-b) < -a$ **by** (*simp only*: *minus-less-iff*)
have $(- (- a) <= -b) = (b <= - a)$
apply (*auto simp only*: *le-less*)
apply (*drule mm*)
apply (*simp-all*)
apply (*drule mm*[*simplified*], *assumption*)
done
then show *?thesis* **by** *simp*
qed

lemma *minus-le-iff*: $- a \leq b \longleftrightarrow - b \leq a$
by (*auto simp add*: *le-less minus-less-iff*)

lemma *less-iff-diff-less-0*: $a < b \longleftrightarrow a - b < 0$
proof −
have $(a < b) = (a + (- b) < b + (-b))$
by (*simp only*: *add-less-cancel-right*)
also have ... = $(a - b < 0)$ **by** (*simp add*: *diff-minus*)
finally show *?thesis* **.**
qed

lemma *diff-less-eq*[*algebra-simps*]: $a - b < c \longleftrightarrow a < c + b$
apply (*subst less-iff-diff-less-0* [*of a*])
apply (*rule less-iff-diff-less-0* [*of* - *c*, *THEN ssubst*])
apply (*simp add*: *diff-minus add-ac*)
done

lemma *less-diff-eq*[*algebra-simps*]: $a < c - b \longleftrightarrow a + b < c$
apply (*subst less-iff-diff-less-0* [*of plus a b*])
apply (*subst less-iff-diff-less-0* [*of a*])
apply (*simp add*: *diff-minus add-ac*)
done

lemma *diff-le-eq*[*algebra-simps*]: $a - b \leq c \longleftrightarrow a \leq c + b$
by (*auto simp add*: *le-less diff-less-eq diff-add-cancel add-diff-cancel*)

lemma *le-diff-eq*[*algebra-simps*]: $a \leq c - b \longleftrightarrow a + b \leq c$
by (*auto simp add*: *le-less less-diff-eq diff-add-cancel add-diff-cancel*)

lemma *le-iff-diff-le-0*: $a \leq b \longleftrightarrow a - b \leq 0$
by (*simp add*: *algebra-simps*)

Legacy - use *algebra-simps*

lemmas *group-simps*[*noatp*] = *algebra-simps*

end

Legacy - use *algebra-simps*

lemmas *group-simps*[*noatp*] = *algebra-simps*

class *ordered-ab-semigroup-add* =
  linorder + *pordered-ab-semigroup-add*

class *ordered-cancel-ab-semigroup-add* =
  linorder + *pordered-cancel-ab-semigroup-add*
begin

subclass *ordered-ab-semigroup-add* **..**

subclass *pordered-ab-semigroup-add-imp-le*
proof
  fix *a b c* :: *'a*
  assume *le*: *c* + *a* <= *c* + *b*
  show *a* <= *b*
  proof (*rule ccontr*)
    assume *w*: ~ *a* ≤ *b*
    hence *b* <= *a* **by** (*simp add*: *linorder-not-le*)
    hence *le2*: *c* + *b* <= *c* + *a* **by** (*rule add-left-mono*)
    have *a* = *b*
      apply (*insert le*)
      apply (*insert le2*)
      apply (*drule antisym*, *simp-all*)
      done
    with *w* **show** *False*
      by (*simp add*: *linorder-not-le* [*symmetric*])
  qed
qed

end

class *ordered-ab-group-add* =
  linorder + *pordered-ab-group-add*
begin

subclass *ordered-cancel-ab-semigroup-add* **..**

lemma *neg-less-eq-nonneg*:
  − *a* ≤ *a* ⟷ *0* ≤ *a*
proof
  assume *A*: − *a* ≤ *a* **show** *0* ≤ *a*
  proof (*rule classical*)

```
    assume ¬ 0 ≤ a
    then have a < 0 by auto
    with A have − a < 0 by (rule le-less-trans)
    then show ?thesis by auto
  qed
next
  assume A: 0 ≤ a show − a ≤ a
  proof (rule order-trans)
    show − a ≤ 0 using A by (simp add: minus-le-iff)
  next
    show 0 ≤ a using A .
  qed
qed

lemma less-eq-neg-nonpos:
  a ≤ − a ⟷ a ≤ 0
proof
  assume A: a ≤ − a show a ≤ 0
  proof (rule classical)
    assume ¬ a ≤ 0
    then have 0 < a by auto
    then have 0 < − a using A by (rule less-le-trans)
    then show ?thesis by auto
  qed
next
  assume A: a ≤ 0 show a ≤ − a
  proof (rule order-trans)
    show 0 ≤ − a using A by (simp add: minus-le-iff)
  next
    show a ≤ 0 using A .
  qed
qed

lemma equal-neg-zero:
  a = − a ⟷ a = 0
proof
  assume a = 0 then show a = − a by simp
next
  assume A: a = − a show a = 0
  proof (cases 0 ≤ a)
    case True with A have 0 ≤ − a by auto
    with le-minus-iff have a ≤ 0 by simp
    with True show ?thesis by (auto intro: order-trans)
  next
    case False then have B: a ≤ 0 by auto
    with A have − a ≤ 0 by auto
    with B show ?thesis by (auto intro: order-trans)
  qed
qed
```

lemma *neg-equal-zero*:
  $-\ a = a \longleftrightarrow a = 0$
  unfolding *equal-neg-zero* [*symmetric*] **by** *auto*

end

— FIXME localize the following

lemma *add-increasing*:
  fixes $c :: 'a::\{$*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*$\}$
  shows $[|0 \leq a;\ b \leq c|] ==> b \leq a + c$
by (*insert add-mono* [*of 0 a b c*], *simp*)

lemma *add-increasing2*:
  fixes $c :: 'a::\{$*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*$\}$
  shows $[|0 \leq c;\ b \leq a|] ==> b \leq a + c$
by (*simp add*:*add-increasing add-commute*[*of a*])

lemma *add-strict-increasing*:
  fixes $c :: 'a::\{$*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*$\}$
  shows $[|0 < a;\ b \leq c|] ==> b < a + c$
by (*insert add-less-le-mono* [*of 0 a b c*], *simp*)

lemma *add-strict-increasing2*:
  fixes $c :: 'a::\{$*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*$\}$
  shows $[|0 \leq a;\ b < c|] ==> b < a + c$
by (*insert add-le-less-mono* [*of 0 a b c*], *simp*)

class *pordered-ab-group-add-abs* = *pordered-ab-group-add* + *abs* +
  assumes *abs-ge-zero* [*simp*]: $|a| \geq 0$
    and *abs-ge-self*: $a \leq |a|$
    and *abs-leI*: $a \leq b \Longrightarrow -\ a \leq b \Longrightarrow |a| \leq b$
    and *abs-minus-cancel* [*simp*]: $|-a| = |a|$
    and *abs-triangle-ineq*: $|a + b| \leq |a| + |b|$
begin

lemma *abs-minus-le-zero*: $-\ |a| \leq 0$
  unfolding *neg-le-0-iff-le* **by** *simp*

lemma *abs-of-nonneg* [*simp*]:
  assumes *nonneg*: $0 \leq a$ **shows** $|a| = a$
proof (*rule antisym*)
  from *nonneg le-imp-neg-le* **have** $-\ a \leq 0$ **by** *simp*
  from *this nonneg* **have** $-\ a \leq a$ **by** (*rule order-trans*)
  then show $|a| \leq a$ **by** (*auto intro*: *abs-leI*)
qed (*rule abs-ge-self*)

lemma *abs-idempotent* [*simp*]: $||a|| = |a|$
by (*rule antisym*)
(*auto intro*!: *abs-ge-self abs-leI order-trans* [*of uminus* (*abs a*) *zero abs a*])

lemma *abs-eq-0* [*simp*]: $|a| = 0 \longleftrightarrow a = 0$
proof −
have $|a| = 0 \Longrightarrow a = 0$
proof (*rule antisym*)
assume *zero*: $|a| = 0$
with *abs-ge-self* **show** $a \leq 0$ **by** *auto*
from *zero* **have** $|-a| = 0$ **by** *simp*
with *abs-ge-self* [*of uminus a*] **have** $- a \leq 0$ **by** *auto*
with *neg-le-0-iff-le* **show** $0 \leq a$ **by** *auto*
qed
then show *?thesis* **by** *auto*
qed

lemma *abs-zero* [*simp*]: $|0| = 0$
by *simp*

lemma *abs-0-eq* [*simp*, *noatp*]: $0 = |a| \longleftrightarrow a = 0$
proof −
have $0 = |a| \longleftrightarrow |a| = 0$ **by** (*simp only*: *eq-ac*)
thus *?thesis* **by** *simp*
qed

lemma *abs-le-zero-iff* [*simp*]: $|a| \leq 0 \longleftrightarrow a = 0$
proof
assume $|a| \leq 0$
then have $|a| = 0$ **by** (*rule antisym*) *simp*
thus $a = 0$ **by** *simp*
next
assume $a = 0$
thus $|a| \leq 0$ **by** *simp*
qed

lemma *zero-less-abs-iff* [*simp*]: $0 < |a| \longleftrightarrow a \neq 0$
by (*simp add*: *less-le*)

lemma *abs-not-less-zero* [*simp*]: $\neg\ |a| < 0$
proof −
have *a*: $\bigwedge x\ y.\ x \leq y \Longrightarrow \neg\ y < x$ **by** *auto*
show *?thesis* **by** (*simp add*: *a*)
qed

lemma *abs-ge-minus-self*: $- a \leq |a|$
proof −
have $- a \leq |-a|$ **by** (*rule abs-ge-self*)
then show *?thesis* **by** *simp*

qed

lemma *abs-minus-commute*:
$|a - b| = |b - a|$
proof −
have $|a - b| = |-(a - b)|$ **by** (*simp only*: *abs-minus-cancel*)
also have ... $= |b - a|$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *abs-of-pos*: $0 < a \Longrightarrow |a| = a$
by (*rule abs-of-nonneg*, *rule less-imp-le*)

lemma *abs-of-nonpos* [*simp*]:
assumes $a \leq 0$ **shows** $|a| = - a$
proof −
let *?b* $= - a$
have $- ?b \leq 0 \Longrightarrow |- ?b| = - (- ?b)$
unfolding *abs-minus-cancel* [*of ?b*]
unfolding *neg-le-0-iff-le* [*of ?b*]
unfolding *minus-minus* **by** (*erule abs-of-nonneg*)
then show *?thesis* **using** *assms* **by** *auto*
qed

lemma *abs-of-neg*: $a < 0 \Longrightarrow |a| = - a$
by (*rule abs-of-nonpos*, *rule less-imp-le*)

lemma *abs-le-D1*: $|a| \leq b \Longrightarrow a \leq b$
by (*insert abs-ge-self*, *blast intro*: *order-trans*)

lemma *abs-le-D2*: $|a| \leq b \Longrightarrow - a \leq b$
by (*insert abs-le-D1* [*of uminus a*], *simp*)

lemma *abs-le-iff*: $|a| \leq b \longleftrightarrow a \leq b \wedge - a \leq b$
by (*blast intro*: *abs-leI dest*: *abs-le-D1 abs-le-D2*)

lemma *abs-triangle-ineq2*: $|a| - |b| \leq |a - b|$
apply (*simp add*: *algebra-simps*)
apply (*subgoal-tac abs a = abs* (*plus b* (*minus a b*)))
apply (*erule ssubst*)
apply (*rule abs-triangle-ineq*)
apply (*rule arg-cong*[*of - - abs*])
apply (*simp add*: *algebra-simps*)
done

lemma *abs-triangle-ineq3*: $||a| - |b|| \leq |a - b|$
apply (*subst abs-le-iff*)
apply *auto*
apply (*rule abs-triangle-ineq2*)

apply (*subst abs-minus-commute*)
apply (*rule abs-triangle-ineq2*)
done

lemma *abs-triangle-ineq4*: $|a - b| \leq |a| + |b|$
proof −
have $abs(a - b) = abs(a + - b)$ **by** (*subst diff-minus*, *rule refl*)
also have ... <= *abs a* + *abs* (− *b*) **by** (*rule abs-triangle-ineq*)
finally show *?thesis* **by** *simp*
qed

lemma *abs-diff-triangle-ineq*: $|a + b - (c + d)| \leq |a - c| + |b - d|$
proof −
have $|a + b - (c+d)| = |(a-c) + (b-d)|$ **by** (*simp add*: *diff-minus add-ac*)
also have ... $\leq |a-c| + |b-d|$ **by** (*rule abs-triangle-ineq*)
finally show *?thesis* **.**
qed

lemma *abs-add-abs* [*simp*]:
$||a| + |b|| = |a| + |b|$ (**is** *?L* = *?R*)
proof (*rule antisym*)
show *?L* $\geq$ *?R* **by**(*rule abs-ge-self*)
next
have *?L* $\leq ||a|| + ||b||$ **by**(*rule abs-triangle-ineq*)
also have ... = *?R* **by** *simp*
finally show *?L* $\leq$ *?R* **.**
qed

end

10.5 Lattice Ordered (Abelian) Groups

class *lordered-ab-group-add-meet* = *pordered-ab-group-add* + *lower-semilattice*
begin

lemma *add-inf-distrib-left*:
a + *inf b c* = *inf* (*a* + *b*) (*a* + *c*)
apply (*rule antisym*)
apply (*simp-all add*: *le-infI*)
apply (*rule add-le-imp-le-left* [*of uminus a*])
apply (*simp only*: *add-assoc* [*symmetric*], *simp*)
apply *rule*
apply (*rule add-le-imp-le-left*[*of a*], *simp only*: *add-assoc*[*symmetric*], *simp*)+
done

lemma *add-inf-distrib-right*:
inf a b + *c* = *inf* (*a* + *c*) (*b* + *c*)
proof −
have *c* + *inf a b* = *inf* (*c*+*a*) (*c*+*b*) **by** (*simp add*: *add-inf-distrib-left*)

thus *?thesis* **by** (*simp add*: *add-commute*)
qed

end

class *lordered-ab-group-add-join* = *pordered-ab-group-add* + *upper-semilattice*
begin

lemma *add-sup-distrib-left*:
a + *sup b c* = *sup* (*a* + *b*) (*a* + *c*)
apply (*rule antisym*)
apply (*rule add-le-imp-le-left* [*of uminus a*])
apply (*simp only*: *add-assoc*[*symmetric*], *simp*)
apply *rule*
apply (*rule add-le-imp-le-left* [*of a*], *simp only*: *add-assoc*[*symmetric*], *simp*)+
apply (*rule le-supI*)
apply (*simp-all*)
done

lemma *add-sup-distrib-right*:
sup a b + *c* = *sup* (*a*+*c*) (*b*+*c*)
proof −
have *c* + *sup a b* = *sup* (*c*+*a*) (*c*+*b*) **by** (*simp add*: *add-sup-distrib-left*)
thus *?thesis* **by** (*simp add*: *add-commute*)
qed

end

class *lordered-ab-group-add* = *pordered-ab-group-add* + *lattice*
begin

subclass *lordered-ab-group-add-meet* ..
subclass *lordered-ab-group-add-join* ..

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

lemma *inf-eq-neg-sup*: *inf a b* = − *sup* (−*a*) (−*b*)
proof (*rule inf-unique*)
fix *a b* :: *'a*
show − *sup* (−*a*) (−*b*) ≤ *a*
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
next
fix *a b* :: *'a*
show − *sup* (−*a*) (−*b*) ≤ *b*
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
next

fix $a\ b\ c :: 'a$
assume $a \leq b\ a \leq c$
then show $a \leq -\ sup\ (-b)\ (-c)$ **by** (*subst neg-le-iff-le* [*symmetric*])
(*simp add*: *le-supI*)
qed

lemma *sup-eq-neg-inf*: $sup\ a\ b = -\ inf\ (-a)\ (-b)$
proof (*rule sup-unique*)
fix $a\ b :: 'a$
show $a \leq -\ inf\ (-a)\ (-b)$
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
next
fix $a\ b :: 'a$
show $b \leq -\ inf\ (-a)\ (-b)$
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
next
fix $a\ b\ c :: 'a$
assume $a \leq c\ b \leq c$
then show $-\ inf\ (-a)\ (-b) \leq c$ **by** (*subst neg-le-iff-le* [*symmetric*])
(*simp add*: *le-infI*)
qed

lemma *neg-inf-eq-sup*: $-\ inf\ a\ b = sup\ (-a)\ (-b)$
by (*simp add*: *inf-eq-neg-sup*)

lemma *neg-sup-eq-inf*: $-\ sup\ a\ b = inf\ (-a)\ (-b)$
by (*simp add*: *sup-eq-neg-inf*)

lemma *add-eq-inf-sup*: $a + b = sup\ a\ b + inf\ a\ b$
proof −
have $0 = -\ inf\ 0\ (a-b) + inf\ (a-b)\ 0$ **by** (*simp add*: *inf-commute*)
hence $0 = sup\ 0\ (b-a) + inf\ (a-b)\ 0$ **by** (*simp add*: *inf-eq-neg-sup*)
hence $0 = (-a + sup\ a\ b) + (inf\ a\ b + (-b))$
by (*simp add*: *add-sup-distrib-left add-inf-distrib-right*)
(*simp add*: *algebra-simps*)
thus *?thesis* **by** (*simp add*: *algebra-simps*)
qed

10.6 Positive Part, Negative Part, Absolute Value

definition
$nprt :: 'a \Rightarrow 'a$ **where**
$nprt\ x = inf\ x\ 0$

definition
$pprt :: 'a \Rightarrow 'a$ **where**
$pprt\ x = sup\ x\ 0$

lemma *pprt-neg*: *pprt* (− *x*) = − *nprt x*
proof −
have *sup* (− *x*) *0* = *sup* (− *x*) (− *0*) **unfolding** *minus-zero* **..**
also have ... = − *inf x 0* **unfolding** *neg-inf-eq-sup* **..**
finally have *sup* (− *x*) *0* = − *inf x 0* **.**
then show *?thesis* **unfolding** *pprt-def nprt-def* **.**
qed

lemma *nprt-neg*: *nprt* (− *x*) = − *pprt x*
proof −
from *pprt-neg* **have** *pprt* (− (− *x*)) = − *nprt* (− *x*) **.**
then have *pprt x* = − *nprt* (− *x*) **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *prts*: *a* = *pprt a* + *nprt a*
by (*simp add*: *pprt-def nprt-def add-eq-inf-sup*[*symmetric*])

lemma *zero-le-pprt*[*simp*]: *0* ≤ *pprt a*
by (*simp add*: *pprt-def*)

lemma *nprt-le-zero*[*simp*]: *nprt a* ≤ *0*
by (*simp add*: *nprt-def*)

lemma *le-eq-neg*: *a* ≤ − *b* ⟷ *a* + *b* ≤ *0* (**is** *?l* = *?r*)
proof −
have *a*: *?l* ⟶ *?r*
apply (*auto*)
apply (*rule add-le-imp-le-right*[*of* - *uminus b* -])
apply (*simp add*: *add-assoc*)
done
have *b*: *?r* ⟶ *?l*
apply (*auto*)
apply (*rule add-le-imp-le-right*[*of* - *b* -])
apply (*simp*)
done
from *a b* **show** *?thesis* **by** *blast*
qed

lemma *pprt-0*[*simp*]: *pprt 0* = *0* **by** (*simp add*: *pprt-def*)
lemma *nprt-0*[*simp*]: *nprt 0* = *0* **by** (*simp add*: *nprt-def*)

lemma *pprt-eq-id* [*simp*, *noatp*]: *0* ≤ *x* ⟹ *pprt x* = *x*
by (*simp add*: *pprt-def le-iff-sup sup-ACI*)

lemma *nprt-eq-id* [*simp*, *noatp*]: *x* ≤ *0* ⟹ *nprt x* = *x*
by (*simp add*: *nprt-def le-iff-inf inf-ACI*)

lemma *pprt-eq-0* [*simp*, *noatp*]: $x \leq 0 \Longrightarrow pprt\ x = 0$
by (*simp add*: *pprt-def le-iff-sup sup-ACI*)

lemma *nprt-eq-0* [*simp*, *noatp*]: $0 \leq x \Longrightarrow nprt\ x = 0$
by (*simp add*: *nprt-def le-iff-inf inf-ACI*)

lemma *sup-0-imp-0*: $sup\ a\ (-\ a) = 0 \Longrightarrow a = 0$
proof −
  {
    fix *a*::'*a*
    assume *hyp*: $sup\ a\ (-a) = 0$
    hence $sup\ a\ (-a) + a = a$ **by** (*simp*)
    hence $sup\ (a+a)\ 0 = a$ **by** (*simp add*: *add-sup-distrib-right*)
    hence $sup\ (a+a)\ 0 <= a$ **by** (*simp*)
    hence $0 <= a$ **by** (*blast intro*: *order-trans inf-sup-ord*)
  }
  note $p = this$
  assume *hyp*:$sup\ a\ (-a) = 0$
  hence *hyp2*:$sup\ (-a)\ (-(-a)) = 0$ **by** (*simp add*: *sup-commute*)
  from *p*[*OF hyp*] *p*[*OF hyp2*] **show** $a = 0$ **by** *simp*
qed

lemma *inf-0-imp-0*: $inf\ a\ (-a) = 0 \Longrightarrow a = 0$
apply (*simp add*: *inf-eq-neg-sup*)
apply (*simp add*: *sup-commute*)
apply (*erule sup-0-imp-0*)
done

lemma *inf-0-eq-0* [*simp*, *noatp*]: $inf\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
by (*rule*, *erule inf-0-imp-0*) *simp*

lemma *sup-0-eq-0* [*simp*, *noatp*]: $sup\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
by (*rule*, *erule sup-0-imp-0*) *simp*

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]:
  $0 \leq a + a \longleftrightarrow 0 \leq a$
proof
  assume $0 <= a + a$
  hence *a*:$inf\ (a+a)\ 0 = 0$ **by** (*simp add*: *le-iff-inf inf-commute*)
  have $(inf\ a\ 0)+(inf\ a\ 0) = inf\ (inf\ (a+a)\ 0)\ a$ (**is** *?l*=-)
    by (*simp add*: *add-sup-inf-distribs inf-ACI*)
  hence $?l = 0 + inf\ a\ 0$ **by** (*simp add*: *a*, *simp add*: *inf-commute*)
  hence $inf\ a\ 0 = 0$ **by** (*simp only*: *add-right-cancel*)
  then show $0 <= a$ **by** (*simp add*: *le-iff-inf inf-commute*)
next
  assume *a*: $0 <= a$
  show $0 <= a + a$ **by** (*simp add*: *add-mono*[*OF a a*, *simplified*])
qed

lemma *double-zero*: $a + a = 0 \longleftrightarrow a = 0$
proof
assume *assm*: $a + a = 0$
then have $a + a + - a = - a$ **by** *simp*
then have $a + (a + - a) = - a$ **by** (*simp only*: *add-assoc*)
then have *a*: $- a = a$ **by** *simp*
show $a = 0$ **apply** (*rule antisym*)
apply (*unfold neg-le-iff-le* [*symmetric*, *of a*])
unfolding *a* **apply** *simp*
unfolding *zero-le-double-add-iff-zero-le-single-add* [*symmetric*, *of a*]
unfolding *assm* **unfolding** *le-less* **apply** *simp-all* **done**
next
assume $a = 0$ **then show** $a + a = 0$ **by** *simp*
qed

lemma *zero-less-double-add-iff-zero-less-single-add*:
$0 < a + a \longleftrightarrow 0 < a$
proof (*cases* $a = 0$)
case *True* **then show** *?thesis* **by** *auto*
next
case *False* **then show** *?thesis*
unfolding *less-le* **apply** *simp* **apply** *rule*
apply *clarify*
apply *rule*
apply *assumption*
apply (*rule notI*)
unfolding *double-zero* [*symmetric*, *of a*] **apply** *simp*
done
qed

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]:
$a + a \leq 0 \longleftrightarrow a \leq 0$
proof −
have $a + a \leq 0 \longleftrightarrow 0 \leq - (a + a)$ **by** (*subst le-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a \leq 0$ **by** (*simp add*: *zero-le-double-add-iff-zero-le-single-add*)
ultimately show *?thesis* **by** *blast*
qed

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]:
$a + a < 0 \longleftrightarrow a < 0$
proof −
have $a + a < 0 \longleftrightarrow 0 < - (a + a)$ **by** (*subst less-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a < 0$ **by** (*simp add*: *zero-less-double-add-iff-zero-less-single-add*)
ultimately show *?thesis* **by** *blast*
qed

declare *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq - a \longleftrightarrow a \leq 0$

proof −
from *add-le-cancel-left* [*of uminus a plus a a zero*]
have (*a* <= −*a*) = (*a*+*a* <= *0*)
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *minus-le-self-iff*: − *a* ≤ *a* ⟷ *0* ≤ *a*
proof −
from *add-le-cancel-left* [*of uminus a zero plus a a*]
have (−*a* <= *a*) = (*0* <= *a*+*a*)
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *zero-le-iff-zero-nprt*: *0* ≤ *a* ⟷ *nprt a* = *0*
by (*simp add*: *le-iff-inf nprt-def inf-commute*)

lemma *le-zero-iff-zero-pprt*: *a* ≤ *0* ⟷ *pprt a* = *0*
by (*simp add*: *le-iff-sup pprt-def sup-commute*)

lemma *le-zero-iff-pprt-id*: *0* ≤ *a* ⟷ *pprt a* = *a*
by (*simp add*: *le-iff-sup pprt-def sup-commute*)

lemma *zero-le-iff-nprt-id*: *a* ≤ *0* ⟷ *nprt a* = *a*
by (*simp add*: *le-iff-inf nprt-def inf-commute*)

lemma *pprt-mono* [*simp*, *noatp*]: *a* ≤ *b* ⟹ *pprt a* ≤ *pprt b*
by (*simp add*: *le-iff-sup pprt-def sup-ACI sup-assoc* [*symmetric*, *of a*])

lemma *nprt-mono* [*simp*, *noatp*]: *a* ≤ *b* ⟹ *nprt a* ≤ *nprt b*
by (*simp add*: *le-iff-inf nprt-def inf-ACI inf-assoc* [*symmetric*, *of a*])

end

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

class *lordered-ab-group-add-abs* = *lordered-ab-group-add* + *abs* +
assumes *abs-lattice*: |*a*| = *sup a* (− *a*)
begin

lemma *abs-prts*: |*a*| = *pprt a* − *nprt a*
proof −
have *0* ≤ |*a*|
proof −
have *a*: *a* ≤ |*a*| **and** *b*: − *a* ≤ |*a*| **by** (*auto simp add*: *abs-lattice*)
show *?thesis* **by** (*rule add-mono* [*OF a b*, *simplified*])

```
  qed
  then have 0 ≤ sup a (− a) unfolding abs-lattice .
  then have sup (sup a (− a)) 0 = sup a (− a) by (rule sup-absorb1)
  then show ?thesis
    by (simp add: add-sup-inf-distribs sup-ACI
      pprt-def nprt-def diff-minus abs-lattice)
qed

subclass pordered-ab-group-add-abs
proof
  have abs-ge-zero [simp]: ⋀a. 0 ≤ |a|
  proof −
    fix a b
    have a: a ≤ |a| and b: − a ≤ |a| by (auto simp add: abs-lattice)
    show 0 ≤ |a| by (rule add-mono [OF a b, simplified])
  qed
  have abs-leI: ⋀a b. a ≤ b ⟹ − a ≤ b ⟹ |a| ≤ b
    by (simp add: abs-lattice le-supI)
  fix a b
  show 0 ≤ |a| by simp
  show a ≤ |a|
    by (auto simp add: abs-lattice)
  show |−a| = |a|
    by (simp add: abs-lattice sup-commute)
  show a ≤ b ⟹ − a ≤ b ⟹ |a| ≤ b by (fact abs-leI)
  show |a + b| ≤ |a| + |b|
  proof −
    have g:abs a + abs b = sup (a+b) (sup (−a−b) (sup (−a+b) (a + (−b))))
(is -=sup ?m ?n)
      by (simp add: abs-lattice add-sup-inf-distribs sup-ACI diff-minus)
    have a:a+b <= sup ?m ?n by (simp)
    have b:−a−b <= ?n by (simp)
    have c:?n <= sup ?m ?n by (simp)
    from b c have d: −a−b <= sup ?m ?n by(rule order-trans)
    have e:−a−b = −(a+b) by (simp add: diff-minus)
    from a d e have abs(a+b) <= sup ?m ?n
      by (drule-tac abs-leI, auto)
    with g[symmetric] show ?thesis by simp
  qed
qed

end

lemma sup-eq-if:
  fixes a :: 'a::{lordered-ab-group-add, linorder}
  shows sup a (− a) = (if a < 0 then − a else a)
proof −
  note add-le-cancel-right [of a a − a, symmetric, simplified]
  moreover note add-le-cancel-right [of −a a a, symmetric, simplified]
```

then show *?thesis* **by** (*auto simp*: *sup-max max-def*)
qed

lemma *abs-if-lattice*:
fixes *a* :: *'a*::{*lordered-ab-group-add-abs*, *linorder*}
shows |*a*| = (*if a < 0 then − a else a*)
by *auto*

Needed for abelian cancellation simprocs:

lemma *add-cancel-21*: ((*x*::*'a*::*ab-group-add*) + (*y* + *z*) = *y* + *u*) = (*x* + *z* = *u*)
apply (*subst add-left-commute*)
apply (*subst add-left-cancel*)
apply *simp*
done

lemma *add-cancel-end*: (*x* + (*y* + *z*) = *y*) = (*x* = − (*z*::*'a*::*ab-group-add*))
apply (*subst add-cancel-21*[*of - - - 0*, *simplified*])
apply (*simp add*: *add-right-cancel*[*symmetric*, *of x −z z*, *simplified*])
done

lemma *less-eqI*: (*x*::*'a*::*pordered-ab-group-add*) − *y* = *x'* − *y'* ⟹ (*x* < *y*) = (*x'* < *y'*)
by (*simp add*: *less-iff-diff-less-0*[*of x y*] *less-iff-diff-less-0*[*of x' y'*])

lemma *le-eqI*: (*x*::*'a*::*pordered-ab-group-add*) − *y* = *x'* − *y'* ⟹ (*y* <= *x*) = (*y'* <= *x'*)
apply (*simp add*: *le-iff-diff-le-0*[*of y x*] *le-iff-diff-le-0*[*of y' x'*])
apply (*simp add*: *neg-le-iff-le*[*symmetric*, *of y−x 0*] *neg-le-iff-le*[*symmetric*, *of y'−x' 0*])
done

lemma *eq-eqI*: (*x*::*'a*::*ab-group-add*) − *y* = *x'* − *y'* ⟹ (*x* = *y*) = (*x'* = *y'*)
by (*simp only*: *eq-iff-diff-eq-0*[*of x y*] *eq-iff-diff-eq-0*[*of x' y'*])

lemma *diff-def*: (*x*::*'a*::*ab-group-add*) − *y* == *x* + (−*y*)
by (*simp add*: *diff-minus*)

lemma *add-minus-cancel*: (*a*::*'a*::*ab-group-add*) + (−*a* + *b*) = *b*
by (*simp add*: *add-assoc*[*symmetric*])

lemma *le-add-right-mono*:
assumes
a <= *b* + (*c*::*'a*::*pordered-ab-group-add*)
c <= *d*
shows *a* <= *b* + *d*
apply (*rule-tac order-trans*[**where** *y* = *b*+*c*])
apply (*simp-all add*: *prems*)
done

lemma *estimate-by-abs*:
  $a + b <= (c::'a::lordered\text{-}ab\text{-}group\text{-}add\text{-}abs) \Longrightarrow a <= c + abs\ b$
proof −
  assume $a+b <= c$
  hence *2*: $a <= c+(-b)$ **by** (*simp add*: *algebra-simps*)
  have *3*: $(-b) <= abs\ b$ **by** (*rule abs-ge-minus-self*)
  show *?thesis* **by** (*rule le-add-right-mono*[*OF 2 3*])
qed

10.7 Tools setup

lemma *add-mono-thms-ordered-semiring* [*noatp*]:
  fixes $i\ j\ k :: 'a::pordered\text{-}ab\text{-}semigroup\text{-}add$
  shows $i \le j \wedge k \le l \Longrightarrow i + k \le j + l$
    and $i = j \wedge k \le l \Longrightarrow i + k \le j + l$
    and $i \le j \wedge k = l \Longrightarrow i + k \le j + l$
    and $i = j \wedge k = l \Longrightarrow i + k = j + l$
by (*rule add-mono*, *clarify*+)+

lemma *add-mono-thms-ordered-field* [*noatp*]:
  fixes $i\ j\ k :: 'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add$
  shows $i < j \wedge k = l \Longrightarrow i + k < j + l$
    and $i = j \wedge k < l \Longrightarrow i + k < j + l$
    and $i < j \wedge k \le l \Longrightarrow i + k < j + l$
    and $i \le j \wedge k < l \Longrightarrow i + k < j + l$
    and $i < j \wedge k < l \Longrightarrow i + k < j + l$
by (*auto intro*: *add-strict-right-mono add-strict-left-mono*
  add-less-le-mono add-le-less-mono add-strict-mono)

Simplification of $x - y < (0::'a)$, etc.

lemmas *diff-less-0-iff-less* [*simp*, *noatp*] = *less-iff-diff-less-0* [*symmetric*]
lemmas *diff-le-0-iff-le* [*simp*, *noatp*] = *le-iff-diff-le-0* [*symmetric*]

ML ⟪
structure ab-group-add-cancel = Abel-Cancel
(

(∗ *term order for abelian groups* ∗)

fun agrp-ord (*Const* (*a*, -)) = *find-index* (*fn a' => a = a'*)
      [@{*const-name HOL.zero*}, @{*const-name HOL.plus*},
        @{*const-name HOL.uminus*}, @{*const-name HOL.minus*}]
  | *agrp-ord* - = ~*1*;

fun termless-agrp (*a*, *b*) = (*TermOrd.term-lpo agrp-ord* (*a*, *b*) = *LESS*);

local
  val ac1 = mk-meta-eq @{*thm add-assoc*};
  val ac2 = mk-meta-eq @{*thm add-commute*};

```
  val ac3 = mk-meta-eq @{thm add-left-commute};
  fun solve-add-ac thy - (- $ (Const (@{const-name HOL.plus},-) $ - $ -) $ -) =
        SOME ac1
    | solve-add-ac thy - (- $ x $ (Const (@{const-name HOL.plus},-) $ y $ z)) =
        if termless-agrp (y, x) then SOME ac3 else NONE
    | solve-add-ac thy - (- $ x $ y) =
        if termless-agrp (y, x) then SOME ac2 else NONE
    | solve-add-ac thy - - = NONE
in
  val add-ac-proc = Simplifier.simproc (the-context ())
    add-ac-proc [x + y::'a::ab-semigroup-add] solve-add-ac;
end;

val eq-reflection = @{thm eq-reflection};

val T = @{typ 'a::ab-group-add};

val cancel-ss = HOL-basic-ss settermless termless-agrp
  addsimprocs [add-ac-proc] addsimps
  [@{thm add-0-left}, @{thm add-0-right}, @{thm diff-def},
   @{thm minus-add-distrib}, @{thm minus-minus}, @{thm minus-zero},
   @{thm right-minus}, @{thm left-minus}, @{thm add-minus-cancel},
   @{thm minus-add-cancel}];

val sum-pats = [@{cterm x + y::'a::ab-group-add}, @{cterm x - y::'a::ab-group-add}];

val eqI-rules = [@{thm less-eqI}, @{thm le-eqI}, @{thm eq-eqI}];

val dest-eqI =
  fst o HOLogic.dest-bin op = HOLogic.boolT o HOLogic.dest-Trueprop o concl-of;

);
⟫

ML ⟪
  Addsimprocs [ab-group-add-cancel.sum-conv, ab-group-add-cancel.rel-conv];
⟫

end
```

11 Ring-and-Field: (Ordered) Rings and Fields

theory *Ring-and-Field*
imports *OrderedGroup*
begin

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +
  assumes *left-distrib*[*algebra-simps*]: $(a + b) * c = a * c + b * c$
  assumes *right-distrib*[*algebra-simps*]: $a * (b + c) = a * b + a * c$
begin

For the *combine-numerals* simproc

lemma *combine-common-factor*:
  $a * e + (b * e + c) = (a + b) * e + c$
by (*simp add*: *left-distrib add-ac*)

end

class *mult-zero* = *times* + *zero* +
  assumes *mult-zero-left* [*simp*]: $0 * a = 0$
  assumes *mult-zero-right* [*simp*]: $a * 0 = 0$

class *semiring-0* = *semiring* + *comm-monoid-add* + *mult-zero*

class *semiring-0-cancel* = *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0*
proof
  fix $a :: 'a$
  have $0 * a + 0 * a = 0 * a + 0$ **by** (*simp add*: *left-distrib* [*symmetric*])
  thus $0 * a = 0$ **by** (*simp only*: *add-left-cancel*)
next
  fix $a :: 'a$
  have $a * 0 + a * 0 = a * 0 + 0$ **by** (*simp add*: *right-distrib* [*symmetric*])
  thus $a * 0 = 0$ **by** (*simp only*: *add-left-cancel*)
qed

end

class *comm-semiring* = *ab-semigroup-add* + *ab-semigroup-mult* +
  assumes *distrib*: $(a + b) * c = a * c + b * c$
begin

subclass *semiring*
proof
  fix *a b c* :: *'a*
  show $(a + b) * c = a * c + b * c$ **by** (*simp add*: *distrib*)
  have $a * (b + c) = (b + c) * a$ **by** (*simp add*: *mult-ac*)
  also have $... = b * a + c * a$ **by** (*simp only*: *distrib*)
  also have $... = a * b + a * c$ **by** (*simp add*: *mult-ac*)
  finally show $a * (b + c) = a * b + a * c$ **by** *blast*
qed

end

class *comm-semiring-0* = *comm-semiring* + *comm-monoid-add* + *mult-zero*
begin

subclass *semiring-0* **..**

end

class *comm-semiring-0-cancel* = *comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* **..**

subclass *comm-semiring-0* **..**

end

class *zero-neq-one* = *zero* + *one* +
  assumes *zero-neq-one* [*simp*]: $0 \neq 1$
begin

lemma *one-neq-zero* [*simp*]: $1 \neq 0$
by (*rule not-sym*) (*rule zero-neq-one*)

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*

Abstract divisibility

class *dvd* = *times*
begin

definition *dvd* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** *dvd 50*) **where**
  [*code del*]: *b dvd a* ⟷ $(\exists k.\ a = b * k)$

lemma *dvdI* [*intro?*]: $a = b * k \Longrightarrow b\ dvd\ a$
  unfolding *dvd-def* **..**

lemma *dvdE* [*elim?*]: *b dvd a* $\Longrightarrow$ ($\bigwedge k.$ *a* = *b* * *k* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
unfolding *dvd-def* **by** *blast*

end

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult* + *dvd*

begin

subclass *semiring-1* **..**

lemma *dvd-refl*[*simp*]: *a dvd a*
proof
show *a* = *a* * *1* **by** *simp*
qed

lemma *dvd-trans*:
assumes *a dvd b* **and** *b dvd c*
shows *a dvd c*
proof −
from *assms* **obtain** *v* **where** *b* = *a* * *v* **by** (*auto elim*!: *dvdE*)
moreover from *assms* **obtain** *w* **where** *c* = *b* * *w* **by** (*auto elim*!: *dvdE*)
ultimately have *c* = *a* * (*v* * *w*) **by** (*simp add*: *mult-assoc*)
then show *?thesis* **..**
qed

lemma *dvd-0-left-iff* [*noatp*, *simp*]: *0 dvd a* $\longleftrightarrow$ *a* = *0*
by (*auto intro*: *dvd-refl elim*!: *dvdE*)

lemma *dvd-0-right* [*iff*]: *a dvd 0*
proof
show *0* = *a* * *0* **by** *simp*
qed

lemma *one-dvd* [*simp*]: *1 dvd a*
by (*auto intro*!: *dvdI*)

lemma *dvd-mult*[*simp*]: *a dvd c* $\Longrightarrow$ *a dvd* (*b* * *c*)
by (*auto intro*!: *mult-left-commute dvdI elim*!: *dvdE*)

lemma *dvd-mult2*[*simp*]: *a dvd b* $\Longrightarrow$ *a dvd* (*b* * *c*)
apply (*subst mult-commute*)
apply (*erule dvd-mult*)
done

lemma *dvd-triv-right* [*simp*]: *a dvd b* * *a*
by (*rule dvd-mult*) (*rule dvd-refl*)

lemma *dvd-triv-left* [*simp*]: *a dvd a * b*
by (*rule dvd-mult2*) (*rule dvd-refl*)

lemma *mult-dvd-mono*:
  assumes *a dvd b*
    and *c dvd d*
  shows *a * c dvd b * d*
proof −
  from ‹*a dvd b*› **obtain** *b'* **where** *b = a * b'* **..**
  moreover from ‹*c dvd d*› **obtain** *d'* **where** *d = c * d'* **..**
  ultimately have *b * d = (a * c) * (b' * d')* **by** (*simp add*: *mult-ac*)
  then show *?thesis* **..**
qed

lemma *dvd-mult-left*: *a * b dvd c ⟹ a dvd c*
by (*simp add*: *dvd-def mult-assoc*, *blast*)

lemma *dvd-mult-right*: *a * b dvd c ⟹ b dvd c*
  unfolding *mult-ac* [*of a*] **by** (*rule dvd-mult-left*)

lemma *dvd-0-left*: *0 dvd a ⟹ a = 0*
by *simp*

lemma *dvd-add*[*simp*]:
  assumes *a dvd b* **and** *a dvd c* **shows** *a dvd (b + c)*
proof −
  from ‹*a dvd b*› **obtain** *b'* **where** *b = a * b'* **..**
  moreover from ‹*a dvd c*› **obtain** *c'* **where** *c = a * c'* **..**
  ultimately have *b + c = a * (b' + c')* **by** (*simp add*: *right-distrib*)
  then show *?thesis* **..**
qed

end

class *no-zero-divisors = zero + times +*
  assumes *no-zero-divisors*: *a ≠ 0 ⟹ b ≠ 0 ⟹ a * b ≠ 0*

class *semiring-1-cancel = semiring + cancel-comm-monoid-add*
  + zero-neq-one + monoid-mult
begin

subclass *semiring-0-cancel* **..**

subclass *semiring-1* **..**

end

class *comm-semiring-1-cancel = comm-semiring + cancel-comm-monoid-add*

+ zero-neq-one + comm-monoid-mult
begin

subclass *semiring-1-cancel* **..**
subclass *comm-semiring-0-cancel* **..**
subclass *comm-semiring-1* **..**

end

class *ring* = *semiring + ab-group-add*
begin

subclass *semiring-0-cancel* **..**

Distribution rules

lemma *minus-mult-left*: $-(a * b) = -a * b$
by (*rule equals-zero-I*) (*simp add*: *left-distrib* [*symmetric*])

lemma *minus-mult-right*: $-(a * b) = a * -b$
by (*rule equals-zero-I*) (*simp add*: *right-distrib* [*symmetric*])

Extract signs from products

lemmas *mult-minus-left* [*simp*, *noatp*] = *minus-mult-left* [*symmetric*]
lemmas *mult-minus-right* [*simp*,*noatp*] = *minus-mult-right* [*symmetric*]

lemma *minus-mult-minus* [*simp*]: $-a * -b = a * b$
by *simp*

lemma *minus-mult-commute*: $-a * b = a * -b$
by *simp*

lemma *right-diff-distrib*[*algebra-simps*]: $a * (b - c) = a * b - a * c$
by (*simp add*: *right-distrib diff-minus*)

lemma *left-diff-distrib*[*algebra-simps*]: $(a - b) * c = a * c - b * c$
by (*simp add*: *left-distrib diff-minus*)

lemmas *ring-distribs*[*noatp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

Legacy - use *algebra-simps*

lemmas *ring-simps*[*noatp*] = *algebra-simps*

lemma *eq-add-iff1*:
$a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
by (*simp add*: *algebra-simps*)

lemma *eq-add-iff2*:
$a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$

by (*simp add*: *algebra-simps*)

end

lemmas *ring-distribs*[*noatp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

class *comm-ring* = *comm-semiring* + *ab-group-add*
begin

subclass *ring* ..
subclass *comm-semiring-0-cancel* ..

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* ..

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*

begin

subclass *ring-1* ..
subclass *comm-semiring-1-cancel* ..

lemma *dvd-minus-iff* [*simp*]: *x dvd* − *y* ⟷ *x dvd y*
proof
assume *x dvd* − *y*
then have *x dvd* − *1* ∗ − *y* **by** (*rule dvd-mult*)
then show *x dvd y* **by** *simp*
next
assume *x dvd y*
then have *x dvd* − *1* ∗ *y* **by** (*rule dvd-mult*)
then show *x dvd* − *y* **by** *simp*
qed

lemma *minus-dvd-iff* [*simp*]: − *x dvd y* ⟷ *x dvd y*
proof
assume − *x dvd y*
then obtain *k* **where** *y* = − *x* ∗ *k* ..
then have *y* = *x* ∗ − *k* **by** *simp*
then show *x dvd y* ..
next
assume *x dvd y*
then obtain *k* **where** *y* = *x* ∗ *k* ..

then have $y = - x * - k$ **by** *simp*
then show $- x$ *dvd* y **..**
qed

lemma *dvd-diff*[*simp*]: x *dvd* $y \Longrightarrow x$ *dvd* $z \Longrightarrow x$ *dvd* $(y - z)$
by (*simp add*: *diff-minus dvd-minus-iff*)

end

class *ring-no-zero-divisors* = *ring* + *no-zero-divisors*
begin

lemma *mult-eq-0-iff* [*simp*]:
shows $a * b = 0 \longleftrightarrow (a = 0 \vee b = 0)$
proof (*cases* $a = 0 \vee b = 0$)
case *False* **then have** $a \neq 0$ **and** $b \neq 0$ **by** *auto*
then show *?thesis* **using** *no-zero-divisors* **by** *simp*
next
case *True* **then show** *?thesis* **by** *auto*
qed

Cancellation of equalities with a common factor

lemma *mult-cancel-right* [*simp*, *noatp*]:
$a * c = b * c \longleftrightarrow c = 0 \vee a = b$
proof −
have $(a * c = b * c) = ((a - b) * c = 0)$
by (*simp add*: *algebra-simps right-minus-eq*)
thus *?thesis* **by** (*simp add*: *disj-commute right-minus-eq*)
qed

lemma *mult-cancel-left* [*simp*, *noatp*]:
$c * a = c * b \longleftrightarrow c = 0 \vee a = b$
proof −
have $(c * a = c * b) = (c * (a - b) = 0)$
by (*simp add*: *algebra-simps right-minus-eq*)
thus *?thesis* **by** (*simp add*: *right-minus-eq*)
qed

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

lemma *mult-cancel-right1* [*simp*]:
$c = b * c \longleftrightarrow c = 0 \vee b = 1$
by (*insert mult-cancel-right* [*of 1 c b*], *force*)

lemma *mult-cancel-right2* [*simp*]:
$a * c = c \longleftrightarrow c = 0 \vee a = 1$

by (*insert mult-cancel-right* [*of a c 1*], *simp*)

lemma *mult-cancel-left1* [*simp*]:
  $c = c * b \longleftrightarrow c = 0 \vee b = 1$
by (*insert mult-cancel-left* [*of c 1 b*], *force*)

lemma *mult-cancel-left2* [*simp*]:
  $c * a = c \longleftrightarrow c = 0 \vee a = 1$
by (*insert mult-cancel-left* [*of c a 1*], *simp*)

end

class *idom* = *comm-ring-1* + *no-zero-divisors*
begin

subclass *ring-1-no-zero-divisors* **..**

lemma *square-eq-iff*: $a * a = b * b \longleftrightarrow (a = b \vee a = - b)$
proof
  assume $a * a = b * b$
  then have $(a - b) * (a + b) = 0$
    by (*simp add*: *algebra-simps*)
  then show $a = b \vee a = - b$
    by (*simp add*: *right-minus-eq eq-neg-iff-add-eq-0*)
next
  assume $a = b \vee a = - b$
  then show $a * a = b * b$ **by** *auto*
qed

lemma *dvd-mult-cancel-right* [*simp*]:
  $a * c \ dvd \ b * c \longleftrightarrow c = 0 \vee a \ dvd \ b$
proof −
  have $a * c \ dvd \ b * c \longleftrightarrow (\exists k.\ b * c = (a * k) * c)$
    unfolding *dvd-def* **by** (*simp add*: *mult-ac*)
  also have $(\exists k.\ b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \ dvd \ b$
    unfolding *dvd-def* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *dvd-mult-cancel-left* [*simp*]:
  $c * a \ dvd \ c * b \longleftrightarrow c = 0 \vee a \ dvd \ b$
proof −
  have $c * a \ dvd \ c * b \longleftrightarrow (\exists k.\ b * c = (a * k) * c)$
    unfolding *dvd-def* **by** (*simp add*: *mult-ac*)
  also have $(\exists k.\ b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \ dvd \ b$
    unfolding *dvd-def* **by** *simp*
  finally show *?thesis* **.**
qed

end

class *division-ring* = *ring-1* + *inverse* +
  assumes *left-inverse* [*simp*]: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
  assumes *right-inverse* [*simp*]: $a \neq 0 \Longrightarrow a * inverse\ a = 1$
begin

subclass *ring-1-no-zero-divisors*
proof
  fix *a b* :: $'a$
  assume *a*: $a \neq 0$ **and** *b*: $b \neq 0$
  show $a * b \neq 0$
  proof
    assume *ab*: $a * b = 0$
    hence $0 = inverse\ a * (a * b) * inverse\ b$ **by** *simp*
    also have $\ldots = (inverse\ a * a) * (b * inverse\ b)$
      by (*simp only*: *mult-assoc*)
    also have $\ldots = 1$ **using** *a b* **by** *simp*
    finally show *False* **by** *simp*
  qed
qed

lemma *nonzero-imp-inverse-nonzero*:
  $a \neq 0 \Longrightarrow inverse\ a \neq 0$
proof
  assume *ianz*: $inverse\ a = 0$
  assume $a \neq 0$
  hence $1 = a * inverse\ a$ **by** *simp*
  also have ... $= 0$ **by** (*simp add*: *ianz*)
  finally have $1 = 0$ **.**
  thus *False* **by** (*simp add*: *eq-commute*)
qed

lemma *inverse-zero-imp-zero*:
  $inverse\ a = 0 \Longrightarrow a = 0$
apply (*rule classical*)
apply (*drule nonzero-imp-inverse-nonzero*)
apply *auto*
done

lemma *inverse-unique*:
  assumes *ab*: $a * b = 1$
  shows $inverse\ a = b$
proof −
  have $a \neq 0$ **using** *ab* **by** (*cases* $a = 0$) *simp-all*
  moreover have $inverse\ a * (a * b) = inverse\ a$ **by** (*simp add*: *ab*)
  ultimately show *?thesis* **by** (*simp add*: *mult-assoc* [*symmetric*])
qed

lemma *nonzero-inverse-minus-eq*:
  $a \neq 0 \Longrightarrow inverse\ (-\ a) = -\ inverse\ a$
by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-inverse-eq*:
  $a \neq 0 \Longrightarrow inverse\ (inverse\ a) = a$
by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-eq-imp-eq*:
  assumes $inverse\ a = inverse\ b$ **and** $a \neq 0$ **and** $b \neq 0$
  shows $a = b$
proof −
  from ‹$inverse\ a = inverse\ b$›
  have $inverse\ (inverse\ a) = inverse\ (inverse\ b)$ **by** (*rule arg-cong*)
  with ‹$a \neq 0$› **and** ‹$b \neq 0$› **show** $a = b$
    by (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-1* [*simp*]: $inverse\ 1 = 1$
by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-mult-distrib*:
  assumes $a \neq 0$ **and** $b \neq 0$
  shows $inverse\ (a * b) = inverse\ b * inverse\ a$
proof −
  have $a * (b * inverse\ b) * inverse\ a = 1$ **using** *assms* **by** *simp*
  hence $a * b * (inverse\ b * inverse\ a) = 1$ **by** (*simp only*: *mult-assoc*)
  thus *?thesis* **by** (*rule inverse-unique*)
qed

lemma *division-ring-inverse-add*:
  $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ a + inverse\ b = inverse\ a * (a + b) * inverse\ b$
by (*simp add*: *algebra-simps*)

lemma *division-ring-inverse-diff*:
  $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ a - inverse\ b = inverse\ a * (b - a) * inverse\ b$
by (*simp add*: *algebra-simps*)

end

class *field* = *comm-ring-1* + *inverse* +
  assumes *field-inverse*: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
  assumes *divide-inverse*: $a\ /\ b = a * inverse\ b$
begin

subclass *division-ring*
proof
  fix $a :: 'a$
  assume $a \neq 0$

thus *inverse a ∗ a = 1* **by** (*rule field-inverse*)
thus *a ∗ inverse a = 1* **by** (*simp only*: *mult-commute*)
qed

subclass *idom* **..**

lemma *right-inverse-eq*: *b ≠ 0 ⟹ a / b = 1 ⟷ a = b*
proof
assume *neq*: *b ≠ 0*
{
hence *a* = (*a / b*) ∗ *b* **by** (*simp add*: *divide-inverse mult-ac*)
also assume *a / b = 1*
finally show *a = b* **by** *simp*
next
assume *a = b*
with *neq* **show** *a / b = 1* **by** (*simp add*: *divide-inverse*)
}
qed

lemma *nonzero-inverse-eq-divide*: *a ≠ 0 ⟹ inverse a = 1 / a*
by (*simp add*: *divide-inverse*)

lemma *divide-self* [*simp*]: *a ≠ 0 ⟹ a / a = 1*
by (*simp add*: *divide-inverse*)

lemma *divide-zero-left* [*simp*]: *0 / a = 0*
by (*simp add*: *divide-inverse*)

lemma *inverse-eq-divide*: *inverse a = 1 / a*
by (*simp add*: *divide-inverse*)

lemma *add-divide-distrib*: (*a*+*b*) / *c* = *a*/*c* + *b*/*c*
by (*simp add*: *divide-inverse algebra-simps*)

There is no slick version using division by zero.

lemma *inverse-add*:
[| *a ≠ 0*; *b ≠ 0* |]
==> *inverse a* + *inverse b* = (*a* + *b*) ∗ *inverse a* ∗ *inverse b*
by (*simp add*: *division-ring-inverse-add mult-ac*)

lemma *nonzero-mult-divide-mult-cancel-left* [*simp*, *noatp*]:
assumes [*simp*]: *b≠0* **and** [*simp*]: *c≠0* **shows** (*c∗a*)/(*c∗b*) = *a*/*b*
proof −
have (*c∗a*)/(*c∗b*) = *c* ∗ *a* ∗ (*inverse b* ∗ *inverse c*)
by (*simp add*: *divide-inverse nonzero-inverse-mult-distrib*)
also have ... = *a* ∗ *inverse b* ∗ (*inverse c* ∗ *c*)
by (*simp only*: *mult-ac*)
also have ... = *a* ∗ *inverse b* **by** *simp*
finally show *?thesis* **by** (*simp add*: *divide-inverse*)

qed

lemma *nonzero-mult-divide-mult-cancel-right* [*simp*, *noatp*]:
$[\![b \neq 0;\ c \neq 0]\!] \Longrightarrow (a * c) \ / \ (b * c) = a \ / \ b$
by (*simp add*: *mult-commute* [*of* - *c*])

lemma *divide-1* [*simp*]: $a \ / \ 1 = a$
by (*simp add*: *divide-inverse*)

lemma *times-divide-eq-right*: $a * (b \ / \ c) = (a * b) \ / \ c$
by (*simp add*: *divide-inverse mult-assoc*)

lemma *times-divide-eq-left*: $(b \ / \ c) * a = (b * a) \ / \ c$
by (*simp add*: *divide-inverse mult-ac*)

These are later declared as simp rules.

lemmas *times-divide-eq* [*noatp*] = *times-divide-eq-right times-divide-eq-left*

lemma *add-frac-eq*:
assumes $y \neq 0$ **and** $z \neq 0$
shows $x \ / \ y + w \ / \ z = (x * z + w * y) \ / \ (y * z)$
proof −
have $x \ / \ y + w \ / \ z = (x * z) \ / \ (y * z) + (y * w) \ / \ (y * z)$
using *assms* **by** *simp*
also have $\ldots = (x * z + y * w) \ / \ (y * z)$
by (*simp only*: *add-divide-distrib*)
finally show *?thesis*
by (*simp only*: *mult-commute*)
qed

Special Cancellation Simprules for Division

lemma *nonzero-mult-divide-cancel-right* [*simp*, *noatp*]:
$b \neq 0 \Longrightarrow a * b \ / \ b = a$
using *nonzero-mult-divide-mult-cancel-right* [*of 1 b a*] **by** *simp*

lemma *nonzero-mult-divide-cancel-left* [*simp*, *noatp*]:
$a \neq 0 \Longrightarrow a * b \ / \ a = b$
using *nonzero-mult-divide-mult-cancel-left* [*of 1 a b*] **by** *simp*

lemma *nonzero-divide-mult-cancel-right* [*simp*, *noatp*]:
$[\![a \neq 0;\ b \neq 0]\!] \Longrightarrow b \ / \ (a * b) = 1 \ / \ a$
using *nonzero-mult-divide-mult-cancel-right* [*of a b 1*] **by** *simp*

lemma *nonzero-divide-mult-cancel-left* [*simp*, *noatp*]:
$[\![a \neq 0;\ b \neq 0]\!] \Longrightarrow a \ / \ (a * b) = 1 \ / \ b$
using *nonzero-mult-divide-mult-cancel-left* [*of b a 1*] **by** *simp*

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*, *noatp*]:
$[\![b \neq 0;\ c \neq 0]\!] \Longrightarrow (c * a) \ / \ (b * c) = a \ / \ b$

using *nonzero-mult-divide-mult-cancel-left* [*of b c a*] **by** (*simp add*: *mult-ac*)

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*, *noatp*]:
$[\![b \neq 0; c \neq 0]\!] \Longrightarrow (a * c) / (c * b) = a / b$
using *nonzero-mult-divide-mult-cancel-right* [*of b c a*] **by** (*simp add*: *mult-ac*)

lemma *minus-divide-left*: $-(a / b) = (-a) / b$
by (*simp add*: *divide-inverse*)

lemma *nonzero-minus-divide-right*: $b \neq 0 ==> -(a / b) = a / (-b)$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *nonzero-minus-divide-divide*: $b \neq 0 ==> (-a) / (-b) = a / b$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *divide-minus-left* [*simp*, *noatp*]: $(-a) / b = -(a / b)$
by (*simp add*: *divide-inverse*)

lemma *diff-divide-distrib*: $(a - b) / c = a / c - b / c$
by (*simp add*: *diff-minus add-divide-distrib*)

lemma *add-divide-eq-iff*:
$z \neq 0 \Longrightarrow x + y / z = (z * x + y) / z$
by (*simp add*: *add-divide-distrib*)

lemma *divide-add-eq-iff*:
$z \neq 0 \Longrightarrow x / z + y = (x + z * y) / z$
by (*simp add*: *add-divide-distrib*)

lemma *diff-divide-eq-iff*:
$z \neq 0 \Longrightarrow x - y / z = (z * x - y) / z$
by (*simp add*: *diff-divide-distrib*)

lemma *divide-diff-eq-iff*:
$z \neq 0 \Longrightarrow x / z - y = (x - z * y) / z$
by (*simp add*: *diff-divide-distrib*)

lemma *nonzero-eq-divide-eq*: $c \neq 0 \Longrightarrow a = b / c \longleftrightarrow a * c = b$
proof −
assume [*simp*]: $c \neq 0$
have $a = b / c \longleftrightarrow a * c = (b / c) * c$ **by** *simp*
also have ... $\longleftrightarrow a * c = b$ **by** (*simp add*: *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *nonzero-divide-eq-eq*: $c \neq 0 \Longrightarrow b / c = a \longleftrightarrow b = a * c$
proof −
assume [*simp*]: $c \neq 0$
have $b / c = a \longleftrightarrow (b / c) * c = a * c$ **by** *simp*

also have ... $\longleftrightarrow$ $b = a * c$ **by** (*simp add*: *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *divide-eq-imp*: $c \neq 0 \Longrightarrow b = a * c \Longrightarrow b \mathbin{/} c = a$
by *simp*

lemma *eq-divide-imp*: $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b \mathbin{/} c$
by (*erule subst*, *simp*)

lemmas *field-eq-simps*[*noatp*] = *algebra-simps*

add-divide-eq-iff divide-add-eq-iff
diff-divide-eq-iff divide-diff-eq-iff

nonzero-eq-divide-eq nonzero-divide-eq-eq

An example:

lemma $[\![a{\neq}b;\ c{\neq}d;\ e{\neq}f]\!] \Longrightarrow ((a-b)*(c-d)*(e-f))/((c-d)*(e-f)*(a-b)) = 1$
apply(*subgoal-tac* $(c-d)*(e-f)*(a-b) \neq 0$)
apply(*simp add*:*field-eq-simps*)
apply(*simp*)
done

lemma *diff-frac-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y - w \mathbin{/} z = (x * z - w * y) \mathbin{/} (y * z)$
by (*simp add*: *field-eq-simps times-divide-eq*)

lemma *frac-eq-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow (x \mathbin{/} y = w \mathbin{/} z) = (x * z = w * y)$
by (*simp add*: *field-eq-simps times-divide-eq*)

end

class *division-by-zero* = *zero* + *inverse* +
assumes *inverse-zero* [*simp*]: *inverse 0* = *0*

lemma *divide-zero* [*simp*]:
$a \mathbin{/} 0 = (0::'a::\{field, division\text{-}by\text{-}zero\})$
by (*simp add*: *divide-inverse*)

lemma *divide-self-if* [*simp*]:
$a \mathbin{/} (a::'a::\{field, division\text{-}by\text{-}zero\}) = (if\ a{=}0\ then\ 0\ else\ 1)$
by *simp*

class *mult-mono* = *times* + *zero* + *ord* +
assumes *mult-left-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
assumes *mult-right-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * c$

class *pordered-semiring* = *mult-mono* + *semiring-0* + *pordered-ab-semigroup-add*

begin

lemma *mult-mono*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq b \Longrightarrow 0 \leq c$
$\Longrightarrow a * c \leq b * d$
apply (*erule mult-right-mono* [*THEN order-trans*], *assumption*)
apply (*erule mult-left-mono*, *assumption*)
done

lemma *mult-mono'*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq a \Longrightarrow 0 \leq c$
$\Longrightarrow a * c \leq b * d$
apply (*rule mult-mono*)
apply (*fast intro*: *order-trans*)+
done

end

class *pordered-cancel-semiring* = *mult-mono* + *pordered-ab-semigroup-add*
+ *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* **..**
subclass *pordered-semiring* **..**

lemma *mult-nonneg-nonneg*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * b$
using *mult-left-mono* [*of zero b a*] **by** *simp*

lemma *mult-nonneg-nonpos*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow a * b \leq 0$
using *mult-left-mono* [*of b zero a*] **by** *simp*

lemma *mult-nonpos-nonneg*: $a \leq 0 \Longrightarrow 0 \leq b \Longrightarrow a * b \leq 0$
using *mult-right-mono* [*of a zero b*] **by** *simp*

Legacy - use *mult-nonpos-nonneg*

lemma *mult-nonneg-nonpos2*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow b * a \leq 0$
by (*drule mult-right-mono* [*of b zero*], *auto*)

lemma *split-mult-neg-le*: $(0 \leq a \;\&\; b \leq 0) \mid (a \leq 0 \;\&\; 0 \leq b) \Longrightarrow a * b \leq 0$
by (*auto simp add*: *mult-nonneg-nonpos mult-nonneg-nonpos2*)

end

class *ordered-semiring* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add*
+ *mult-mono*
begin

subclass *pordered-cancel-semiring* **..**

subclass *pordered-comm-monoid-add* **..**

lemma *mult-left-less-imp-less*:
  $c * a < c * b \Longrightarrow 0 \leq c \Longrightarrow a < b$
by (*force simp add*: *mult-left-mono not-le* [*symmetric*])

lemma *mult-right-less-imp-less*:
  $a * c < b * c \Longrightarrow 0 \leq c \Longrightarrow a < b$
by (*force simp add*: *mult-right-mono not-le* [*symmetric*])

end

class *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add* +
  assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
  assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* **..**

subclass *ordered-semiring*
proof
  fix *a b c* :: $'a$
  assume *A*: $a \leq b$ $0 \leq c$
  from *A* **show** $c * a \leq c * b$
    unfolding *le-less*
    using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
  from *A* **show** $a * c \leq b * c$
    unfolding *le-less*
    using *mult-strict-right-mono* **by** (*cases* $c = 0$) *auto*
qed

lemma *mult-left-le-imp-le*:
  $c * a \leq c * b \Longrightarrow 0 < c \Longrightarrow a \leq b$
by (*force simp add*: *mult-strict-left-mono -not-less* [*symmetric*])

lemma *mult-right-le-imp-le*:
  $a * c \leq b * c \Longrightarrow 0 < c \Longrightarrow a \leq b$
by (*force simp add*: *mult-strict-right-mono not-less* [*symmetric*])

lemma *mult-pos-pos*: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
using *mult-strict-left-mono* [*of zero b a*] **by** *simp*

lemma *mult-pos-neg*: $0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$
using *mult-strict-left-mono* [*of b zero a*] **by** *simp*

lemma *mult-neg-pos*: $a < 0 \Longrightarrow 0 < b \Longrightarrow a * b < 0$

using *mult-strict-right-mono* [*of a zero b*] **by** *simp*

Legacy - use *mult-neg-pos*

lemma *mult-pos-neg2*: $0 < a \Longrightarrow b < 0 \Longrightarrow b * a < 0$
by (*drule mult-strict-right-mono* [*of b zero*], *auto*)

lemma *zero-less-mult-pos*:
$0 < a * b \Longrightarrow 0 < a \Longrightarrow 0 < b$
apply (*cases b≤0*)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

lemma *zero-less-mult-pos2*:
$0 < b * a \Longrightarrow 0 < a \Longrightarrow 0 < b$
apply (*cases b≤0*)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg2* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes $a < b$ **and** $c < d$ **and** $0 < b$ **and** $0 \leq c$
shows $a * c < b * d$
using *assms* **apply** (*cases c=0*)
apply (*simp add*: *mult-pos-pos*)
apply (*erule mult-strict-right-mono* [*THEN less-trans*])
apply (*force simp add*: *le-less*)
apply (*erule mult-strict-left-mono*, *assumption*)
done

This weaker variant has more natural premises

lemma *mult-strict-mono'*:
assumes $a < b$ **and** $c < d$ **and** $0 \leq a$ **and** $0 \leq c$
shows $a * c < b * d$
by (*rule mult-strict-mono*) (*insert assms*, *auto*)

lemma *mult-less-le-imp-less*:
assumes $a < b$ **and** $c \leq d$ **and** $0 \leq a$ **and** $0 < c$
shows $a * c < b * d$
using *assms* **apply** (*subgoal-tac* $a * c < b * c$)
apply (*erule less-le-trans*)
apply (*erule mult-left-mono*)
apply *simp*
apply (*erule mult-strict-right-mono*)
apply *assumption*
done

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
using *assms* **apply** (*subgoal-tac* $a * c \leq b * c$)
apply (*erule le-less-trans*)
apply (*erule mult-strict-left-mono*)
apply *simp*
apply (*erule mult-right-mono*)
apply *simp*
done

lemma *mult-less-imp-less-left*:
assumes *less*: $c * a < c * b$ **and** *nonneg*: $0 \leq c$
shows $a < b$
proof (*rule ccontr*)
assume $\neg\ a < b$
hence $b \leq a$ **by** (*simp add*: *linorder-not-less*)
hence $c * b \leq c * a$ **using** *nonneg* **by** (*rule mult-left-mono*)
with *this* **and** *less* **show** *False* **by** (*simp add*: *not-less* [*symmetric*])
qed

lemma *mult-less-imp-less-right*:
assumes *less*: $a * c < b * c$ **and** *nonneg*: $0 \leq c$
shows $a < b$
proof (*rule ccontr*)
assume $\neg\ a < b$
hence $b \leq a$ **by** (*simp add*: *linorder-not-less*)
hence $b * c \leq a * c$ **using** *nonneg* **by** (*rule mult-right-mono*)
with *this* **and** *less* **show** *False* **by** (*simp add*: *not-less* [*symmetric*])
qed

end

class *mult-mono1* = *times* + *zero* + *ord* +
assumes *mult-mono1*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$

class *pordered-comm-semiring* = *comm-semiring-0*
+ *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-semiring*
proof
fix *a b c* :: $'a$
assume $a \leq b$ $0 \leq c$
thus $c * a \leq c * b$ **by** (*rule mult-mono1*)
thus $a * c \leq b * c$ **by** (*simp only*: *mult-commute*)
qed

end

class *pordered-cancel-comm-semiring* = *comm-semiring-0-cancel*
+ *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-comm-semiring* ..
subclass *pordered-cancel-semiring* ..

end

class *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add*
+
assumes *mult-strict-left-mono-comm*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *ordered-semiring-strict*
proof
fix $a\ b\ c :: 'a$
assume $a < b\ 0 < c$
thus $c * a < c * b$ **by** (*rule mult-strict-left-mono-comm*)
thus $a * c < b * c$ **by** (*simp only*: *mult-commute*)
qed

subclass *pordered-cancel-comm-semiring*
proof
fix $a\ b\ c :: 'a$
assume $a \leq b\ 0 \leq c$
thus $c * a \leq c * b$
unfolding *le-less*
using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
qed

end

class *pordered-ring* = *ring* + *pordered-cancel-semiring*
begin

subclass *pordered-ab-group-add* ..

Legacy - use *algebra-simps*

lemmas *ring-simps*[*noatp*] = *algebra-simps*

lemma *less-add-iff1*:
$a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$
by (*simp add*: *algebra-simps*)

lemma *less-add-iff2*:
$a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$

by (*simp add*: *algebra-simps*)

lemma *le-add-iff1*:
  $a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$
by (*simp add*: *algebra-simps*)

lemma *le-add-iff2*:
  $a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$
by (*simp add*: *algebra-simps*)

lemma *mult-left-mono-neg*:
  $b \leq a \Longrightarrow c \leq 0 \Longrightarrow c * a \leq c * b$
  apply (*drule mult-left-mono* [*of* - - *uminus c*])
  apply (*simp-all add*: *minus-mult-left* [*symmetric*])
  done

lemma *mult-right-mono-neg*:
  $b \leq a \Longrightarrow c \leq 0 \Longrightarrow a * c \leq b * c$
  apply (*drule mult-right-mono* [*of* - - *uminus c*])
  apply (*simp-all add*: *minus-mult-right* [*symmetric*])
  done

lemma *mult-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a * b$
using *mult-right-mono-neg* [*of a zero b*] **by** *simp*

lemma *split-mult-pos-le*:
  $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \Longrightarrow 0 \leq a * b$
by (*auto simp add*: *mult-nonneg-nonneg mult-nonpos-nonpos*)

end

class *abs-if* = *minus* + *uminus* + *ord* + *zero* + *abs* +
  assumes *abs-if*: $|a| = (if\ a < 0\ then\ -a\ else\ a)$

class *sgn-if* = *minus* + *uminus* + *zero* + *one* + *ord* + *sgn* +
  assumes *sgn-if*: $sgn\ x = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -1)$

lemma (**in** *sgn-if*) *sgn0*[*simp*]: *sgn 0 = 0*
by(*simp add*:*sgn-if*)

class *ordered-ring* = *ring* + *ordered-semiring*
  + *ordered-ab-group-add* + *abs-if*
begin

subclass *pordered-ring* **..**

subclass *pordered-ab-group-add-abs*
proof
  fix *a b*

show $|a + b| \leq |a| + |b|$
by (*auto simp add*: *abs-if not-less neg-less-eq-nonneg less-eq-neg-nonpos*)
(*auto simp del*: *minus-add-distrib simp add*: *minus-add-distrib* [*symmetric*]
neg-less-eq-nonneg less-eq-neg-nonpos, *auto intro*: *add-nonneg-nonneg*,
auto intro!: *less-imp-le add-neg-neg*)
qed (*auto simp add*: *abs-if less-eq-neg-nonpos neg-equal-zero*)

end

class *ordered-ring-strict* = *ring* + *ordered-semiring-strict*
+ *ordered-ab-group-add* + *abs-if*
begin

subclass *ordered-ring* **..**

lemma *mult-strict-left-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow c * a < c * b$
using *mult-strict-left-mono* [*of b a* − *c*] **by** *simp*

lemma *mult-strict-right-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow a * c < b * c$
using *mult-strict-right-mono* [*of b a* − *c*] **by** *simp*

lemma *mult-neg-neg*: $a < 0 \Longrightarrow b < 0 \Longrightarrow 0 < a * b$
using *mult-strict-right-mono-neg* [*of a zero b*] **by** *simp*

subclass *ring-no-zero-divisors*
proof
fix *a b*
assume $a \neq 0$ **then have** *A*: $a < 0 \vee 0 < a$ **by** (*simp add*: *neq-iff*)
assume $b \neq 0$ **then have** *B*: $b < 0 \vee 0 < b$ **by** (*simp add*: *neq-iff*)
have $a * b < 0 \vee 0 < a * b$
proof (*cases* $a < 0$)
case *True* **note** $A' = this$
show *?thesis* **proof** (*cases* $b < 0$)
case *True* **with** A'
show *?thesis* **by** (*auto dest*: *mult-neg-neg*)
next
case *False* **with** *B* **have** $0 < b$ **by** *auto*
with A' **show** *?thesis* **by** (*auto dest*: *mult-strict-right-mono*)
qed
next
case *False* **with** *A* **have** A': $0 < a$ **by** *auto*
show *?thesis* **proof** (*cases* $b < 0$)
case *True* **with** A'
show *?thesis* **by** (*auto dest*: *mult-strict-right-mono-neg*)
next
case *False* **with** *B* **have** $0 < b$ **by** *auto*
with A' **show** *?thesis* **by** (*auto dest*: *mult-pos-pos*)
qed

```
  qed
  then show a * b ≠ 0 by (simp add: neq-iff)
qed

lemma zero-less-mult-iff:
  0 < a * b ⟷ 0 < a ∧ 0 < b ∨ a < 0 ∧ b < 0
  apply (auto simp add: mult-pos-pos mult-neg-neg)
  apply (simp-all add: not-less le-less)
  apply (erule disjE) apply assumption defer
  apply (erule disjE) defer apply (drule sym) apply simp
  apply (erule disjE) defer apply (drule sym) apply simp
  apply (erule disjE) apply assumption apply (drule sym) apply simp
  apply (drule sym) apply simp
  apply (blast dest: zero-less-mult-pos)
  apply (blast dest: zero-less-mult-pos2)
  done

lemma zero-le-mult-iff:
  0 ≤ a * b ⟷ 0 ≤ a ∧ 0 ≤ b ∨ a ≤ 0 ∧ b ≤ 0
by (auto simp add: eq-commute [of 0] le-less not-less zero-less-mult-iff)

lemma mult-less-0-iff:
  a * b < 0 ⟷ 0 < a ∧ b < 0 ∨ a < 0 ∧ 0 < b
  apply (insert zero-less-mult-iff [of -a b])
  apply (force simp add: minus-mult-left[symmetric])
  done

lemma mult-le-0-iff:
  a * b ≤ 0 ⟷ 0 ≤ a ∧ b ≤ 0 ∨ a ≤ 0 ∧ 0 ≤ b
  apply (insert zero-le-mult-iff [of -a b])
  apply (force simp add: minus-mult-left[symmetric])
  done

lemma zero-le-square [simp]: 0 ≤ a * a
by (simp add: zero-le-mult-iff linear)

lemma not-square-less-zero [simp]: ¬ (a * a < 0)
by (simp add: not-less)
```

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations $\leq$ and equality.

These "disjunction" versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

```
lemma mult-less-cancel-right-disj:
  a * c < b * c ⟷ 0 < c ∧ a < b ∨ c < 0 ∧  b < a
  apply (cases c = 0)
  apply (auto simp add: neq-iff mult-strict-right-mono
                        mult-strict-right-mono-neg)
```

apply (*auto simp add*: *not-less*
not-le [*symmetric*, *of* $a*c$]
not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *less-imp-le mult-right-mono*
mult-right-mono-neg)
done

lemma *mult-less-cancel-left-disj*:
$c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge\ b < a$
apply (*cases* $c = 0$)
apply (*auto simp add*: *neq-iff mult-strict-left-mono*
mult-strict-left-mono-neg)
apply (*auto simp add*: *not-less*
not-le [*symmetric*, *of* $c*a$]
not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *less-imp-le mult-left-mono*
mult-left-mono-neg)
done

The "conjunction of implication" lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma *mult-less-cancel-right*:
$a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-right-disj* [*of* a c b] **by** *auto*

lemma *mult-less-cancel-left*:
$c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-left-disj* [*of* c a b] **by** *auto*

lemma *mult-le-cancel-right*:
$a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-right-disj*)

lemma *mult-le-cancel-left*:
$c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-left-disj*)

lemma *mult-le-cancel-left-pos*:
$0 < c \Longrightarrow c * a \leq c * b \longleftrightarrow a \leq b$
by (*auto simp*: *mult-le-cancel-left*)

lemma *mult-le-cancel-left-neg*:
$c < 0 \Longrightarrow c * a \leq c * b \longleftrightarrow b \leq a$
by (*auto simp*: *mult-le-cancel-left*)

lemma *mult-less-cancel-left-pos*:
$0 < c \Longrightarrow c * a < c * b \longleftrightarrow a < b$

by (*auto simp*: *mult-less-cancel-left*)

lemma *mult-less-cancel-left-neg*:
$c < 0 \Longrightarrow c * a < c * b \longleftrightarrow b < a$
by (*auto simp*: *mult-less-cancel-left*)

end

Legacy - use *algebra-simps*

lemmas *ring-simps*[*noatp*] = *algebra-simps*

lemmas *mult-sign-intros* =
mult-nonneg-nonneg mult-nonneg-nonpos
mult-nonpos-nonneg mult-nonpos-nonpos
mult-pos-pos mult-pos-neg
mult-neg-pos mult-neg-neg

class *pordered-comm-ring* = *comm-ring* + *pordered-comm-semiring*
begin

subclass *pordered-ring* ..
subclass *pordered-cancel-comm-semiring* ..

end

class *ordered-semidom* = *comm-semiring-1-cancel* + *ordered-comm-semiring-strict*
+

assumes *zero-less-one* [*simp*]: $0 < 1$
begin

lemma *pos-add-strict*:
shows $0 < a \Longrightarrow b < c \Longrightarrow b < a + c$
using *add-strict-mono* [*of zero a b c*] **by** *simp*

lemma *zero-le-one* [*simp*]: $0 \leq 1$
by (*rule zero-less-one* [*THEN less-imp-le*])

lemma *not-one-le-zero* [*simp*]: $\neg\ 1 \leq 0$
by (*simp add*: *not-le*)

lemma *not-one-less-zero* [*simp*]: $\neg\ 1 < 0$
by (*simp add*: *not-less*)

lemma *less-1-mult*:
assumes $1 < m$ **and** $1 < n$
shows $1 < m * n$
using *assms mult-strict-mono* [*of 1 m 1 n*]
by (*simp add*: *less-trans* [*OF zero-less-one*])

end

class *ordered-idom* = *comm-ring-1* +
ordered-comm-semiring-strict + *ordered-ab-group-add* +
abs-if + *sgn-if*

begin

subclass *ordered-ring-strict* **..**
subclass *pordered-comm-ring* **..**
subclass *idom* **..**

subclass *ordered-semidom*
proof
have $0 \leq 1 * 1$ **by** (*rule zero-le-square*)
thus $0 < 1$ **by** (*simp add*: *le-less*)
qed

lemma *linorder-neqE-ordered-idom*:
assumes $x \neq y$ **obtains** $x < y \mid y < x$
using *assms* **by** (*rule neqE*)

These cancellation simprules also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*:
$c \leq b * c \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
by (*insert mult-le-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-le-cancel-right2*:
$a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
by (*insert mult-le-cancel-right* [*of a c 1*], *simp*)

lemma *mult-le-cancel-left1*:
$c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
by (*insert mult-le-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-le-cancel-left2*:
$c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
by (*insert mult-le-cancel-left* [*of c a 1*], *simp*)

lemma *mult-less-cancel-right1*:
$c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
by (*insert mult-less-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-less-cancel-right2*:
$a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
by (*insert mult-less-cancel-right* [*of a c 1*], *simp*)

lemma *mult-less-cancel-left1*:
  $c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
by (*insert mult-less-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-less-cancel-left2*:
  $c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
by (*insert mult-less-cancel-left* [*of c a 1*], *simp*)

lemma *sgn-sgn* [*simp*]:
  sgn (*sgn a*) = *sgn a*
unfolding *sgn-if* **by** *simp*

lemma *sgn-0-0*:
  sgn a = *0* $\longleftrightarrow$ *a* = *0*
unfolding *sgn-if* **by** *simp*

lemma *sgn-1-pos*:
  sgn a = *1* $\longleftrightarrow$ *a* > *0*
unfolding *sgn-if* **by** (*simp add*: *neg-equal-zero*)

lemma *sgn-1-neg*:
  sgn a = − *1* $\longleftrightarrow$ *a* < *0*
unfolding *sgn-if* **by** (*auto simp add*: *equal-neg-zero*)

lemma *sgn-pos* [*simp*]:
  0 < *a* $\Longrightarrow$ *sgn a* = *1*
unfolding *sgn-1-pos* **.**

lemma *sgn-neg* [*simp*]:
  a < *0* $\Longrightarrow$ *sgn a* = − *1*
unfolding *sgn-1-neg* **.**

lemma *sgn-times*:
  sgn (*a* * *b*) = *sgn a* * *sgn b*
by (*auto simp add*: *sgn-if zero-less-mult-iff*)

lemma *abs-sgn*: *abs k* = *k* * *sgn k*
unfolding *sgn-if abs-if* **by** *auto*

lemma *sgn-greater* [*simp*]:
  0 < *sgn a* $\longleftrightarrow$ *0* < *a*
  unfolding *sgn-if* **by** *auto*

lemma *sgn-less* [*simp*]:
  sgn a < *0* $\longleftrightarrow$ *a* < *0*
  unfolding *sgn-if* **by** *auto*

lemma *abs-dvd-iff* [*simp*]: (*abs m*) *dvd k* $\longleftrightarrow$ *m dvd k*
  by (*simp add*: *abs-if*)

lemma *dvd-abs-iff* [*simp*]: *m dvd* (*abs k*) ⟷ *m dvd k*
by (*simp add*: *abs-if*)

end

class *ordered-field* = *field* + *ordered-idom*

Simprules for comparisons where common factors can be cancelled.

lemmas *mult-compare-simps*[*noatp*] =
mult-le-cancel-right mult-le-cancel-left
mult-le-cancel-right1 mult-le-cancel-right2
mult-le-cancel-left1 mult-le-cancel-left2
mult-less-cancel-right mult-less-cancel-left
mult-less-cancel-right1 mult-less-cancel-right2
mult-less-cancel-left1 mult-less-cancel-left2
mult-cancel-right mult-cancel-left
mult-cancel-right1 mult-cancel-right2
mult-cancel-left1 mult-cancel-left2

— FIXME continue localization here

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
(*inverse a* = *0*) = (*a* = (*0*::*'a*::{*division-ring*,*division-by-zero*}))
by (*force dest*: *inverse-zero-imp-zero*)

lemma *inverse-minus-eq* [*simp*]:
inverse(−*a*) = −*inverse*(*a*::*'a*::{*division-ring*,*division-by-zero*})
proof *cases*
assume *a*=*0* **thus** *?thesis* **by** (*simp add*: *inverse-zero*)
next
assume *a*≠*0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-minus-eq*)
qed

lemma *inverse-eq-imp-eq*:
inverse a = *inverse b* ==> *a* = (*b*::*'a*::{*division-ring*,*division-by-zero*})
apply (*cases a*=*0* | *b*=*0*)
apply (*force dest*!: *inverse-zero-imp-zero*
simp add: *eq-commute* [*of 0*::*'a*])
apply (*force dest*!: *nonzero-inverse-eq-imp-eq*)
done

lemma *inverse-eq-iff-eq* [*simp*]:
(*inverse a* = *inverse b*) = (*a* = (*b*::*'a*::{*division-ring*,*division-by-zero*}))
by (*force dest*!: *inverse-eq-imp-eq*)

lemma *inverse-inverse-eq* [*simp*]:
inverse(*inverse* (*a*::*'a*::{*division-ring*,*division-by-zero*})) = *a*

proof *cases*
assume *a=0* **thus** *?thesis* **by** *simp*
next
assume *a≠0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [*simp*]:
inverse(*a*∗*b*) = *inverse*(*a*) ∗ *inverse*(*b*::′*a*::{*field*,*division-by-zero*})
proof *cases*
assume *a* ≠ *0* & *b* ≠ *0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-mult-distrib mult-commute*)
next
assume ~ (*a* ≠ *0* & *b* ≠ *0*)
thus *?thesis* **by** *force*
qed

lemma *inverse-divide* [*simp*]:
inverse (*a*/*b*) = *b* / (*a*::′*a*::{*field*,*division-by-zero*})
by (*simp add*: *divide-inverse mult-commute*)

11.1 Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

lemma *mult-divide-mult-cancel-left*:
c≠0 ==> (*c*∗*a*) / (*c*∗*b*) = *a* / (*b*::′*a*::{*field*,*division-by-zero*})
apply (*cases b* = *0*)
apply (*simp-all add*: *nonzero-mult-divide-mult-cancel-left*)
done

lemma *mult-divide-mult-cancel-right*:
c≠0 ==> (*a*∗*c*) / (*b*∗*c*) = *a* / (*b*::′*a*::{*field*,*division-by-zero*})
apply (*cases b* = *0*)
apply (*simp-all add*: *nonzero-mult-divide-mult-cancel-right*)
done

lemma *divide-divide-eq-right* [*simp*,*noatp*]:
a / (*b*/*c*) = (*a*∗*c*) / (*b*::′*a*::{*field*,*division-by-zero*})
by (*simp add*: *divide-inverse mult-ac*)

lemma *divide-divide-eq-left* [*simp*,*noatp*]:
(*a* / *b*) / (*c*::′*a*::{*field*,*division-by-zero*}) = *a* / (*b*∗*c*)
by (*simp add*: *divide-inverse mult-assoc*)

11.1.1 Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if*[*simp*,*noatp*]:
fixes $c :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $(c*a) \,/\, (c*b) = (if\ c=0\ then\ 0\ else\ a/b)$
by (*simp add*: *mult-divide-mult-cancel-left*)

11.2 Division and Unary Minus

lemma *minus-divide-right*: $-(a/b) = a \,/\, -(b::'a::\{field, division\text{-}by\text{-}zero\})$
by (*simp add*: *divide-inverse*)

lemma *divide-minus-right* [*simp*, *noatp*]:
$a \,/\, -(b::'a::\{field, division\text{-}by\text{-}zero\}) = -(a \,/\, b)$
by (*simp add*: *divide-inverse*)

lemma *minus-divide-divide*:
$(-a)/(-b) = a \,/\, (b::'a::\{field, division\text{-}by\text{-}zero\})$
apply (*cases* $b=0$, *simp*)
apply (*simp add*: *nonzero-minus-divide-divide*)
done

lemma *eq-divide-eq*:
$((a::'a::\{field, division\text{-}by\text{-}zero\}) = b/c) = (if\ c\neq 0\ then\ a*c = b\ else\ a=0)$
by (*simp add*: *nonzero-eq-divide-eq*)

lemma *divide-eq-eq*:
$(b/c = (a::'a::\{field, division\text{-}by\text{-}zero\})) = (if\ c\neq 0\ then\ b = a*c\ else\ a=0)$
by (*force simp add*: *nonzero-divide-eq-eq*)

11.3 Ordered Fields

lemma *positive-imp-inverse-positive*:
assumes *a-gt-0*: $0 < a$ **shows** $0 < inverse\ (a::'a::ordered\text{-}field)$
proof −
have $0 < a * inverse\ a$
by (*simp add*: *a-gt-0* [*THEN order-less-imp-not-eq2*] *zero-less-one*)
thus $0 < inverse\ a$
by (*simp add*: *a-gt-0* [*THEN order-less-not-sym*] *zero-less-mult-iff*)
qed

lemma *negative-imp-inverse-negative*:
$a < 0 \Longrightarrow inverse\ a < (0::'a::ordered\text{-}field)$
by (*insert positive-imp-inverse-positive* [*of* $-a$],
simp add: *nonzero-inverse-minus-eq order-less-imp-not-eq*)

lemma *inverse-le-imp-le*:
assumes *invle*: $inverse\ a \leq inverse\ b$ **and** *apos*: $0 < a$
shows $b \leq (a::'a::ordered\text{-}field)$
proof (*rule classical*)

assume $\sim\ b \leq a$
hence $a < b$ **by** (*simp add*: *linorder-not-le*)
hence *bpos*: $0 < b$ **by** (*blast intro*: *apos order-less-trans*)
hence $a * inverse\ a \leq a * inverse\ b$
by (*simp add*: *apos invle order-less-imp-le mult-left-mono*)
hence $(a * inverse\ a) * b \leq (a * inverse\ b) * b$
by (*simp add*: *bpos order-less-imp-le mult-right-mono*)
thus $b \leq a$ **by** (*simp add*: *mult-assoc apos bpos order-less-imp-not-eq2*)
qed

lemma *inverse-positive-imp-positive*:
assumes *inv-gt-0*: $0 < inverse\ a$ **and** *nz*: $a \neq 0$
shows $0 < (a::'a::ordered\text{-}field)$
proof −
have $0 < inverse\ (inverse\ a)$
using *inv-gt-0* **by** (*rule positive-imp-inverse-positive*)
thus $0 < a$
using *nz* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-positive-iff-positive* [*simp*]:
$(0 < inverse\ a) = (0 < (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
apply (*cases* $a = 0$, *simp*)
apply (*blast intro*: *inverse-positive-imp-positive positive-imp-inverse-positive*)
done

lemma *inverse-negative-imp-negative*:
assumes *inv-less-0*: $inverse\ a < 0$ **and** *nz*: $a \neq 0$
shows $a < (0::'a::ordered\text{-}field)$
proof −
have $inverse\ (inverse\ a) < 0$
using *inv-less-0* **by** (*rule negative-imp-inverse-negative*)
thus $a < 0$ **using** *nz* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-negative-iff-negative* [*simp*]:
$(inverse\ a < 0) = (a < (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
apply (*cases* $a = 0$, *simp*)
apply (*blast intro*: *inverse-negative-imp-negative negative-imp-inverse-negative*)
done

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
$(0 \leq inverse\ a) = (0 \leq (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
$(inverse\ a \leq 0) = (a \leq (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *ordered-field-no-lb*: $\forall\ x.\ \exists\, y.\ y < (x::'a::ordered\text{-}field)$
proof
  fix $x::'a$
  have *m1*: $-\ (1::'a) < 0$ **by** *simp*
  from *add-strict-right-mono*[*OF m1*, **where** $c{=}x$]
  have $(-\ 1) + x < x$ **by** *simp*
  thus $\exists\, y.\ y < x$ **by** *blast*
qed

lemma *ordered-field-no-ub*: $\forall\ x.\ \exists\, y.\ y > (x::'a::ordered\text{-}field)$
proof
  fix $x::'a$
  have *m1*: $(1::'a) > 0$ **by** *simp*
  from *add-strict-right-mono*[*OF m1*, **where** $c{=}x$]
  have $1 + x > x$ **by** *simp*
  thus $\exists\, y.\ y > x$ **by** *blast*
qed

11.4 Anti-Monotonicity of *inverse*

lemma *less-imp-inverse-less*:
assumes *less*: $a < b$ **and** *apos*: $0 < a$
shows *inverse* $b <$ *inverse* $(a::'a::ordered\text{-}field)$
proof (*rule ccontr*)
  assume $\sim$ *inverse* $b <$ *inverse* a
  hence *inverse* $a \leq$ *inverse* b **by** (*simp add*: *linorder-not-less*)
  hence $\sim (a < b)$
    by (*simp add*: *linorder-not-less inverse-le-imp-le* [*OF* - *apos*])
  thus *False* **by** (*rule notE* [*OF* - *less*])
qed

lemma *inverse-less-imp-less*:
  $[|$*inverse* $a <$ *inverse* b; $0 < a|] ==> b < (a::'a::ordered\text{-}field)$
apply (*simp add*: *order-less-le* [*of inverse a*] *order-less-le* [*of b*])
apply (*force dest*!: *inverse-le-imp-le nonzero-inverse-eq-imp-eq*)
done

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*,*noatp*]:
  $[|0 < a;\ 0 < b|] ==>$ (*inverse* $a <$ *inverse* $b) = (b < (a::'a::ordered\text{-}field))$
by (*blast intro*: *less-imp-inverse-less dest*: *inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
  $[|a \leq b;\ 0 < a|] ==>$ *inverse* $b \leq$ *inverse* $(a::'a::ordered\text{-}field)$
by (*force simp add*: *order-le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le* [*simp*,*noatp*]:
 $[|0 < a;\ 0 < b|] ==>$ (*inverse* $a \leq$ *inverse* $b) = (b \leq (a::'a::ordered\text{-}field))$
by (*blast intro*: *le-imp-inverse-le dest*: *inverse-le-imp-le*)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
[|*inverse a* ≤ *inverse b*; *b* < *0*|] ==> *b* ≤ (*a*::'*a*::*ordered-field*)
apply (*rule classical*)
apply (*subgoal-tac a* < *0*)
prefer *2* **apply** (*force simp add*: *linorder-not-le intro*: *order-less-trans*)
apply (*insert inverse-le-imp-le* [*of* −*b* −*a*])
apply (*simp add*: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *less-imp-inverse-less-neg*:
[|*a* < *b*; *b* < *0*|] ==> *inverse b* < *inverse* (*a*::'*a*::*ordered-field*)
apply (*subgoal-tac a* < *0*)
prefer *2* **apply** (*blast intro*: *order-less-trans*)
apply (*insert less-imp-inverse-less* [*of* −*b* −*a*])
apply (*simp add*: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *inverse-less-imp-less-neg*:
[|*inverse a* < *inverse b*; *b* < *0*|] ==> *b* < (*a*::'*a*::*ordered-field*)
apply (*rule classical*)
apply (*subgoal-tac a* < *0*)
prefer *2*
apply (*force simp add*: *linorder-not-less intro*: *order-le-less-trans*)
apply (*insert inverse-less-imp-less* [*of* −*b* −*a*])
apply (*simp add*: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *inverse-less-iff-less-neg* [*simp*,*noatp*]:
[|*a* < *0*; *b* < *0*|] ==> (*inverse a* < *inverse b*) = (*b* < (*a*::'*a*::*ordered-field*))
apply (*insert inverse-less-iff-less* [*of* −*b* −*a*])
apply (*simp del*: *inverse-less-iff-less*
add: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *le-imp-inverse-le-neg*:
[|*a* ≤ *b*; *b* < *0*|] ==> *inverse b* ≤ *inverse* (*a*::'*a*::*ordered-field*)
by (*force simp add*: *order-le-less less-imp-inverse-less-neg*)

lemma *inverse-le-iff-le-neg* [*simp*,*noatp*]:
[|*a* < *0*; *b* < *0*|] ==> (*inverse a* ≤ *inverse b*) = (*b* ≤ (*a*::'*a*::*ordered-field*))
by (*blast intro*: *le-imp-inverse-le-neg dest*: *inverse-le-imp-le-neg*)

11.5 Inverses and the Number One

lemma *one-less-inverse-iff*:
(*1* < *inverse x*) = (*0* < *x* & *x* < (*1*::'*a*::{*ordered-field*,*division-by-zero*}))
proof *cases*

```
  assume 0 < x
    with inverse-less-iff-less [OF zero-less-one, of x]
    show ?thesis by simp
next
  assume notless: ~ (0 < x)
  have ~ (1 < inverse x)
  proof
    assume 1 < inverse x
    also with notless have ... ≤ 0 by (simp add: linorder-not-less)
    also have ... < 1 by (rule zero-less-one)
    finally show False by auto
  qed
  with notless show ?thesis by simp
qed

lemma inverse-eq-1-iff [simp]:
  (inverse x = 1) = (x = (1::'a::{field,division-by-zero}))
by (insert inverse-eq-iff-eq [of x 1], simp)

lemma one-le-inverse-iff:
  (1 ≤ inverse x) = (0 < x & x ≤ (1::'a::{ordered-field,division-by-zero}))
by (force simp add: order-le-less one-less-inverse-iff zero-less-one
                    eq-commute [of 1])

lemma inverse-less-1-iff:
  (inverse x < 1) = (x ≤ 0 | 1 < (x::'a::{ordered-field,division-by-zero}))
by (simp add: linorder-not-le [symmetric] one-le-inverse-iff)

lemma inverse-le-1-iff:
  (inverse x ≤ 1) = (x ≤ 0 | 1 ≤ (x::'a::{ordered-field,division-by-zero}))
by (simp add: linorder-not-less [symmetric] one-less-inverse-iff)
```

11.6 Simplification of Inequalities Involving Literal Divisors

```
lemma pos-le-divide-eq: 0 < (c::'a::ordered-field) ==> (a ≤ b/c) = (a*c ≤ b)
proof -
  assume less: 0<c
  hence (a ≤ b/c) = (a*c ≤ (b/c)*c)
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
  also have ... = (a*c ≤ b)
    by (simp add: order-less-imp-not-eq2 [OF less] divide-inverse mult-assoc)
  finally show ?thesis .
qed

lemma neg-le-divide-eq: c < (0::'a::ordered-field) ==> (a ≤ b/c) = (b ≤ a*c)
proof -
  assume less: c<0
  hence (a ≤ b/c) = ((b/c)*c ≤ a*c)
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
```

also have ... = $(b \leq a*c)$
by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *le-divide-eq*:
$(a \leq b/c) =$
(*if* $0 < c$ *then* $a*c \leq b$
else if $c < 0$ *then* $b \leq a*c$
else $a \leq (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})$)
apply (*cases c=0*, *simp*)
apply (*force simp add*: *pos-le-divide-eq neg-le-divide-eq linorder-neq-iff*)
done

lemma *pos-divide-le-eq*: $0 < (c::'a::ordered\text{-}field) ==> (b/c \leq a) = (b \leq a*c)$
proof −
assume *less*: *0<c*
hence $(b/c \leq a) = ((b/c)*c \leq a*c)$
by (*simp add*: *mult-le-cancel-right order-less-not-sym* [*OF less*])
also have ... = $(b \leq a*c)$
by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-divide-le-eq*: $c < (0::'a::ordered\text{-}field) ==> (b/c \leq a) = (a*c \leq b)$
proof −
assume *less*: *c<0*
hence $(b/c \leq a) = (a*c \leq (b/c)*c)$
by (*simp add*: *mult-le-cancel-right order-less-not-sym* [*OF less*])
also have ... = $(a*c \leq b)$
by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *divide-le-eq*:
$(b/c \leq a) =$
(*if* $0 < c$ *then* $b \leq a*c$
else if $c < 0$ *then* $a*c \leq b$
else $0 \leq (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})$)
apply (*cases c=0*, *simp*)
apply (*force simp add*: *pos-divide-le-eq neg-divide-le-eq linorder-neq-iff*)
done

lemma *pos-less-divide-eq*:
$0 < (c::'a::ordered\text{-}field) ==> (a < b/c) = (a*c < b)$
proof −
assume *less*: *0<c*
hence $(a < b/c) = (a*c < (b/c)*c)$
by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])

also have ... = $(a*c < b)$
by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-less-divide-eq*:
$c < (0::'a::ordered\text{-}field) ==> (a < b/c) = (b < a*c)$
proof −
assume *less*: *c<0*
hence $(a < b/c) = ((b/c)*c < a*c)$
by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
also have ... = $(b < a*c)$
by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *less-divide-eq*:
$(a < b/c) =$
(*if* $0 < c$ *then* $a*c < b$
else if $c < 0$ *then* $b < a*c$
else $a < (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})$)
apply (*cases c=0*, *simp*)
apply (*force simp add*: *pos-less-divide-eq neg-less-divide-eq linorder-neq-iff*)
done

lemma *pos-divide-less-eq*:
$0 < (c::'a::ordered\text{-}field) ==> (b/c < a) = (b < a*c)$
proof −
assume *less*: *0<c*
hence $(b/c < a) = ((b/c)*c < a*c)$
by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
also have ... = $(b < a*c)$
by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-divide-less-eq*:
$c < (0::'a::ordered\text{-}field) ==> (b/c < a) = (a*c < b)$
proof −
assume *less*: *c<0*
hence $(b/c < a) = (a*c < (b/c)*c)$
by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
also have ... = $(a*c < b)$
by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *divide-less-eq*:
$(b/c < a) =$

(*if* $0 < c$ *then* $b < a*c$
else if $c < 0$ *then* $a*c < b$
else $0 < (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-divide-less-eq neg-divide-less-eq linorder-neq-iff*)
done

11.7 Field simplification

Lemmas *field-simps* multiply with denominators in in(equations) if they can be proved to be non-zero (for equations) or positive/negative (for inequations). Can be too aggressive and is therefore separate from the more benign *algebra-simps*.

lemmas *field-simps*[*noatp*] = *field-eq-simps*

pos-divide-less-eq neg-divide-less-eq
pos-less-divide-eq neg-less-divide-eq
pos-divide-le-eq neg-divide-le-eq
pos-le-divide-eq neg-le-divide-eq

Lemmas *sign-simps* is a first attempt to automate proofs of positivity/negativity needed for *field-simps*. Have not added *sign-simps* to *field-simps* because the former can lead to case explosions.

lemmas *sign-simps*[*noatp*] = *group-simps*
zero-less-mult-iff *mult-less-0-iff*

11.8 Division and Signs

lemma *zero-less-divide-iff*:
$((0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}) < a/b) = (0 < a \;\&\; 0 < b \mid a < 0 \;\&\; b < 0)$
by (*simp add*: *divide-inverse zero-less-mult-iff*)

lemma *divide-less-0-iff*:
$(a/b < (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})) =$
$(0 < a \;\&\; b < 0 \mid a < 0 \;\&\; 0 < b)$
by (*simp add*: *divide-inverse mult-less-0-iff*)

lemma *zero-le-divide-iff*:
$((0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}) \leq a/b) =$
$(0 \leq a \;\&\; 0 \leq b \mid a \leq 0 \;\&\; b \leq 0)$
by (*simp add*: *divide-inverse zero-le-mult-iff*)

lemma *divide-le-0-iff*:
$(a/b \leq (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})) =$
$(0 \leq a \;\&\; b \leq 0 \mid a \leq 0 \;\&\; 0 \leq b)$
by (*simp add*: *divide-inverse mult-le-0-iff*)

lemma *divide-eq-0-iff* [*simp*,*noatp*]:
(*a*/*b* = *0*) = (*a*=*0* | *b*=(*0*::'*a*::{*field*,*division-by-zero*}))
by (*simp add*: *divide-inverse*)

lemma *divide-pos-pos*:
0 < (*x*::'*a*::*ordered-field*) ==> *0* < *y* ==> *0* < *x* / *y*
by(*simp add*:*field-simps*)

lemma *divide-nonneg-pos*:
0 <= (*x*::'*a*::*ordered-field*) ==> *0* < *y* ==> *0* <= *x* / *y*
by(*simp add*:*field-simps*)

lemma *divide-neg-pos*:
(*x*::'*a*::*ordered-field*) < *0* ==> *0* < *y* ==> *x* / *y* < *0*
by(*simp add*:*field-simps*)

lemma *divide-nonpos-pos*:
(*x*::'*a*::*ordered-field*) <= *0* ==> *0* < *y* ==> *x* / *y* <= *0*
by(*simp add*:*field-simps*)

lemma *divide-pos-neg*:
0 < (*x*::'*a*::*ordered-field*) ==> *y* < *0* ==> *x* / *y* < *0*
by(*simp add*:*field-simps*)

lemma *divide-nonneg-neg*:
0 <= (*x*::'*a*::*ordered-field*) ==> *y* < *0* ==> *x* / *y* <= *0*
by(*simp add*:*field-simps*)

lemma *divide-neg-neg*:
(*x*::'*a*::*ordered-field*) < *0* ==> *y* < *0* ==> *0* < *x* / *y*
by(*simp add*:*field-simps*)

lemma *divide-nonpos-neg*:
(*x*::'*a*::*ordered-field*) <= *0* ==> *y* < *0* ==> *0* <= *x* / *y*
by(*simp add*:*field-simps*)

11.9 Cancellation Laws for Division

lemma *divide-cancel-right* [*simp*,*noatp*]:
(*a*/*c* = *b*/*c*) = (*c* = *0* | *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)
done

lemma *divide-cancel-left* [*simp*,*noatp*]:
(*c*/*a* = *c*/*b*) = (*c* = *0* | *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)

done

11.10 Division and the Number One

Simplify expressions equated with 1

lemma *divide-eq-1-iff* [*simp*,*noatp*]:
(*a*/*b* = *1*) = (*b* ≠ *0* & *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*cases b=0*, *simp*)
apply (*simp add*: *right-inverse-eq*)
done

lemma *one-eq-divide-iff* [*simp*,*noatp*]:
(*1* = *a*/*b*) = (*b* ≠ *0* & *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
by (*simp add*: *eq-commute* [*of 1*])

lemma *zero-eq-1-divide-iff* [*simp*,*noatp*]:
((*0*::'*a*::{*ordered-field*,*division-by-zero*}) = *1*/*a*) = (*a* = *0*)
apply (*cases a=0*, *simp*)
apply (*auto simp add*: *nonzero-eq-divide-eq*)
done

lemma *one-divide-eq-0-iff* [*simp*,*noatp*]:
(*1*/*a* = (*0*::'*a*::{*ordered-field*,*division-by-zero*})) = (*a* = *0*)
apply (*cases a=0*, *simp*)
apply (*insert zero-neq-one* [*THEN not-sym*])
apply (*auto simp add*: *nonzero-divide-eq-eq*)
done

Simplify expressions such as *0* < *1*/*x* to *0* < *x*

lemmas *zero-less-divide-1-iff* = *zero-less-divide-iff* [*of 1*, *simplified*]
lemmas *divide-less-0-1-iff* = *divide-less-0-iff* [*of 1*, *simplified*]
lemmas *zero-le-divide-1-iff* = *zero-le-divide-iff* [*of 1*, *simplified*]
lemmas *divide-le-0-1-iff* = *divide-le-0-iff* [*of 1*, *simplified*]

declare *zero-less-divide-1-iff* [*simp*,*noatp*]
declare *divide-less-0-1-iff* [*simp*,*noatp*]
declare *zero-le-divide-1-iff* [*simp*,*noatp*]
declare *divide-le-0-1-iff* [*simp*,*noatp*]

11.11 Ordering Rules for Division

lemma *divide-strict-right-mono*:
[|*a* < *b*; *0* < *c*|] ==> *a* / *c* < *b* / (*c*::'*a*::*ordered-field*)
by (*simp add*: *order-less-imp-not-eq2 divide-inverse mult-strict-right-mono positive-imp-inverse-positive*)

lemma *divide-right-mono*:
[|*a* ≤ *b*; *0* ≤ *c*|] ==> *a*/*c* ≤ *b*/(*c*::'*a*::{*ordered-field*,*division-by-zero*})
by (*force simp add*: *divide-strict-right-mono order-le-less*)

lemma *divide-right-mono-neg*: (*a*::'*a*::{*division-by-zero*,*ordered-field*}) <= *b*
==> *c* <= *0* ==> *b* / *c* <= *a* / *c*
apply (*drule divide-right-mono* [*of* - - − *c*])
apply *auto*
done

lemma *divide-strict-right-mono-neg*:
[|*b* < *a*; *c* < *0*|] ==> *a* / *c* < *b* / (*c*::'*a*::*ordered-field*)
apply (*drule divide-strict-right-mono* [*of* - - −*c*], *simp*)
apply (*simp add*: *order-less-imp-not-eq nonzero-minus-divide-right* [*symmetric*])
done

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:
[|*b* < *a*; *0* < *c*; *0* < *a*∗*b*|] ==> *c* / *a* < *c* / (*b*::'*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono*)

lemma *divide-left-mono*:
[|*b* ≤ *a*; *0* ≤ *c*; *0* < *a*∗*b*|] ==> *c* / *a* ≤ *c* / (*b*::'*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-right-mono*)

lemma *divide-left-mono-neg*: (*a*::'*a*::{*division-by-zero*,*ordered-field*}) <= *b*
==> *c* <= *0* ==> *0* < *a* ∗ *b* ==> *c* / *a* <= *c* / *b*
apply (*drule divide-left-mono* [*of* - - − *c*])
apply (*auto simp add*: *mult-commute*)
done

lemma *divide-strict-left-mono-neg*:
[|*a* < *b*; *c* < *0*; *0* < *a*∗*b*|] ==> *c* / *a* < *c* / (*b*::'*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono-neg*)

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*1* ≤ *b* / *a*) = ((*0* < *a* & *a* ≤ *b*) | (*a* < *0* & *b* ≤ *a*))
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*b* / *a* ≤ *1*) = ((*0* < *a* & *b* ≤ *a*) | (*a* < *0* & *a* ≤ *b*) | *a*=*0*)
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*1* < *b* / *a*) = ((*0* < *a* & *a* < *b*) | (*a* < *0* & *b* < *a*))
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1* [*noatp*]:

fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(b \,/\, a < 1) = ((0 < a \;\&\; b < a) \mid (a < 0 \;\&\; a < b) \mid a{=}0)$
by (*auto simp add*: *divide-less-eq*)

11.12 Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
by (*auto simp add*: *le-divide-eq*)

lemma *le-divide-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
by (*auto simp add*: *divide-le-eq*)

lemma *divide-le-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (1 < b/a) = (a < b)$
by (*auto simp add*: *less-divide-eq*)

lemma *less-divide-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (1 < b/a) = (b < a)$
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (b/a < 1) = (b < a)$
by (*auto simp add*: *divide-less-eq*)

lemma *divide-less-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow b/a < 1 <-> a < b$
by (*auto simp add*: *divide-less-eq*)

lemma *eq-divide-eq-1* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(1 = b/a) = ((a \neq 0 \;\&\; a = b))$

by (*auto simp add*: *eq-divide-eq*)

lemma *divide-eq-eq-1* [*simp*,*noatp*]:
  fixes *a* :: *'a* :: {*ordered-field*,*division-by-zero*}
  shows (*b*/*a* = *1*) = ((*a* ≠ *0* & *a* = *b*))
by (*auto simp add*: *divide-eq-eq*)

11.13 Reasoning about inequalities with division

lemma *mult-right-le-one-le*: *0* <= (*x*::*'a*::*ordered-idom*) ==> *0* <= *y* ==> *y* <= *1*
    ==> *x* * *y* <= *x*
by (*auto simp add*: *mult-compare-simps*)

lemma *mult-left-le-one-le*: *0* <= (*x*::*'a*::*ordered-idom*) ==> *0* <= *y* ==> *y* <= *1*
    ==> *y* * *x* <= *x*
by (*auto simp add*: *mult-compare-simps*)

lemma *mult-imp-div-pos-le*: *0* < (*y*::*'a*::*ordered-field*) ==> *x* <= *z* * *y* ==>
    x / *y* <= *z*
by (*subst pos-divide-le-eq*, *assumption*+)

lemma *mult-imp-le-div-pos*: *0* < (*y*::*'a*::*ordered-field*) ==> *z* * *y* <= *x* ==>
    z <= *x* / *y*
by(*simp add*:*field-simps*)

lemma *mult-imp-div-pos-less*: *0* < (*y*::*'a*::*ordered-field*) ==> *x* < *z* * *y* ==>
    x / *y* < *z*
by(*simp add*:*field-simps*)

lemma *mult-imp-less-div-pos*: *0* < (*y*::*'a*::*ordered-field*) ==> *z* * *y* < *x* ==>
    z < *x* / *y*
by(*simp add*:*field-simps*)

lemma *frac-le*: (*0*::*'a*::*ordered-field*) <= *x* ==>
    x <= *y* ==> *0* < *w* ==> *w* <= *z* ==> *x* / *z* <= *y* / *w*
  apply (*rule mult-imp-div-pos-le*)
  apply *simp*
  apply (*subst times-divide-eq-left*)
  apply (*rule mult-imp-le-div-pos*, *assumption*)
  apply (*rule mult-mono*)
  apply *simp-all*
done

lemma *frac-less*: (*0*::*'a*::*ordered-field*) <= *x* ==>
    x < *y* ==> *0* < *w* ==> *w* <= *z* ==> *x* / *z* < *y* / *w*
  apply (*rule mult-imp-div-pos-less*)
  apply *simp*

apply (*subst times-divide-eq-left*)
apply (*rule mult-imp-less-div-pos*, *assumption*)
apply (*erule mult-less-le-imp-less*)
apply *simp-all*
done

lemma *frac-less2*: (*0*::*'a*::*ordered-field*) < *x* ==>
x <= *y* ==> *0* < *w* ==> *w* < *z* ==> *x* / *z* < *y* / *w*
apply (*rule mult-imp-div-pos-less*)
apply *simp-all*
apply (*subst times-divide-eq-left*)
apply (*rule mult-imp-less-div-pos*, *assumption*)
apply (*erule mult-le-less-imp-less*)
apply *simp-all*
done

It's not obvious whether these should be simprules or not. Their effect is to gather terms into one big fraction, like a*b*c / x*y*z. The rationale for that is unclear, but many proofs seem to need them.

declare *times-divide-eq* [*simp*]

11.14 Ordered Fields are Dense

context *ordered-semidom*
begin

lemma *less-add-one*: *a* < *a* + *1*
proof −
have *a* + *0* < *a* + *1*
by (*blast intro*: *zero-less-one add-strict-left-mono*)
thus *?thesis* **by** *simp*
qed

lemma *zero-less-two*: *0* < *1* + *1*
by (*blast intro*: *less-trans zero-less-one less-add-one*)

end

lemma *less-half-sum*: *a* < *b* ==> *a* < (*a*+*b*) / (*1*+*1*::*'a*::*ordered-field*)
by (*simp add*: *field-simps zero-less-two*)

lemma *gt-half-sum*: *a* < *b* ==> (*a*+*b*)/(*1*+*1*::*'a*::*ordered-field*) < *b*
by (*simp add*: *field-simps zero-less-two*)

instance *ordered-field* < *dense-linear-order*
proof
fix *x y* :: *'a*
have *x* < *x* + *1* **by** *simp*
then show ∃ *y*. *x* < *y* **..**

have $x - 1 < x$ **by** *simp*
then show $\exists\, y.\ y < x$ **..**
show $x < y \Longrightarrow \exists\, z{>}x.\ z < y$ **by** (*blast intro*!: *less-half-sum gt-half-sum*)
qed

11.15 Absolute Value

context *ordered-idom*
begin

lemma *mult-sgn-abs*: *sgn x * abs x = x*
unfolding *abs-if sgn-if* **by** *auto*

end

lemma *abs-one* [*simp*]: *abs 1 = (1::'a::ordered-idom)*
by (*simp add*: *abs-if zero-less-one* [*THEN order-less-not-sym*])

class *pordered-ring-abs = pordered-ring + pordered-ab-group-add-abs +*
assumes *abs-eq-mult*:
$(0 \le a \vee a \le 0) \wedge (0 \le b \vee b \le 0) \Longrightarrow |a * b| = |a| * |b|$

class *lordered-ring = pordered-ring + lordered-ab-group-add-abs*
begin

subclass *lordered-ab-group-add-meet* **..**
subclass *lordered-ab-group-add-join* **..**

end

lemma *abs-le-mult*: *abs (a * b) ≤ (abs a) * (abs (b::'a::lordered-ring))*
proof −
let *?x = pprt a * pprt b − pprt a * nprt b − nprt a * pprt b + nprt a * nprt b*
let *?y = pprt a * pprt b + pprt a * nprt b + nprt a * pprt b + nprt a * nprt b*
have *a*: *(abs a) * (abs b) = ?x*
by (*simp only*: *abs-prts*[*of a*] *abs-prts*[*of b*] *algebra-simps*)
{
fix *u v* :: *'a*
have *bh*: $[\![u = a;\ v = b]\!] \Longrightarrow$
*u * v = pprt a * pprt b + pprt a * nprt b +*
*nprt a * pprt b + nprt a * nprt b*
apply (*subst prts*[*of u*], *subst prts*[*of v*])
apply (*simp add*: *algebra-simps*)
done
}
note *b = this*[*OF refl*[*of a*] *refl*[*of b*]]
note *addm = add-mono*[*of 0::'a - 0::'a, simplified*]
note *addm2 = add-mono*[*of - 0::'a - 0::'a, simplified*]

```
  have xy: − ?x <= ?y
    apply (simp)
    apply (rule-tac y=0::'a in order-trans)
    apply (rule addm2)
    apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
    apply (rule addm)
    apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
    done
  have yx: ?y <= ?x
    apply (simp add:diff-def)
    apply (rule-tac y=0 in order-trans)
    apply (rule addm2, (simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)+)
    apply (rule addm, (simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)+)
    done
  have i1: a*b <= abs a * abs b by (simp only: a b yx)
  have i2: − (abs a * abs b) <= a*b by (simp only: a b xy)
  show ?thesis
    apply (rule abs-leI)
    apply (simp add: i1)
    apply (simp add: i2[simplified minus-le-iff])
    done
qed

instance lordered-ring ⊆ pordered-ring-abs
proof
  fix a b :: 'a:: lordered-ring
  assume (0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0)
  show abs (a*b) = abs a * abs b
proof −
  have s: (0 <= a*b) | (a*b <= 0)
    apply (auto)
    apply (rule-tac split-mult-pos-le)
    apply (rule-tac contrapos-np[of a*b <= 0])
    apply (simp)
    apply (rule-tac split-mult-neg-le)
    apply (insert prems)
    apply (blast)
    done
  have mulprts: a * b = (pprt a + nprt a) * (pprt b + nprt b)
    by (simp add: prts[symmetric])
  show ?thesis
  proof cases
    assume 0 <= a * b
    then show ?thesis
      apply (simp-all add: mulprts abs-prts)
      apply (insert prems)
      apply (auto simp add:
        algebra-simps
        iffD1[OF zero-le-iff-zero-nprt] iffD1[OF le-zero-iff-zero-pprt]
```

```
      iffD1[OF le-zero-iff-pprt-id] iffD1[OF zero-le-iff-nprt-id])
      apply(drule (1) mult-nonneg-nonpos[of a b], simp)
      apply(drule (1) mult-nonneg-nonpos2[of b a], simp)
     done
  next
    assume ~(0 <= a*b)
    with s have a*b <= 0 by simp
    then show ?thesis
      apply (simp-all add: mulprts abs-prts)
      apply (insert prems)
      apply (auto simp add: algebra-simps)
      apply(drule (1) mult-nonneg-nonneg[of a b],simp)
      apply(drule (1) mult-nonpos-nonpos[of a b],simp)
      done
  qed
qed
qed

instance ordered-idom ⊆ pordered-ring-abs
by default (auto simp add: abs-if not-less
  equal-neg-zero neg-equal-zero mult-less-0-iff)

lemma abs-mult: abs (a * b) = abs a * abs (b::'a::ordered-idom)
by (simp add: abs-eq-mult linorder-linear)

lemma abs-mult-self: abs a * abs a = a * (a::'a::ordered-idom)
by (simp add: abs-if)

lemma nonzero-abs-inverse:
     a ≠ 0 ==> abs (inverse (a::'a::ordered-field)) = inverse (abs a)
apply (auto simp add: linorder-neq-iff abs-if nonzero-inverse-minus-eq
                      negative-imp-inverse-negative)
apply (blast intro: positive-imp-inverse-positive elim: order-less-asym)
done

lemma abs-inverse [simp]:
     abs (inverse (a::'a::{ordered-field,division-by-zero})) =
      inverse (abs a)
apply (cases a=0, simp)
apply (simp add: nonzero-abs-inverse)
done

lemma nonzero-abs-divide:
     b ≠ 0 ==> abs (a / (b::'a::ordered-field)) = abs a / abs b
by (simp add: divide-inverse abs-mult nonzero-abs-inverse)

lemma abs-divide [simp]:
     abs (a / (b::'a::{ordered-field,division-by-zero})) = abs a / abs b
apply (cases b=0, simp)
```

apply (*simp add*: *nonzero-abs-divide*)
done

lemma *abs-mult-less*:
  [| *abs a* < *c*; *abs b* < *d* |] ==> *abs a* * *abs b* < *c*(*d*::'*a*::*ordered-idom*)
proof −
  assume *ac*: *abs a* < *c*
  hence *cpos*: *0*<*c* **by** (*blast intro*: *order-le-less-trans abs-ge-zero*)
  assume *abs b* < *d*
  thus *?thesis* **by** (*simp add*: *ac cpos mult-strict-mono*)
qed

lemmas *eq-minus-self-iff*[*noatp*] = *equal-neg-zero*

lemma *less-minus-self-iff*: (*a* < −*a*) = (*a* < (*0*::'*a*::*ordered-idom*))
  unfolding *order-less-le less-eq-neg-nonpos equal-neg-zero* **..**

lemma *abs-less-iff*: (*abs a* < *b*) = (*a* < *b* & −*a* < (*b*::'*a*::*ordered-idom*))
apply (*simp add*: *order-less-le abs-le-iff*)
apply (*auto simp add*: *abs-if neg-less-eq-nonneg less-eq-neg-nonpos*)
done

lemma *abs-mult-pos*: (*0*::'*a*::*ordered-idom*) <= *x* ==>
    (*abs y*) * *x* = *abs* (*y* * *x*)
  apply (*subst abs-mult*)
  apply *simp*
done

lemma *abs-div-pos*: (*0*::'*a*::{*division-by-zero*,*ordered-field*}) < *y* ==>
    abs x / *y* = *abs* (*x* / *y*)
  apply (*subst abs-divide*)
  apply (*simp add*: *order-less-imp-le*)
done

11.16 Bounds of products via negative and positive Part

lemma *mult-le-prts*:
  assumes
  a1 <= (*a*::'*a*::*lordered-ring*)
  a <= *a2*
  b1 <= *b*
  b <= *b2*
  shows
  a * *b* <= *pprt a2* * *pprt b2* + *pprt a1* * *nprt b2* + *nprt a2* * *pprt b1* + *nprt a1*
* *nprt b1*
proof −
  have *a* * *b* = (*pprt a* + *nprt a*) * (*pprt b* + *nprt b*)
    apply (*subst prts*[*symmetric*])+
    apply *simp*

```
    done
  then have a * b = pprt a * pprt b + pprt a * nprt b + nprt a * pprt b + nprt
a * nprt b
    by (simp add: algebra-simps)
  moreover have pprt a * pprt b <= pprt a2 * pprt b2
    by (simp-all add: prems mult-mono)
  moreover have pprt a * nprt b <= pprt a1 * nprt b2
  proof -
    have pprt a * nprt b <= pprt a * nprt b2
      by (simp add: mult-left-mono prems)
    moreover have pprt a * nprt b2 <= pprt a1 * nprt b2
      by (simp add: mult-right-mono-neg prems)
    ultimately show ?thesis
      by simp
  qed
  moreover have nprt a * pprt b <= nprt a2 * pprt b1
  proof -
    have nprt a * pprt b <= nprt a2 * pprt b
      by (simp add: mult-right-mono prems)
    moreover have nprt a2 * pprt b <= nprt a2 * pprt b1
      by (simp add: mult-left-mono-neg prems)
    ultimately show ?thesis
      by simp
  qed
  moreover have nprt a * nprt b <= nprt a1 * nprt b1
  proof -
    have nprt a * nprt b <= nprt a * nprt b1
      by (simp add: mult-left-mono-neg prems)
    moreover have nprt a * nprt b1 <= nprt a1 * nprt b1
      by (simp add: mult-right-mono-neg prems)
    ultimately show ?thesis
      by simp
  qed
  ultimately show ?thesis
    by - (rule add-mono | simp)+
qed

lemma mult-ge-prts:
  assumes
  a1 <= (a::'a::lordered-ring)
  a <= a2
  b1 <= b
  b <= b2
  shows
  a * b >= nprt a1 * pprt b2 + nprt a2 * nprt b2 + pprt a1 * pprt b1 + pprt a2
* nprt b1
proof -
  from prems have a1:- a2 <= -a by auto
  from prems have a2: -a <= -a1 by auto
```

from *mult-le-prts*[*of* $-a2$ $-a$ $-a1$ *b1 b b2*, *OF a1 a2 prems*(*3*) *prems*(*4*), *simplified nprt-neg pprt-neg*]
have *le*: $-(a * b) <= -\ nprt\ a1 * pprt\ b2 + -\ nprt\ a2 * nprt\ b2 + -\ pprt\ a1 * pprt\ b1 + -\ pprt\ a2 * nprt\ b1$ **by** *simp*
then have $-(-\ nprt\ a1 * pprt\ b2 + -\ nprt\ a2 * nprt\ b2 + -\ pprt\ a1 * pprt\ b1 + -\ pprt\ a2 * nprt\ b1) <= a * b$
by (*simp only*: *minus-le-iff*)
then show *?thesis* **by** *simp*
qed

end

12 Nat: Natural numbers

theory *Nat*
imports *Inductive Ring-and-Field*
uses
~~/src/Tools/rat.ML
~~/src/Provers/Arith/cancel-sums.ML
Tools/arith-data.ML
(*Tools/nat-arith.ML*)
~~/src/Provers/Arith/fast-lin-arith.ML
(*Tools/lin-arith.ML*)
begin

12.1 Type *ind*

typedecl *ind*

axiomatization
Zero-Rep :: *ind* **and**
Suc-Rep :: $ind \Rightarrow ind$
where
— the axiom of infinity in 2 parts
inj-Suc-Rep: *inj Suc-Rep* **and**
Suc-Rep-not-Zero-Rep: $Suc\text{-}Rep\ x \neq Zero\text{-}Rep$

12.2 Type nat

Type definition

inductive *Nat* :: $ind \Rightarrow bool$
where
Zero-RepI: *Nat Zero-Rep*
| *Suc-RepI*: $Nat\ i \Longrightarrow Nat\ (Suc\text{-}Rep\ i)$

global

typedef (**open** *Nat*)
nat = *Nat*
by (*rule exI*, *unfold mem-def*, *rule Nat.Zero-RepI*)

constdefs
Suc :: *nat* => *nat*
Suc-def: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat n*)))

local

instantiation *nat* :: *zero*
begin

definition *Zero-nat-def* [*code del*]:
0 = *Abs-Nat Zero-Rep*

instance ..

end

lemma *Suc-not-Zero*: *Suc m* ≠ *0*
by (*simp add*: *Zero-nat-def Suc-def Abs-Nat-inject* [*unfolded mem-def*]
Rep-Nat [*unfolded mem-def*] *Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep* [*unfolded mem-def*])

lemma *Zero-not-Suc*: *0* ≠ *Suc m*
by (*rule not-sym*, *rule Suc-not-Zero not-sym*)

rep-datatype *0* :: *nat Suc*
apply (*unfold Zero-nat-def Suc-def*)
apply (*rule Rep-Nat-inverse* [*THEN subst*]) — types force good instantiation
apply (*erule Rep-Nat* [*unfolded mem-def*, *THEN Nat.induct*])
apply (*iprover elim*: *Abs-Nat-inverse* [*unfolded mem-def*, *THEN subst*])
apply (*simp-all add*: *Abs-Nat-inject* [*unfolded mem-def*] *Rep-Nat* [*unfolded mem-def*]
Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep [*unfolded mem-def*]
Suc-Rep-not-Zero-Rep [*unfolded mem-def*, *symmetric*]
inj-Suc-Rep [*THEN inj-eq*] *Rep-Nat-inject*)
done

lemma *nat-induct* [*case-names 0 Suc*, *induct type*: *nat*]:
— for backward compatibility – names of variables differ
fixes *n*
assumes *P 0*
and ⋀*n*. *P n* ⟹ *P* (*Suc n*)
shows *P n*
using *assms* **by** (*rule nat.induct*)

declare *nat.exhaust* [*case-names 0 Suc*, *cases type*: *nat*]

lemmas *nat-rec-0* = *nat.recs*(*1*)
  and *nat-rec-Suc* = *nat.recs*(*2*)

lemmas *nat-case-0* = *nat.cases*(*1*)
  and *nat-case-Suc* = *nat.cases*(*2*)

Injectiveness and distinctness lemmas

lemma *inj-Suc*[*simp*]: *inj-on Suc N*
  by (*simp add*: *inj-on-def*)

lemma *Suc-neq-Zero*: *Suc m* = *0* ⟹ *R*
by (*rule notE*, *rule Suc-not-Zero*)

lemma *Zero-neq-Suc*: *0* = *Suc m* ⟹ *R*
by (*rule Suc-neq-Zero*, *erule sym*)

lemma *Suc-inject*: *Suc x* = *Suc y* ⟹ *x* = *y*
by (*rule inj-Suc* [*THEN injD*])

lemma *n-not-Suc-n*: *n* ≠ *Suc n*
by (*induct n*) *simp-all*

lemma *Suc-n-not-n*: *Suc n* ≠ *n*
by (*rule not-sym*, *rule n-not-Suc-n*)

A special form of induction for reasoning about *m* < *n* and *m* − *n*

lemma *diff-induct*: (!!*x*. *P x 0*) ==> (!!*y*. *P 0* (*Suc y*)) ==>
    (!!*x y*. *P x y* ==> *P* (*Suc x*) (*Suc y*)) ==> *P m n*
  apply (*rule-tac x* = *m* **in** *spec*)
  apply (*induct n*)
  prefer *2*
  apply (*rule allI*)
  apply (*induct-tac x*, *iprover*+)
  done

12.3 Arithmetic operators

instantiation *nat* :: {*minus*, *comm-monoid-add*}
begin

primrec *plus-nat*
where
  add-0: *0* + *n* = (*n*::*nat*)
  | *add-Suc*: *Suc m* + *n* = *Suc* (*m* + *n*)

lemma *add-0-right* [*simp*]: *m* + *0* = (*m*::*nat*)
  by (*induct m*) *simp-all*

lemma *add-Suc-right* [*simp*]: $m + Suc\ n = Suc\ (m + n)$
by (*induct m*) *simp-all*

declare *add-0* [*code*]

lemma *add-Suc-shift* [*code*]: $Suc\ m + n = m + Suc\ n$
by *simp*

primrec *minus-nat*
where
diff-0: $m - 0 = (m::nat)$
| *diff-Suc*: $m - Suc\ n = (case\ m - n\ of\ 0 => 0 \mid Suc\ k => k)$

declare *diff-Suc* [*simp del*]
declare *diff-0* [*code*]

lemma *diff-0-eq-0* [*simp*, *code*]: $0 - n = (0::nat)$
by (*induct n*) (*simp-all add*: *diff-Suc*)

lemma *diff-Suc-Suc* [*simp*, *code*]: $Suc\ m - Suc\ n = m - n$
by (*induct n*) (*simp-all add*: *diff-Suc*)

instance proof
fix $n\ m\ q :: nat$
show $(n + m) + q = n + (m + q)$ **by** (*induct n*) *simp-all*
show $n + m = m + n$ **by** (*induct n*) *simp-all*
show $0 + n = n$ **by** *simp*
qed

end

instantiation *nat* :: *comm-semiring-1-cancel*
begin

definition
One-nat-def [*simp*]: $1 = Suc\ 0$

primrec *times-nat*
where
mult-0: $0 * n = (0::nat)$
| *mult-Suc*: $Suc\ m * n = n + (m * n)$

lemma *mult-0-right* [*simp*]: $(m::nat) * 0 = 0$
by (*induct m*) *simp-all*

lemma *mult-Suc-right* [*simp*]: $m * Suc\ n = m + (m * n)$
by (*induct m*) (*simp-all add*: *add-left-commute*)

lemma *add-mult-distrib*: $(m + n) * k = (m * k) + ((n * k)::nat)$

by (*induct m*) (*simp-all add*: *add-assoc*)

instance proof
fix *n m q* :: *nat*
show *0* ≠ (*1*::*nat*) **unfolding** *One-nat-def* **by** *simp*
show *1* ∗ *n* = *n* **unfolding** *One-nat-def* **by** *simp*
show *n* ∗ *m* = *m* ∗ *n* **by** (*induct n*) *simp-all*
show (*n* ∗ *m*) ∗ *q* = *n* ∗ (*m* ∗ *q*) **by** (*induct n*) (*simp-all add*: *add-mult-distrib*)
show (*n* + *m*) ∗ *q* = *n* ∗ *q* + *m* ∗ *q* **by** (*rule add-mult-distrib*)
assume *n* + *m* = *n* + *q* **thus** *m* = *q* **by** (*induct n*) *simp-all*
qed

end

12.3.1 Addition

lemma *nat-add-assoc*: (*m* + *n*) + *k* = *m* + ((*n* + *k*)::*nat*)
by (*rule add-assoc*)

lemma *nat-add-commute*: *m* + *n* = *n* + (*m*::*nat*)
by (*rule add-commute*)

lemma *nat-add-left-commute*: *x* + (*y* + *z*) = *y* + ((*x* + *z*)::*nat*)
by (*rule add-left-commute*)

lemma *nat-add-left-cancel* [*simp*]: (*k* + *m* = *k* + *n*) = (*m* = (*n*::*nat*))
by (*rule add-left-cancel*)

lemma *nat-add-right-cancel* [*simp*]: (*m* + *k* = *n* + *k*) = (*m*=(*n*::*nat*))
by (*rule add-right-cancel*)

Reasoning about *m* + *0* = *0*, etc.

lemma *add-is-0* [*iff*]:
fixes *m n* :: *nat*
shows (*m* + *n* = *0*) = (*m* = *0* & *n* = *0*)
by (*cases m*) *simp-all*

lemma *add-is-1*:
(*m*+*n*= *Suc 0*) = (*m*= *Suc 0* & *n*=*0* | *m*=*0* & *n*= *Suc 0*)
by (*cases m*) *simp-all*

lemma *one-is-add*:
(*Suc 0* = *m* + *n*) = (*m* = *Suc 0* & *n* = *0* | *m* = *0* & *n* = *Suc 0*)
by (*rule trans*, *rule eq-commute*, *rule add-is-1*)

lemma *add-eq-self-zero*:
fixes *m n* :: *nat*
shows *m* + *n* = *m* ⟹ *n* = *0*
by (*induct m*) *simp-all*

lemma *inj-on-add-nat*[*simp*]: *inj-on* (%*n*::*nat*. *n*+*k*) *N*
apply (*induct k*)
apply *simp*
apply(*drule comp-inj-on*[*OF - inj-Suc*])
apply (*simp add*:*o-def*)
done

12.3.2 Difference

lemma *diff-self-eq-0* [*simp*]: (*m*::*nat*) − *m* = *0*
by (*induct m*) *simp-all*

lemma *diff-diff-left*: (*i*::*nat*) − *j* − *k* = *i* − (*j* + *k*)
by (*induct i j rule*: *diff-induct*) *simp-all*

lemma *Suc-diff-diff* [*simp*]: (*Suc m* − *n*) − *Suc k* = *m* − *n* − *k*
by (*simp add*: *diff-diff-left*)

lemma *diff-commute*: (*i*::*nat*) − *j* − *k* = *i* − *k* − *j*
by (*simp add*: *diff-diff-left add-commute*)

lemma *diff-add-inverse*: (*n* + *m*) − *n* = (*m*::*nat*)
by (*induct n*) *simp-all*

lemma *diff-add-inverse2*: (*m* + *n*) − *n* = (*m*::*nat*)
by (*simp add*: *diff-add-inverse add-commute* [*of m n*])

lemma *diff-cancel*: (*k* + *m*) − (*k* + *n*) = *m* − (*n*::*nat*)
by (*induct k*) *simp-all*

lemma *diff-cancel2*: (*m* + *k*) − (*n* + *k*) = *m* − (*n*::*nat*)
by (*simp add*: *diff-cancel add-commute*)

lemma *diff-add-0*: *n* − (*n* + *m*) = (*0*::*nat*)
by (*induct n*) *simp-all*

lemma *diff-Suc-1* [*simp*]: *Suc n* − *1* = *n*
unfolding *One-nat-def* **by** *simp*

Difference distributes over multiplication

lemma *diff-mult-distrib*: ((*m*::*nat*) − *n*) ∗ *k* = (*m* ∗ *k*) − (*n* ∗ *k*)
by (*induct m n rule*: *diff-induct*) (*simp-all add*: *diff-cancel*)

lemma *diff-mult-distrib2*: *k* ∗ ((*m*::*nat*) − *n*) = (*k* ∗ *m*) − (*k* ∗ *n*)
by (*simp add*: *diff-mult-distrib mult-commute* [*of k*])
— NOT added as rewrites, since sometimes they are used from right-to-left

12.3.3 Multiplication

lemma *nat-mult-assoc*: $(m * n) * k = m * ((n * k)::nat)$
by (*rule mult-assoc*)

lemma *nat-mult-commute*: $m * n = n * (m::nat)$
by (*rule mult-commute*)

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + ((k * n)::nat)$
by (*rule right-distrib*)

lemma *mult-is-0* [*simp*]: $((m::nat) * n = 0) = (m=0 \mid n=0)$
by (*induct m*) *auto*

lemmas *nat-distrib* =
add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

lemma *mult-eq-1-iff* [*simp*]: $(m * n = Suc\ 0) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
apply (*induct m*)
apply *simp*
apply (*induct n*)
apply *auto*
done

lemma *one-eq-mult-iff* [*simp*,*noatp*]: $(Suc\ 0 = m * n) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
apply (*rule trans*)
apply (*rule-tac* [*2*] *mult-eq-1-iff*, *fastsimp*)
done

lemma *nat-mult-eq-1-iff* [*simp*]: $m * n = (1::nat) \longleftrightarrow m = 1 \wedge n = 1$
unfolding *One-nat-def* **by** (*rule mult-eq-1-iff*)

lemma *nat-1-eq-mult-iff* [*simp*]: $(1::nat) = m * n \longleftrightarrow m = 1 \wedge n = 1$
unfolding *One-nat-def* **by** (*rule one-eq-mult-iff*)

lemma *mult-cancel1* [*simp*]: $(k * m = k * n) = (m = n \mid (k = (0::nat)))$
proof −
have $k \neq 0 \Longrightarrow k * m = k * n \Longrightarrow m = n$
proof (*induct n arbitrary*: *m*)
case *0* **then show** $m = 0$ **by** *simp*
next
case (*Suc n*) **then show** $m = Suc\ n$
by (*cases m*) (*simp-all add*: *eq-commute* [*of 0*])
qed
then show *?thesis* **by** *auto*
qed

lemma *mult-cancel2* [*simp*]: $(m * k = n * k) = (m = n \mid (k = (0::nat)))$
by (*simp add*: *mult-commute*)

lemma *Suc-mult-cancel1*: $(Suc\ k * m = Suc\ k * n) = (m = n)$
by (*subst mult-cancel1*) *simp*

12.4 Orders on *nat*

12.4.1 Operation definition

instantiation *nat* :: *linorder*
begin

primrec *less-eq-nat* **where**
$(0::nat) \leq n \longleftrightarrow True$
| $Suc\ m \leq n \longleftrightarrow (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*]
lemma [*code*]: $(0::nat) \leq n \longleftrightarrow True$ **by** (*simp add*: *less-eq-nat.simps*)
lemma *le0* [*iff*]: $0 \leq (n::nat)$ **by** (*simp add*: *less-eq-nat.simps*)

definition *less-nat* **where**
less-eq-Suc-le: $n < m \longleftrightarrow Suc\ n \leq m$

lemma *Suc-le-mono* [*iff*]: $Suc\ n \leq Suc\ m \longleftrightarrow n \leq m$
by (*simp add*: *less-eq-nat.simps*(*2*))

lemma *Suc-le-eq* [*code*]: $Suc\ m \leq n \longleftrightarrow m < n$
unfolding *less-eq-Suc-le* **..**

lemma *le-0-eq* [*iff*]: $(n::nat) \leq 0 \longleftrightarrow n = 0$
by (*induct n*) (*simp-all add*: *less-eq-nat.simps*(*2*))

lemma *not-less0* [*iff*]: $\neg\ n < (0::nat)$
by (*simp add*: *less-eq-Suc-le*)

lemma *less-nat-zero-code* [*code*]: $n < (0::nat) \longleftrightarrow False$
by *simp*

lemma *Suc-less-eq* [*iff*]: $Suc\ m < Suc\ n \longleftrightarrow m < n$
by (*simp add*: *less-eq-Suc-le*)

lemma *less-Suc-eq-le* [*code*]: $m < Suc\ n \longleftrightarrow m \leq n$
by (*simp add*: *less-eq-Suc-le*)

lemma *le-SucI*: $m \leq n \Longrightarrow m \leq Suc\ n$
by (*induct m arbitrary*: *n*)
(*simp-all add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)

lemma *Suc-leD*: $Suc\ m \leq n \Longrightarrow m \leq n$
by (*cases n*) (*auto intro*: *le-SucI*)

lemma *less-SucI*: $m < n \Longrightarrow m < Suc\ n$
by (*simp add*: *less-eq-Suc-le*) (*erule Suc-leD*)

lemma *Suc-lessD*: $Suc\ m < n \Longrightarrow m < n$
by (*simp add*: *less-eq-Suc-le*) (*erule Suc-leD*)

instance
proof
fix $n\ m$ :: *nat*
show $n < m \longleftrightarrow n \leq m \wedge \neg\ m \leq n$
proof (*induct n arbitrary*: *m*)
case *0* **then show** *?case* **by** (*cases m*) (*simp-all add*: *less-eq-Suc-le*)
next
case (*Suc n*) **then show** *?case* **by** (*cases m*) (*simp-all add*: *less-eq-Suc-le*)
qed
next
fix *n* :: *nat* **show** $n \leq n$ **by** (*induct n*) *simp-all*
next
fix $n\ m$ :: *nat* **assume** $n \leq m$ **and** $m \leq n$
then show $n = m$
by (*induct n arbitrary*: *m*)
(*simp-all add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)
next
fix $n\ m\ q$ :: *nat* **assume** $n \leq m$ **and** $m \leq q$
then show $n \leq q$
proof (*induct n arbitrary*: *m q*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **then show** *?case*
by (*simp-all* (*no-asm-use*) *add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*, *clarify*,
simp-all (*no-asm-use*) *add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*, *clarify*,
simp-all (*no-asm-use*) *add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)
qed
next
fix $n\ m$ :: *nat* **show** $n \leq m \vee m \leq n$
by (*induct n arbitrary*: *m*)
(*simp-all add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)
qed

end

instantiation *nat* :: *bot*
begin

definition *bot-nat* :: *nat* **where**
bot-nat = *0*

instance proof
qed (*simp add*: *bot-nat-def*)

end

12.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < Suc\ n$
by (*simp add*: *less-Suc-eq-le*)

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
by (*simp add*: *less-Suc-eq-le*)

12.4.3 Elimination properties

lemma *less-not-refl*: $\sim n < (n::nat)$
by (*rule order-less-irrefl*)

lemma *less-not-refl2*: $n < m \Longrightarrow m \neq (n::nat)$
by (*rule not-sym*) (*rule less-imp-neq*)

lemma *less-not-refl3*: $(s::nat) < t \Longrightarrow s \neq t$
by (*rule less-imp-neq*)

lemma *less-irrefl-nat*: $(n::nat) < n \Longrightarrow R$
by (*rule notE*, *rule less-not-refl*)

lemma *less-zeroE*: $(n::nat) < 0 \Longrightarrow R$
by (*rule notE*) (*rule not-less0*)

lemma *less-Suc-eq*: $(m < Suc\ n) = (m < n \mid m = n)$
unfolding *less-Suc-eq-le le-less* **..**

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
by (*simp add*: *less-Suc-eq*)

lemma *less-one* [*iff*, *noatp*]: $(n < (1::nat)) = (n = 0)$
unfolding *One-nat-def* **by** (*rule less-Suc0*)

lemma *Suc-mono*: $m < n \Longrightarrow Suc\ m < Suc\ n$
by *simp*

"Less than" is antisymmetric, sort of

lemma *less-antisym*: $[\![\neg n < m;\ n < Suc\ m]\!] \Longrightarrow m = n$
unfolding *not-less less-Suc-eq-le* **by** (*rule antisym*)

lemma *nat-neq-iff*: $((m::nat) \neq n) = (m < n \mid n < m)$
by (*rule linorder-neq-iff*)

lemma *nat-less-cases*: **assumes** *major*: $(m::nat) < n \Longrightarrow P\ n\ m$
and *eqCase*: $m = n \Longrightarrow P\ n\ m$ **and** *lessCase*: $n<m \Longrightarrow P\ n\ m$
shows $P\ n\ m$

apply (*rule less-linear* [*THEN disjE*])
apply (*erule-tac* [*2*] *disjE*)
apply (*erule lessCase*)
apply (*erule sym* [*THEN eqCase*])
apply (*erule major*)
done

12.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \Longrightarrow Suc\ m \neq n \Longrightarrow Suc\ m < n$
unfolding *less-eq-Suc-le* [*of m*] *le-less* **by** *simp*

lemma *lessE*:
assumes *major*: $i < k$
and *p1*: $k = Suc\ i \Longrightarrow P$ **and** *p2*: $!!j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$
shows P
proof −
from *major* **have** $\exists j.\ i \leq j \wedge k = Suc\ j$
unfolding *less-eq-Suc-le* **by** (*induct k*) *simp-all*
then have $(\exists j.\ i < j \wedge k = Suc\ j) \vee k = Suc\ i$
by (*clarsimp simp add*: *less-le*)
with *p1 p2* **show** P **by** *auto*
qed

lemma *less-SucE*: **assumes** *major*: $m < Suc\ n$
and *less*: $m < n \Longrightarrow P$ **and** *eq*: $m = n \Longrightarrow P$ **shows** P
apply (*rule major* [*THEN lessE*])
apply (*rule eq*, *blast*)
apply (*rule less*, *blast*)
done

lemma *Suc-lessE*: **assumes** *major*: $Suc\ i < k$
and *minor*: $!!j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$ **shows** P
apply (*rule major* [*THEN lessE*])
apply (*erule lessI* [*THEN minor*])
apply (*erule Suc-lessD* [*THEN minor*], *assumption*)
done

lemma *Suc-less-SucD*: $Suc\ m < Suc\ n \Longrightarrow m < n$
by *simp*

lemma *less-trans-Suc*:
assumes *le*: $i < j$ **shows** $j < k \Longrightarrow Suc\ i < k$
apply (*induct k*, *simp-all*)
apply (*insert le*)
apply (*simp add*: *less-Suc-eq*)
apply (*blast dest*: *Suc-lessD*)
done

Can be used with *less-Suc-eq* to get $n = m \vee n < m$

lemma *not-less-eq*: $\neg\ m < n \longleftrightarrow n < Suc\ m$
unfolding *not-less less-Suc-eq-le* **..**

lemma *not-less-eq-eq*: $\neg\ m \leq n \longleftrightarrow Suc\ n \leq m$
unfolding *not-le Suc-le-eq* **..**

Properties of "less than or equal"

lemma *le-imp-less-Suc*: $m \leq n \Longrightarrow m < Suc\ n$
unfolding *less-Suc-eq-le* **.**

lemma *Suc-n-not-le-n*: $\sim\ Suc\ n \leq n$
unfolding *not-le less-Suc-eq-le* **..**

lemma *le-Suc-eq*: $(m \leq Suc\ n) = (m \leq n \mid m = Suc\ n)$
by (*simp add*: *less-Suc-eq-le* [*symmetric*] *less-Suc-eq*)

lemma *le-SucE*: $m \leq Suc\ n \Longrightarrow (m \leq n \Longrightarrow R) \Longrightarrow (m = Suc\ n \Longrightarrow R) \Longrightarrow R$
by (*drule le-Suc-eq* [*THEN iffD1*], *iprover*+)

lemma *Suc-leI*: $m < n \Longrightarrow Suc(m) \leq n$
unfolding *Suc-le-eq* **.**

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: $Suc\ m \leq n \Longrightarrow m < n$
unfolding *Suc-le-eq* **.**

lemma *less-imp-le-nat*: $m < n \Longrightarrow m \leq (n::nat)$
unfolding *less-eq-Suc-le* **by** (*rule Suc-leD*)

For instance, $(Suc\ m < Suc\ n) = (Suc\ m \leq n) = (m < n)$

lemmas *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *less-or-eq-imp-le*: $m < n \mid m = n \Longrightarrow m \leq (n::nat)$
unfolding *le-less* **.**

lemma *le-eq-less-or-eq*: $(m \leq (n::nat)) = (m < n \mid m=n)$
by (*rule le-less*)

Useful with *blast*.

lemma *eq-imp-le*: $(m::nat) = n \Longrightarrow m \leq n$
by *auto*

lemma *le-refl*: $n \leq (n::nat)$
by *simp*

lemma *le-trans*: $[|\ i \leq j; j \leq k\ |] \Longrightarrow i \leq (k::nat)$

by (*rule order-trans*)

lemma *le-anti-sym*: [| $m \leq n$; $n \leq m$ |] ==> $m = (n::nat)$
by (*rule antisym*)

lemma *nat-less-le*: $((m::nat) < n) = (m \leq n \;\&\; m \neq n)$
by (*rule less-le*)

lemma *le-neq-implies-less*: $(m::nat) \leq n$ ==> $m \neq n$ ==> $m < n$
unfolding *less-le* **..**

lemma *nat-le-linear*: $(m::nat) \leq n \mid n \leq m$
by (*rule linear*)

lemmas *linorder-neqE-nat* = *linorder-neqE* [**where** $'a = nat$]

lemma *le-less-Suc-eq*: $m \leq n$ ==> $(n < Suc\ m) = (n = m)$
unfolding *less-Suc-eq-le* **by** *auto*

lemma *not-less-less-Suc-eq*: ~ $n < m$ ==> $(n < Suc\ m) = (n = m)$
unfolding *not-less* **by** (*rule le-less-Suc-eq*)

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

These two rules ease the use of primitive recursion. NOTE USE OF ==

lemma *def-nat-rec-0*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n)$ ==> $f\ 0 = c$
by *simp*

lemma *def-nat-rec-Suc*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n)$ ==> $f\ (Suc\ n) = h\ n\ (f\ n)$
by *simp*

lemma *not0-implies-Suc*: $n \neq 0$ ==> $\exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr0-implies-Suc*: $n > 0$ ==> $\exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr-implies-not0*: **fixes** $n :: nat$ **shows** $m<n$ ==> $n \neq 0$
by (*cases n*) *simp-all*

lemma *neq0-conv*[*iff*]: **fixes** $n :: nat$ **shows** $(n \neq 0) = (0 < n)$
by (*cases n*) *simp-all*

This theorem is useful with *blast*

lemma *gr0I*: $((n::nat) = 0$ ==> $False)$ ==> $0 < n$
by (*rule neq0-conv*[*THEN iffD1*], *iprover*)

lemma *gr0-conv-Suc*: $(0 < n) = (\exists\, m.\ n = Suc\ m)$
by (*fast intro*: *not0-implies-Suc*)

lemma *not-gr0* [*iff*,*noatp*]: !!*n*::*nat*. (~ (*0* < *n*)) = (*n* = *0*)
using *neq0-conv* **by** *blast*

lemma *Suc-le-D*: (*Suc n* ≤ *m'*) ==> (*? m*. *m'* = *Suc m*)
by (*induct m'*) *simp-all*

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: (*m* < *Suc n*) = (*m* = *0* | (∃ *j*. *m* = *Suc j* & *j* < *n*))
by (*cases m*) *simp-all*

12.4.5 *min* and *max*

lemma *mono-Suc*: *mono Suc*
by (*rule monoI*) *simp*

lemma *min-0L* [*simp*]: *min 0 n* = (*0*::*nat*)
by (*rule min-leastL*) *simp*

lemma *min-0R* [*simp*]: *min n 0* = (*0*::*nat*)
by (*rule min-leastR*) *simp*

lemma *min-Suc-Suc* [*simp*]: *min* (*Suc m*) (*Suc n*) = *Suc* (*min m n*)
by (*simp add*: *mono-Suc min-of-mono*)

lemma *min-Suc1*:
min (*Suc n*) *m* = (*case m of 0* => *0* | *Suc m'* => *Suc*(*min n m'*))
by (*simp split*: *nat.split*)

lemma *min-Suc2*:
min m (*Suc n*) = (*case m of 0* => *0* | *Suc m'* => *Suc*(*min m' n*))
by (*simp split*: *nat.split*)

lemma *max-0L* [*simp*]: *max 0 n* = (*n*::*nat*)
by (*rule max-leastL*) *simp*

lemma *max-0R* [*simp*]: *max n 0* = (*n*::*nat*)
by (*rule max-leastR*) *simp*

lemma *max-Suc-Suc* [*simp*]: *max* (*Suc m*) (*Suc n*) = *Suc*(*max m n*)
by (*simp add*: *mono-Suc max-of-mono*)

lemma *max-Suc1*:
max (*Suc n*) *m* = (*case m of 0* => *Suc n* | *Suc m'* => *Suc*(*max n m'*))
by (*simp split*: *nat.split*)

lemma *max-Suc2*:
max m (*Suc n*) = (*case m of 0* => *Suc n* | *Suc m'* => *Suc*(*max m' n*))
by (*simp split*: *nat.split*)

12.4.6 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: *n>0 ==> Suc (n − Suc 0) = n*
by (*simp add*: *diff-Suc split*: *nat.split*)

lemma *Suc-diff-1* [*simp*]: *0 < n ==> Suc (n − 1) = n*
unfolding *One-nat-def* **by** (*rule Suc-pred*)

lemma *nat-add-left-cancel-le* [*simp*]: *(k + m ≤ k + n) = (m≤(n::nat))*
by (*induct k*) *simp-all*

lemma *nat-add-left-cancel-less* [*simp*]: *(k + m < k + n) = (m<(n::nat))*
by (*induct k*) *simp-all*

lemma *add-gr-0* [*iff*]: *!!m::nat. (m + n > 0) = (m>0 | n>0)*
by(*auto dest*:*gr0-implies-Suc*)

strict, in 1st argument

lemma *add-less-mono1*: *i < j ==> i + k < j + (k::nat)*
by (*induct k*) *simp-all*

strict, in both arguments

lemma *add-less-mono*: *[|i < j; k < l|] ==> i + k < j + (l::nat)*
  apply (*rule add-less-mono1* [*THEN less-trans*], *assumption*+)
  apply (*induct j*, *simp-all*)
  done

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

lemma *less-imp-Suc-add*: *m < n ==> (∃ k. n = Suc (m + k))*
  apply (*induct n*)
  apply (*simp-all add*: *order-le-less*)
  apply (*blast elim*!: *less-SucE*
        intro!: *add-0-right* [*symmetric*] *add-Suc-right* [*symmetric*])
  done

strict, in 1st argument; proof is by induction on *k > 0*

lemma *mult-less-mono2*: *(i::nat) < j ==> 0<k ==> k * i < k * j*
apply(*auto simp*: *gr0-conv-Suc*)
apply (*induct-tac m*)
apply (*simp-all add*: *add-less-mono*)
done

The naturals form an ordered *comm-semiring-1-cancel*

instance *nat* :: *ordered-semidom*
proof
  fix *i j k* :: *nat*
  show *0 < (1::nat)* **by** *simp*
  show *i ≤ j ==> k + i ≤ k + j* **by** *simp*
  show *i < j ==> 0 < k ==> k * i < k * j* **by** (*simp add*: *mult-less-mono2*)

qed

instance *nat* :: *no-zero-divisors*
proof
fix *a*::*nat* **and** *b*::*nat* **show** $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$ **by** *auto*
qed

lemma *nat-mult-1*: $(1::nat) * n = n$
by *simp*

lemma *nat-mult-1-right*: $n * (1::nat) = n$
by *simp*

12.4.7 Additional theorems about *op* ≤

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder* **proof**
fix *P* **and** *n* :: *nat*
assume *step*: $\bigwedge n::nat.\ (\bigwedge m.\ m < n \Longrightarrow P\ m) \Longrightarrow P\ n$
have $\bigwedge q.\ q \leq n \Longrightarrow P\ q$
proof (*induct n*)
case (*0 n*)
have *P 0* **by** (*rule step*) *auto*
thus *?case* **using** *0* **by** *auto*
next
case (*Suc m n*)
then have $n \leq m \vee n = Suc\ m$ **by** (*simp add*: *le-Suc-eq*)
thus *?case*
proof
assume $n \leq m$ **thus** *P n* **by** (*rule Suc*(*1*))
next
assume *n*: $n = Suc\ m$
show *P n*
by (*rule step*) (*rule Suc*(*1*), *simp add*: *n le-simps*)
qed
qed
then show *P n* **by** *auto*
qed

lemma *Least-Suc*:
[| *P n*; ~ *P 0* |] ==> (*LEAST n. P n*) = *Suc* (*LEAST m. P*(*Suc m*))
apply (*case-tac n*, *auto*)
apply (*frule LeastI*)
apply (*drule-tac P* = %*x. P* (*Suc x*) **in** *LeastI*)
apply (*subgoal-tac* (*LEAST x. P x*) ≤ *Suc* (*LEAST x. P* (*Suc x*)))
apply (*erule-tac* [*2*] *Least-le*)
apply (*case-tac LEAST x. P x*, *auto*)
apply (*drule-tac P* = %*x. P* (*Suc x*) **in** *Least-le*)
apply (*blast intro*: *order-antisym*)

done

lemma *Least-Suc2*:
[|*P n*; *Q m*; ~*P 0*; !*k*. *P* (*Suc k*) = *Q k*|] ==> *Least P* = *Suc* (*Least Q*)
apply (*erule* (*1*) *Least-Suc* [*THEN ssubst*])
apply *simp*
done

lemma *ex-least-nat-le*: ¬*P*(*0*) ⟹ *P*(*n*::*nat*) ⟹ ∃*k*≤*n*. (∀*i*<*k*. ¬*P i*) & *P*(*k*)
apply (*cases n*)
apply *blast*
apply (*rule-tac x*=*LEAST k*. *P*(*k*) **in** *exI*)
apply (*blast intro*: *Least-le dest*: *not-less-Least intro*: *LeastI-ex*)
done

lemma *ex-least-nat-less*: ¬*P*(*0*) ⟹ *P*(*n*::*nat*) ⟹ ∃*k*<*n*. (∀*i*≤*k*. ¬*P i*) & *P*(*k*+*1*)
unfolding *One-nat-def*
apply (*cases n*)
apply *blast*
apply (*frule* (*1*) *ex-least-nat-le*)
apply (*erule exE*)
apply (*case-tac k*)
apply *simp*
apply (*rename-tac k1*)
apply (*rule-tac x*=*k1* **in** *exI*)
apply (*auto simp add*: *less-eq-Suc-le*)
done

lemma *nat-less-induct*:
assumes !!*n*. ∀*m*::*nat*. *m* < *n* --> *P m* ==> *P n* **shows** *P n*
using *assms less-induct* **by** *blast*

lemma *measure-induct-rule* [*case-names less*]:
fixes *f* :: *'a* ⇒ *nat*
assumes *step*: ⋀*x*. (⋀*y*. *f y* < *f x* ⟹ *P y*) ⟹ *P x*
shows *P a*
by (*induct m*≡*f a arbitrary*: *a rule*: *less-induct*) (*auto intro*: *step*)

old style induction rules:

lemma *measure-induct*:
fixes *f* :: *'a* ⇒ *nat*
shows (⋀*x*. ∀*y*. *f y* < *f x* ⟶ *P y* ⟹ *P x*) ⟹ *P a*
by (*rule measure-induct-rule* [*of f P a*]) *iprover*

lemma *full-nat-induct*:
assumes *step*: (!!*n*. (*ALL m*. *Suc m* <= *n* --> *P m*) ==> *P n*)
shows *P n*
by (*rule less-induct*) (*auto intro*: *step simp*:*le-simps*)

An induction rule for estabilishing binary relations

lemma *less-Suc-induct*:
assumes *less*: $i < j$
and *step*: $!!i.\ P\ i\ (Suc\ i)$
and *trans*: $!!i\ j\ k.\ P\ i\ j \Longrightarrow P\ j\ k \Longrightarrow P\ i\ k$
shows $P\ i\ j$
proof −
from *less* **obtain** k **where** j: $j = Suc(i+k)$ **by** (*auto dest*: *less-imp-Suc-add*)
have $P\ i\ (Suc\ (i + k))$
proof (*induct k*)
case *0*
show *?case* **by** (*simp add*: *step*)
next
case (*Suc k*)
thus *?case* **by** (*auto intro*: *assms*)
qed
thus $P\ i\ j$ **by** (*simp add*: *j*)
qed

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. $P(n)$ is true for all $n \in \mathbb{N}$ if

- case "0": given $n = 0$ prove $P(n)$,
- case "smaller": given $n > 0$ and $\neg P(n)$ prove there exists a smaller integer m such that $\neg P(m)$.

A compact version without explicit base case:

lemma *infinite-descent*:
$[\![\ !!n::nat.\ \neg\ P\ n \Longrightarrow\ \exists\, m<n.\ \neg\ \ P\ m\ ]\!] \Longrightarrow\ \ P\ n$
by (*induct n rule*: *less-induct*, *auto*)

lemma *infinite-descent0*[*case-names 0 smaller*]:
$[\![\ P\ 0;\ !!n.\ n>0 \Longrightarrow \neg\ P\ n \Longrightarrow (\exists\, m::nat.\ m < n \wedge \neg P\ m)\ ]\!] \Longrightarrow P\ n$
by (*rule infinite-descent*) (*case-tac n>0*, *auto*)

Infinite descent using a mapping to $\mathbb{N}$: $P(x)$ is true for all $x \in D$ if there exists a $V : D \to \mathbb{N}$ and

- case "0": given $V(x) = 0$ prove $P(x)$,
- case "smaller": given $V(x) > 0$ and $\neg P(x)$ prove there exists a $y \in D$ such that $V(y) < V(x)$ and $\neg P(y)$.

NB: the proof also shows how to use the previous lemma.

corollary *infinite-descent0-measure* [*case-names 0 smaller*]:
assumes *A0*: $!!x.\ V\ x = (0::nat) \Longrightarrow P\ x$
and *A1*: $!!x.\ V\ x > 0 \Longrightarrow \neg P\ x \Longrightarrow (\exists\, y.\ V\ y < V\ x \wedge \neg P\ y)$
shows $P\ x$

proof −
obtain n **where** $n = V\ x$ **by** *auto*
moreover have $\bigwedge x.\ V\ x = n \Longrightarrow P\ x$
proof (*induct n rule*: *infinite-descent0*)
case *0* — i.e. $V(x) = 0$
with *A0* **show** $P\ x$ **by** *auto*
next — now $n > 0$ and $P(x)$ does not hold for some x with $V(x) = n$
case (*smaller n*)
then obtain x **where** *vxn*: $V\ x = n$ **and** $V\ x > 0 \wedge \neg\ P\ x$ **by** *auto*
with *A1* **obtain** y **where** $V\ y < V\ x \wedge \neg\ P\ y$ **by** *auto*
with *vxn* **obtain** m **where** $m = V\ y \wedge m<n \wedge \neg\ P\ y$ **by** *auto*
then show *?case* **by** *auto*
qed
ultimately show $P\ x$ **by** *auto*
qed

Again, without explicit base case:

lemma *infinite-descent-measure*:
assumes $!!x.\ \neg\ P\ x \Longrightarrow \exists\, y.\ (V::'a\Rightarrow nat)\ y < V\ x \wedge \neg\ P\ y$ **shows** $P\ x$
proof −
from *assms* **obtain** n **where** $n = V\ x$ **by** *auto*
moreover have $!!x.\ V\ x = n \Longrightarrow P\ x$
proof (*induct n rule*: *infinite-descent*, *auto*)
fix x **assume** $\neg\ P\ x$
with *assms* **show** $\exists\, m < V\ x.\ \exists\, y.\ V\ y = m \wedge \neg\ P\ y$ **by** *auto*
qed
ultimately show $P\ x$ **by** *auto*
qed

A [clumsy] way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma *less-mono-imp-le-mono*:
$[\![\ !!i\ j::nat.\ i < j \Longrightarrow f\ i < f\ j;\ i \leq j\ ]\!] \Longrightarrow f\ i \leq ((f\ j)::nat)$
by (*simp add*: *order-le-less*) (*blast*)

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j$ ==> $i + k \leq j + (k::nat)$
by (*rule add-right-mono*)

non-strict, in both arguments

lemma *add-le-mono*: [| $i \leq j$; $k \leq l$ |] ==> $i + k \leq j + (l::nat)$
by (*rule add-mono*)

lemma *le-add2*: $n \leq ((m + n)::nat)$
by (*insert add-right-mono* [*of 0 m n*], *simp*)

lemma *le-add1*: $n \leq ((n + m)::nat)$
by (*simp add*: *add-commute*, *rule le-add2*)

lemma *less-add-Suc1*: $i < Suc\ (i + m)$
by (*rule le-less-trans*, *rule le-add1*, *rule lessI*)

lemma *less-add-Suc2*: $i < Suc\ (m + i)$
by (*rule le-less-trans*, *rule le-add2*, *rule lessI*)

lemma *less-iff-Suc-add*: $(m < n) = (\exists k.\ n = Suc\ (m + k))$
by (*iprover intro*!: *less-add-Suc1 less-imp-Suc-add*)

lemma *trans-le-add1*: $(i::nat) \leq j \Longrightarrow i \leq j + m$
by (*rule le-trans*, *assumption*, *rule le-add1*)

lemma *trans-le-add2*: $(i::nat) \leq j \Longrightarrow i \leq m + j$
by (*rule le-trans*, *assumption*, *rule le-add2*)

lemma *trans-less-add1*: $(i::nat) < j \Longrightarrow i < j + m$
by (*rule less-le-trans*, *assumption*, *rule le-add1*)

lemma *trans-less-add2*: $(i::nat) < j \Longrightarrow i < m + j$
by (*rule less-le-trans*, *assumption*, *rule le-add2*)

lemma *add-lessD1*: $i + j < (k::nat) \Longrightarrow i < k$
apply (*rule le-less-trans* [*of* - *i+j*])
apply (*simp-all add*: *le-add1*)
done

lemma *not-add-less1* [*iff*]: $\sim (i + j < (i::nat))$
apply (*rule notI*)
apply (*drule add-lessD1*)
apply (*erule less-irrefl* [*THEN notE*])
done

lemma *not-add-less2* [*iff*]: $\sim (j + i < (i::nat))$
by (*simp add*: *add-commute*)

lemma *add-leD1*: $m + k \leq n \Longrightarrow m \leq (n::nat)$
apply (*rule order-trans* [*of* - *m+k*])
apply (*simp-all add*: *le-add1*)
done

lemma *add-leD2*: $m + k \leq n \Longrightarrow k \leq (n::nat)$
apply (*simp add*: *add-commute*)
apply (*erule add-leD1*)
done

lemma *add-leE*: $(m::nat) + k \leq n \Longrightarrow (m \leq n \Longrightarrow k \leq n \Longrightarrow R) \Longrightarrow R$
by (*blast dest*: *add-leD1 add-leD2*)

needs !!k for *add-ac* to work

lemma *less-add-eq-less*: !!k::*nat*. $k < l$ ==> $m + l = k + n$ ==> $m < n$
by (*force simp del*: *add-Suc-right*
simp add: *less-iff-Suc-add add-Suc-right* [*symmetric*] *add-ac*)

12.4.8 More results about difference

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add-diff-inverse*: ~ $m < n$ ==> $n + (m - n) = (m::nat)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *le-add-diff-inverse* [*simp*]: $n \leq m$ ==> $n + (m - n) = (m::nat)$
by (*simp add*: *add-diff-inverse linorder-not-less*)

lemma *le-add-diff-inverse2* [*simp*]: $n \leq m$ ==> $(m - n) + n = (m::nat)$
by (*simp add*: *add-commute*)

lemma *Suc-diff-le*: $n \leq m$ ==> *Suc* $m - n$ = *Suc* $(m - n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-less-Suc*: $m - n <$ *Suc* m
apply (*induct m n rule*: *diff-induct*)
apply (*erule-tac* [*3*] *less-SucE*)
apply (*simp-all add*: *less-Suc-eq*)
done

lemma *diff-le-self* [*simp*]: $m - n \leq (m::nat)$
by (*induct m n rule*: *diff-induct*) (*simp-all add*: *le-SucI*)

lemma *le-iff-add*: $(m::nat) \leq n = (\exists k.\ n = m + k)$
by (*auto simp*: *le-add1 dest*!: *le-add-diff-inverse sym* [*of* - *n*])

lemma *less-imp-diff-less*: $(j::nat) < k$ ==> $j - n < k$
by (*rule le-less-trans*, *rule diff-le-self*)

lemma *diff-Suc-less* [*simp*]: *0*<n ==> $n -$ *Suc* $i < n$
by (*cases n*) (*auto simp add*: *le-simps*)

lemma *diff-add-assoc*: $k \leq (j::nat)$ ==> $(i + j) - k = i + (j - k)$
by (*induct j k rule*: *diff-induct*) *simp-all*

lemma *diff-add-assoc2*: $k \leq (j::nat)$ ==> $(j + i) - k = (j - k) + i$
by (*simp add*: *add-commute diff-add-assoc*)

lemma *le-imp-diff-is-add*: $i \leq (j::nat)$ ==> $(j - i = k) = (j = k + i)$
by (*auto simp add*: *diff-add-inverse2*)

lemma *diff-is-0-eq* [*simp*]: $((m::nat) - n = 0) = (m \leq n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-is-0-eq'* [*simp*]: $m \leq n \Longrightarrow (m::nat) - n = 0$
by (*rule iffD2*, *rule diff-is-0-eq*)

lemma *zero-less-diff* [*simp*]: $(0 < n - (m::nat)) = (m < n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *less-imp-add-positive*:
assumes $i < j$
shows $\exists k::nat.\ 0 < k\ \&\ i + k = j$
proof
from *assms* **show** $0 < j - i\ \&\ i + (j - i) = j$
by (*simp add*: *order-less-imp-le*)
qed

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*:
fixes *n m* :: *nat*
shows $n - m + m = max\ n\ m$
by (*simp add*: *max-def not-le order-less-imp-le*)

lemma *nat-diff-split*:
$P(a - b::nat) = ((a<b \longrightarrow P\ 0)\ \&\ (ALL\ d.\ a = b + d \longrightarrow P\ d))$
— elimination of $-$ on *nat*
by (*cases* $a < b$)
(*auto simp add*: *diff-is-0-eq* [*THEN iffD2*] *diff-add-inverse*
not-less le-less dest!: *sym* [*of a*] *sym* [*of b*] *add-eq-self-zero*)

lemma *nat-diff-split-asm*:
$P(a - b::nat) = (\sim\ (a < b\ \&\ \sim\ P\ 0 \mid (EX\ d.\ a = b + d\ \&\ \sim\ P\ d)))$
— elimination of $-$ on *nat* in assumptions
by (*auto split*: *nat-diff-split*)

12.4.9 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \leq (j::nat) \Longrightarrow i * k \leq j * k$
by (*simp add*: *mult-right-mono*)

lemma *mult-le-mono2*: $i \leq (j::nat) \Longrightarrow k * i \leq k * j$
by (*simp add*: *mult-left-mono*)

$\leq$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq (j::nat) \Longrightarrow k \leq l \Longrightarrow i * k \leq j * l$
by (*simp add*: *mult-mono*)

lemma *mult-less-mono1*: $(i::nat) < j \Longrightarrow 0 < k \Longrightarrow i * k < j * k$
by (*simp add*: *mult-strict-right-mono*)

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $(0 < (m::nat) * n) = (0 < m \ \& \ 0 < n)$
apply (*induct m*)
apply *simp*
apply (*case-tac n*)
apply *simp-all*
done

lemma *one-le-mult-iff* [*simp*]: $(Suc\ 0 \leq m * n) = (Suc\ 0 \leq m \ \& \ Suc\ 0 \leq n)$
apply (*induct m*)
apply *simp*
apply (*case-tac n*)
apply *simp-all*
done

lemma *mult-less-cancel2* [*simp*]: $((m::nat) * k < n * k) = (0 < k \ \& \ m < n)$
apply (*safe intro*!: *mult-less-mono1*)
apply (*case-tac k*, *auto*)
apply (*simp del*: *le-0-eq add*: *linorder-not-le* [*symmetric*])
apply (*blast intro*: *mult-le-mono1*)
done

lemma *mult-less-cancel1* [*simp*]: $(k * (m::nat) < k * n) = (0 < k \ \& \ m < n)$
by (*simp add*: *mult-commute* [*of k*])

lemma *mult-le-cancel1* [*simp*]: $(k * (m::nat) \leq k * n) = (0 < k \longrightarrow m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *mult-le-cancel2* [*simp*]: $((m::nat) * k \leq n * k) = (0 < k \longrightarrow m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *Suc-mult-less-cancel1*: $(Suc\ k * m < Suc\ k * n) = (m < n)$
by (*subst mult-less-cancel1*) *simp*

lemma *Suc-mult-le-cancel1*: $(Suc\ k * m \leq Suc\ k * n) = (m \leq n)$
by (*subst mult-le-cancel1*) *simp*

lemma *le-square*: $m \leq m * (m::nat)$
by (*cases m*) (*auto intro*: *le-add1*)

lemma *le-cube*: $(m::nat) \leq m * (m * m)$
by (*cases m*) (*auto intro*: *le-add1*)

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $(m::nat) = m * n \Longrightarrow n = 1 \mid m = 0$
apply (*drule sym*)
apply (*rule disjCI*)
apply (*rule nat-less-cases*, *erule-tac* [*2*] -)
apply (*drule-tac* [*2*] *mult-less-mono2*)
apply (*auto*)

done

the lattice order on *nat*

instantiation *nat* :: *distrib-lattice*
begin

definition
(*inf* :: *nat* ⇒ *nat* ⇒ *nat*) = *min*

definition
(*sup* :: *nat* ⇒ *nat* ⇒ *nat*) = *max*

instance by *intro-classes*
(*auto simp add*: *inf-nat-def sup-nat-def max-def not-le min-def*
intro: *order-less-imp-le antisym elim*!: *order-trans order-less-trans*)

end

12.5 Embedding of the Naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

primrec
of-nat :: *nat* ⇒ *'a*
where
of-nat-0: *of-nat 0 = 0*
| *of-nat-Suc*: *of-nat* (*Suc m*) = *1* + *of-nat m*

lemma *of-nat-1* [*simp*]: *of-nat 1 = 1*
unfolding *One-nat-def* **by** *simp*

lemma *of-nat-add* [*simp*]: *of-nat* (*m* + *n*) = *of-nat m* + *of-nat n*
by (*induct m*) (*simp-all add*: *add-ac*)

lemma *of-nat-mult*: *of-nat* (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
by (*induct m*) (*simp-all add*: *add-ac left-distrib*)

primrec *of-nat-aux* :: (*'a* ⇒ *'a*) ⇒ *nat* ⇒ *'a* ⇒ *'a* **where**
of-nat-aux inc 0 i = i
| *of-nat-aux inc* (*Suc n*) *i* = *of-nat-aux inc n* (*inc i*) — tail recursive

lemma *of-nat-code* [*code*, *code unfold*, *code inline del*]:
of-nat n = of-nat-aux (λ*i*. *i* + *1*) *n 0*
proof (*induct n*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc n*)
have ⋀*i*. *of-nat-aux* (λ*i*. *i* + *1*) *n* (*i* + *1*) = *of-nat-aux* (λ*i*. *i* + *1*) *n i* + *1*

by (*induct n*) *simp-all*
from *this* [*of 0*] **have** *of-nat-aux* (λ*i*. *i* + *1*) *n 1* = *of-nat-aux* (λ*i*. *i* + *1*) *n 0* + *1*
by *simp*
with *Suc* **show** *?case* **by** (*simp add*: *add-commute*)
qed

end

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *semiring-char-0* = *semiring-1* +
assumes *of-nat-eq-iff* [*simp*]: *of-nat m* = *of-nat n* ⟷ *m* = *n*
begin

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*, *noatp*]: *0* = *of-nat n* ⟷ *0* = *n*
by (*rule of-nat-eq-iff* [*of 0*, *simplified*])

lemma *of-nat-eq-0-iff* [*simp*, *noatp*]: *of-nat m* = *0* ⟷ *m* = *0*
by (*rule of-nat-eq-iff* [*of - 0*, *simplified*])

lemma *inj-of-nat*: *inj of-nat*
by (*simp add*: *inj-on-def*)

end

context *ordered-semidom*
begin

lemma *zero-le-imp-of-nat*: *0* ≤ *of-nat m*
apply (*induct m*, *simp-all*)
apply (*erule order-trans*)
apply (*rule ord-le-eq-trans* [*OF - add-commute*])
apply (*rule less-add-one* [*THEN less-imp-le*])
done

lemma *less-imp-of-nat-less*: *m* < *n* ⟹ *of-nat m* < *of-nat n*
apply (*induct m n rule*: *diff-induct*, *simp-all*)
apply (*insert add-less-le-mono* [*OF zero-less-one zero-le-imp-of-nat*], *force*)
done

lemma *of-nat-less-imp-less*: *of-nat m* < *of-nat n* ⟹ *m* < *n*
apply (*induct m n rule*: *diff-induct*, *simp-all*)
apply (*insert zero-le-imp-of-nat*)
apply (*force simp add*: *not-less* [*symmetric*])
done

lemma *of-nat-less-iff* [*simp*]: *of-nat m* < *of-nat n* ⟷ *m* < *n*

by (*blast intro*: *of-nat-less-imp-less less-imp-of-nat-less*)

lemma *of-nat-le-iff* [*simp*]: *of-nat m* ≤ *of-nat n* ⟷ *m* ≤ *n*
by (*simp add*: *not-less* [*symmetric*] *linorder-not-less* [*symmetric*])

Every *ordered-semidom* has characteristic zero.

subclass *semiring-char-0*
proof qed (*simp add*: *eq-iff order-eq-iff*)

Special cases where either operand is zero

lemma *of-nat-0-le-iff* [*simp*]: *0* ≤ *of-nat n*
by (*rule of-nat-le-iff* [*of 0*, *simplified*])

lemma *of-nat-le-0-iff* [*simp*, *noatp*]: *of-nat m* ≤ *0* ⟷ *m* = *0*
by (*rule of-nat-le-iff* [*of - 0*, *simplified*])

lemma *of-nat-0-less-iff* [*simp*]: *0* < *of-nat n* ⟷ *0* < *n*
by (*rule of-nat-less-iff* [*of 0*, *simplified*])

lemma *of-nat-less-0-iff* [*simp*]: ¬ *of-nat m* < *0*
by (*rule of-nat-less-iff* [*of - 0*, *simplified*])

end

context *ring-1*
begin

lemma *of-nat-diff*: *n* ≤ *m* ⟹ *of-nat* (*m* − *n*) = *of-nat m* − *of-nat n*
by (*simp add*: *algebra-simps of-nat-add* [*symmetric*])

end

context *ordered-idom*
begin

lemma *abs-of-nat* [*simp*]: |*of-nat n*| = *of-nat n*
unfolding *abs-if* **by** *auto*

end

lemma *of-nat-id* [*simp*]: *of-nat n* = *n*
by (*induct n*) (*auto simp add*: *One-nat-def*)

lemma *of-nat-eq-id* [*simp*]: *of-nat* = *id*
by (*auto simp add*: *expand-fun-eq*)

12.6 The Set of Natural Numbers

context *semiring-1*

begin

definition
Nats :: *'a set* **where**
[*code del*]: *Nats* = *range of-nat*

notation (*xsymbols*)
Nats (ℕ)

lemma *of-nat-in-Nats* [*simp*]: *of-nat n* ∈ ℕ
by (*simp add*: *Nats-def*)

lemma *Nats-0* [*simp*]: *0* ∈ ℕ
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-0* [*symmetric*])
done

lemma *Nats-1* [*simp*]: *1* ∈ ℕ
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-1* [*symmetric*])
done

lemma *Nats-add* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a* + *b* ∈ ℕ
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-add* [*symmetric*])
done

lemma *Nats-mult* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a* * *b* ∈ ℕ
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-mult* [*symmetric*])
done

end

12.7 Further Arithmetic Facts Concerning the Natural Numbers

lemma *subst-equals*:
assumes *1*: *t* = *s* **and** *2*: *u* = *t*
shows *u* = *s*
using *2 1* **by** (*rule trans*)

setup *Arith-Data.setup*

use *Tools/nat-arith.ML*

declaration ⟪ *K Nat-Arith.setup* ⟫

use *Tools/lin-arith.ML*
declaration ⟪ *K Lin-Arith.setup* ⟫

lemmas [*arith-split*] = *nat-diff-split split-min split-max*

context *order*
begin

lemma *lift-Suc-mono-le*:
  assumes *mono*: !!*n*. *f n* ≤ *f*(*Suc n*) **and** *n*≤*n'*
  shows *f n* ≤ *f n'*
proof (*cases n* < *n'*)
  case *True*
  thus *?thesis*
    by (*induct n n' rule*: *less-Suc-induct*[*consumes 1*]) (*auto intro*: *mono*)
qed (*insert* ‹*n* ≤ *n'*›, *auto*) — trivial for *n* = *n'*

lemma *lift-Suc-mono-less*:
  assumes *mono*: !!*n*. *f n* < *f*(*Suc n*) **and** *n* < *n'*
  shows *f n* < *f n'*
using ‹*n* < *n'*›
by (*induct n n' rule*: *less-Suc-induct*[*consumes 1*]) (*auto intro*: *mono*)

lemma *lift-Suc-mono-less-iff*:
  (!!*n*. *f n* < *f*(*Suc n*)) ⟹ *f*(*n*) < *f*(*m*) ⟷ *n*<*m*
by(*blast intro*: *less-asym'* *lift-Suc-mono-less*[*of f*]
        dest: *linorder-not-less*[*THEN iffD1*] *le-eq-less-or-eq*[*THEN iffD1*])

end

lemma *mono-iff-le-Suc*: *mono f* = (∀ *n*. *f n* ≤ *f* (*Suc n*))
unfolding *mono-def*
by (*auto intro*:*lift-Suc-mono-le*[*of f*])

lemma *mono-nat-linear-lb*:
  (!!*m n*::*nat*. *m*<*n* ⟹ *f m* < *f n*) ⟹ *f*(*m*)+*k* ≤ *f*(*m*+*k*)
apply(*induct-tac k*)
 apply *simp*
apply(*erule-tac x*=*m*+*n* **in** *meta-allE*)
apply(*erule-tac x*=*Suc*(*m*+*n*) **in** *meta-allE*)
apply *simp*
done

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*: [| *a* < (*b*::*nat*); *c* ≤ *a* |] ==> *a*−*c* < *b*−*c*
by *arith*

lemma *less-diff-conv*: $(i < j-k) = (i+k < (j::nat))$
by *arith*

lemma *le-diff-conv*: $(j-k \leq (i::nat)) = (j \leq i+k)$
by *arith*

lemma *le-diff-conv2*: $k \leq j \Longrightarrow (i \leq j-k) = (i+k \leq (j::nat))$
by *arith*

lemma *diff-diff-cancel* [*simp*]: $i \leq (n::nat) \Longrightarrow n - (n - i) = i$
by *arith*

lemma *le-add-diff*: $k \leq (n::nat) \Longrightarrow m \leq n + m - k$
by *arith*

lemma *diff-less*[*simp*]: $!!m::nat.\ [|\ 0<n;\ 0<m\ |] \Longrightarrow m - n < m$
by *arith*

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow ((m-k) - (n-k)) = (m-n)$
by (*simp split add*: *nat-diff-split*)

lemma *eq-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k = n-k) = (m=n)$
by (*auto split add*: *nat-diff-split*)

lemma *less-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k < n-k) = (m<n)$
by (*auto split add*: *nat-diff-split*)

lemma *le-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k \leq n-k) = (m \leq n)$
by (*auto split add*: *nat-diff-split*)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq (n::nat) \Longrightarrow (m-l) \leq (n-l)$
by (*simp split add*: *nat-diff-split*)

lemma *diff-le-mono2*: $m \leq (n::nat) \Longrightarrow (l-n) \leq (l-m)$
by (*simp split add*: *nat-diff-split*)

lemma *diff-less-mono2*: $[|\ m < (n::nat);\ m<l\ |] \Longrightarrow (l-n) < (l-m)$
by (*simp split add*: *nat-diff-split*)

lemma *diffs0-imp-equal*: $!!m::nat.\ [|\ m-n = 0;\ n-m = 0\ |] \Longrightarrow\ m=n$
by (*simp split add*: *nat-diff-split*)

lemma *min-diff*: $min\ (m - (i::nat))\ (n - i) = min\ m\ n - i$
unfolding *min-def* **by** *auto*

lemma *inj-on-diff-nat*:

assumes *k-le-n*: $\forall n \in N.\ k \leq (n::nat)$
shows *inj-on* $(\lambda n.\ n - k)\ N$
proof (*rule inj-onI*)
fix $x\ y$
assume a: $x \in N\ y \in N\ x - k = y - k$
with *k-le-n* **have** $x - k + k = y - k + k$ **by** *auto*
with a *k-le-n* **show** $x = y$ **by** *auto*
qed

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j \longrightarrow i - (j - k) = i + (k::nat) - j$
by *arith*

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j \Longrightarrow m - Suc\ (j - k) = m + k - Suc\ j$
by *arith*

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j \Longrightarrow Suc\ (j - k) - m = Suc\ j - (k + m)$
by *arith*

Lemmas for ex/Factorization

lemma *one-less-mult*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow Suc\ 0 < m*n$
by (*cases* m) *auto*

lemma *n-less-m-mult-n*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow n<m*n$
by (*cases* m) *auto*

lemma *n-less-n-mult-m*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow n<n*m$
by (*cases* m) *auto*

Specialized induction principles that work "backwards":

lemma *inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: $i <= j$
assumes *base*: $P\ j$
assumes *step*: $!!i.\ [|\ i < j;\ P\ (Suc\ i)\ |] \Longrightarrow P\ i$
shows $P\ i$
using *less*
proof (*induct* $d{==}j - i$ *arbitrary*: i)
case (*0* i)
hence $i = j$ **by** *simp*
with *base* **show** *?case* **by** *simp*
next
case (*Suc* $d\ i$)
hence $i < j\ P\ (Suc\ i)$
by *simp-all*
thus $P\ i$ **by** (*rule step*)
qed

lemma *strict-inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: $i < j$

assumes *base*: !!*i*. *j* = *Suc i* ==> *P i*
assumes *step*: !!*i*. [| *i* < *j*; *P* (*Suc i*) |] ==> *P i*
shows *P i*
using *less*
proof (*induct d*==*j* − *i* − *1 arbitrary*: *i*)
case (*0 i*)
with ‹*i* < *j*› **have** *j* = *Suc i* **by** *simp*
with *base* **show** *?case* **by** *simp*
next
case (*Suc d i*)
hence *i* < *j P* (*Suc i*)
by *simp-all*
thus *P i* **by** (*rule step*)
qed

lemma *zero-induct-lemma*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P* (*k* − *i*)
using *inc-induct*[*of k* − *i k P*, *simplified*] **by** *blast*

lemma *zero-induct*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P 0*
using *inc-induct*[*of 0 k P*] **by** *blast*

lemma *nat-not-singleton*: (∀ *x*. *x* = (*0*::*nat*)) = *False*
by *auto*

lemmas *add-diff-assoc* = *diff-add-assoc* [*symmetric*]
lemmas *add-diff-assoc2* = *diff-add-assoc2*[*symmetric*]
declare *diff-diff-left* [*simp*] *add-diff-assoc* [*simp*] *add-diff-assoc2*[*simp*]

At present we prove no analogue of *not-less-Least* or *Least-Suc*, since there appears to be no need.

12.8 size of a datatype value

class *size* =
fixes *size* :: *'a* ⇒ *nat* — see further theory *Wellfounded*

end

13 Product-Type: Cartesian products

theory *Product-Type*
imports *Inductive*
uses
(*Tools/split-rule.ML*)
(*Tools/inductive-set-package.ML*)
(*Tools/inductive-realizer.ML*)
(*Tools/datatype-realizer.ML*)

begin

13.1 *bool* is a datatype

rep-datatype *True False* **by** (*auto intro*: *bool-induct*)

declare *case-split* [*cases type*: *bool*]
— prefer plain propositional version

lemma
shows [*code*]: *eq-class.eq False P* ⟷ ¬ *P*
and [*code*]: *eq-class.eq True P* ⟷ *P*
and [*code*]: *eq-class.eq P False* ⟷ ¬ *P*
and [*code*]: *eq-class.eq P True* ⟷ *P*
and [*code nbe*]: *eq-class.eq P P* ⟷ *True*
by (*simp-all add*: *eq*)

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

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

13.2 Unit

typedef *unit* = {*True*}
proof
show *True* : *?unit* **..**
qed

definition
Unity :: *unit* ("()")
where
() = *Abs-unit True*

lemma *unit-eq* [*noatp*]: *u* = ()
by (*induct u*) (*simp add*: *unit-def Unity-def*)

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

ML ⟪
val unit-eq-proc =
let val unit-meta-eq = *mk-meta-eq* @{*thm unit-eq*} *in*
Simplifier.simproc @{*theory*} *unit-eq* [*x*::*unit*]
(*fn* - => *fn* - => *fn t* => *if HOLogic.is-unit t then NONE else SOME unit-meta-eq*)
end;

Addsimprocs [*unit-eq-proc*];

⟫

rep-datatype () **by** *simp*

lemma *unit-all-eq1*: (!!x::*unit*. *PROP P x*) == *PROP P* ()
by *simp*

lemma *unit-all-eq2*: (!!x::*unit*. *PROP P*) == *PROP P*
by (*rule triv-forall-equality*)

This rewrite counters the effect of *unit-eq-proc* on %u::*unit*. f u, replacing it by f rather than by %u. f ().

lemma *unit-abs-eta-conv* [*simp*,*noatp*]: (%u::*unit*. f ()) = f
by (*rule ext*) *simp*

code generator setup

instance *unit* :: *eq* **..**

lemma [*code*]:
eq-class.eq (u::*unit*) v ⟷ *True* **unfolding** *eq unit-eq* [*of u*] *unit-eq* [*of v*] **by** *rule*+

code-type *unit*
(*SML unit*)
(*OCaml unit*)
(*Haskell* ())

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

code-const *eq-class.eq* :: *unit* ⇒ *unit* ⇒ *bool*
(*Haskell* **infixl** 4 ==)

code-const *Unity*
(*SML* ())
(*OCaml* ())
(*Haskell* ())

code-reserved *SML*
unit

code-reserved *OCaml*
unit

13.3 Pairs

13.3.1 Product type, basic operations and concrete syntax

definition

Pair-Rep :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where
Pair-Rep a b = $(\lambda x\ y.\ x = a \wedge y = b)$

global

typedef (*Prod*)
($'a$, $'b$) $*$ (**infixr** $*$ *20*)
= $\{f.\ \exists\ a\ b.\ f = Pair\text{-}Rep\ (a::'a)\ (b::'b)\}$
proof
fix *a b* **show** *Pair-Rep a b* $\in$ *?Prod*
by *rule+*
qed

syntax (*xsymbols*)
$*$ :: [*type*, *type*] => *type* (($-$ $\times$/ $-$) [*21*, *20*] *20*)
syntax (*HTML* **output**)
$*$ :: [*type*, *type*] => *type* (($-$ $\times$/ $-$) [*21*, *20*] *20*)

consts
Pair :: $'a \Rightarrow 'b \Rightarrow 'a \times 'b$
fst :: $'a \times 'b \Rightarrow 'a$
snd :: $'a \times 'b \Rightarrow 'b$
split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
curry :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$

local

defs
Pair-def: *Pair a b* == *Abs-Prod* (*Pair-Rep a b*)
fst-def: *fst p* == *THE a. EX b. p* = *Pair a b*
snd-def: *snd p* == *THE b. EX a. p* = *Pair a b*
split-def: *split* == (%*c p. c* (*fst p*) (*snd p*))
curry-def: *curry* == (%*c x y. c* (*Pair x y*))

Patterns – extends pre-defined type *pttrn* used in abstractions.

nonterminals
tuple-args patterns

syntax
-tuple :: $'a$ => *tuple-args* => $'a * 'b$ (($1'(-,/\ -')$))
-tuple-arg :: $'a$ => *tuple-args* (-)
-tuple-args :: $'a$ => *tuple-args* => *tuple-args* (-,/ -)
-pattern :: [*pttrn*, *patterns*] => *pttrn* ($'(-,/\ -')$)
:: *pttrn* => *patterns* (-)
-patterns :: [*pttrn*, *patterns*] => *patterns* (-,/ -)

translations
(x, y) == *Pair x y*

```
  -tuple x (-tuple-args y z) == -tuple x (-tuple-arg (-tuple y z))
  %(x,y,zs).b  == split(%x (y,zs).b)
  %(x,y).b     == split(%x y. b)
  -abs (Pair x y) t => %(x,y).t
```

print-translation ⟪

```
let fun split-tr' [Abs (x,T,t as (Abs abs))] =
      (* split (%x y. t) => %(x,y) t *)
      let val (y,t') = atomic-abs-tr' abs;
          val (x',t'') = atomic-abs-tr' (x,T,t');

      in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end
    | split-tr' [Abs (x,T,(s as Const (split,-)$t))] =
      (* split (%x. (split (%y z. t))) => %(x,y,z). t *)
      let val (Const (-abs,-)$(Const (-pattern,-)$y$z)$t') = split-tr' [t];
          val (x',t'') = atomic-abs-tr' (x,T,t');
      in Syntax.const -abs$
          (Syntax.const -pattern$x'$(Syntax.const -patterns$y$z))$t'' end
    | split-tr' [Const (split,-)$t] =
      (* split (split (%x y z. t)) => %((x,y),z). t *)
      split-tr' [(split-tr' [t])] (* inner split-tr' creates next pattern *)
    | split-tr' [Const (-abs,-)$x-y$(Abs abs)] =
      (* split (%pttrn z. t) => %(pttrn,z). t *)
      let val (z,t) = atomic-abs-tr' abs;
      in Syntax.const -abs $ (Syntax.const -pattern $x-y$z) $ t end
    | split-tr' - =  raise Match;
in [(split, split-tr')]
end
```

⟫

typed-print-translation ⟪

```
let
  fun split-guess-names-tr' - T [Abs (x,-,Abs -)] = raise Match
    | split-guess-names-tr' - T  [Abs (x,xT,t)] =
        (case (head-of t) of
           Const (split,-) => raise Match
         | - => let
                  val (-::yT::-) = binder-types (domain-type T) handle Bind => raise
Match;
                  val (y,t') = atomic-abs-tr' (y,yT,(incr-boundvars 1 t)$Bound 0);
                  val (x',t'') = atomic-abs-tr' (x,xT,t');
                in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end)
    | split-guess-names-tr' - T [t] =
      (case (head-of t) of
          Const (split,-) => raise Match
```

```
      | - => let
              val (xT::yT::-) = binder-types (domain-type T) handle Bind =>
raise Match;
              val (y,t') =
                  atomic-abs-tr' (y,yT,(incr-boundvars 2 t)$Bound 1$Bound 0);
              val (x',t'') = atomic-abs-tr' (x,xT,t');
            in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end)
   | split-guess-names-tr' - - - = raise Match;
in [(split, split-guess-names-tr')]
end
⟫
```

Towards a datatype declaration

```
lemma surj-pair [simp]: EX x y. p = (x, y)
  apply (unfold Pair-def)
  apply (rule Rep-Prod [unfolded Prod-def, THEN CollectE])
  apply (erule exE, erule exE, rule exI, rule exI)
  apply (rule Rep-Prod-inverse [symmetric, THEN trans])
  apply (erule arg-cong)
  done

lemma PairE [cases type: *]:
  obtains x y where p = (x, y)
  using surj-pair [of p] by blast

lemma ProdI: Pair-Rep a b ∈ Prod
  unfolding Prod-def by rule+

lemma Pair-Rep-inject: Pair-Rep a b = Pair-Rep a' b' ⟹ a = a' ∧ b = b'
  unfolding Pair-Rep-def by (drule fun-cong, drule fun-cong) blast

lemma inj-on-Abs-Prod: inj-on Abs-Prod Prod
  apply (rule inj-on-inverseI)
  apply (erule Abs-Prod-inverse)
  done

lemma Pair-inject:
  assumes (a, b) = (a', b')
    and a = a' ==> b = b' ==> R
  shows R
  apply (insert prems [unfolded Pair-def])
   apply (rule inj-on-Abs-Prod [THEN inj-onD, THEN Pair-Rep-inject, THEN
conjE])
  apply (assumption | rule ProdI)+
  done

rep-datatype (prod) Pair
proof −
  fix P p
```

assume $\bigwedge x\ y.\ P\ (x,\ y)$
then show $P\ p$ **by** (*cases* p) *simp*
qed (*auto elim*: *Pair-inject*)

lemmas *Pair-eq* = *prod.inject*

lemma *fst-conv* [*simp*, *code*]: *fst* $(a,\ b) = a$
unfolding *fst-def* **by** *blast*

lemma *snd-conv* [*simp*, *code*]: *snd* $(a,\ b) = b$
unfolding *snd-def* **by** *blast*

13.3.2 Basic rules and proof tools

lemma *fst-eqD*: *fst* $(x,\ y) = a$ ==> $x = a$
by *simp*

lemma *snd-eqD*: *snd* $(x,\ y) = a$ ==> $y = a$
by *simp*

lemma *pair-collapse* [*simp*]: (*fst* p, *snd* p) = p
by (*cases* p) *simp*

lemmas *surjective-pairing* = *pair-collapse* [*symmetric*]

lemma *split-paired-all*: (!!x. *PROP* $P\ x$) == (!!$a\ b$. *PROP* $P\ (a,\ b)$)
proof
fix $a\ b$
assume !!x. *PROP* $P\ x$
then show *PROP* $P\ (a,\ b)$ **.**
next
fix x
assume !!$a\ b$. *PROP* $P\ (a,\ b)$
from ‹*PROP* P (*fst* x, *snd* x)› **show** *PROP* $P\ x$ **by** *simp*
qed

The rule *split-paired-all* does not work with the Simplifier because it also affects premises in congrence rules, where this can lead to premises of the form !!$a\ b$. ... = *?P*$(a,\ b)$ which cannot be solved by reflexivity.

lemmas *split-tupled-all* = *split-paired-all unit-all-eq2*

ML ‹‹
(* *replace parameters of product type by individual component parameters* *)
val safe-full-simp-tac = *generic-simp-tac true* (*true*, *false*, *false*);
local (* *filtering with exists-paired-all is an essential optimization* *)
fun exists-paired-all (*Const* (*all*, -) $ *Abs* (-, *T*, *t*)) =
can HOLogic.dest-prodT T orelse exists-paired-all t
| *exists-paired-all* (*t* $ *u*) = *exists-paired-all t orelse exists-paired-all u*
| *exists-paired-all* (*Abs* (-, -, *t*)) = *exists-paired-all t*

```
    | exists-paired-all - = false;
  val ss = HOL-basic-ss
   addsimps [@{thm split-paired-all}, @{thm unit-all-eq2}, @{thm unit-abs-eta-conv}]
    addsimprocs [unit-eq-proc];
in
  val split-all-tac = SUBGOAL (fn (t, i) =>
    if exists-paired-all t then safe-full-simp-tac ss i else no-tac);
  val unsafe-split-all-tac = SUBGOAL (fn (t, i) =>
    if exists-paired-all t then full-simp-tac ss i else no-tac);
  fun split-all th =
   if exists-paired-all (Thm.prop-of th) then full-simplify ss th else th;
end;
⟫

declaration ⟪ fn - =>
  Classical.map-cs (fn cs => cs addSbefore (split-all-tac, split-all-tac))
⟫

lemma split-paired-All [simp]: (ALL x. P x) = (ALL a b. P (a, b))
  — [iff] is not a good idea because it makes blast loop
  by fast

lemma split-paired-Ex [simp]: (EX x. P x) = (EX a b. P (a, b))
  by fast

lemma Pair-fst-snd-eq: s = t ⟷ fst s = fst t ∧ snd s = snd t
  by (cases s, cases t) simp

lemma prod-eqI [intro?]: fst p = fst q ⟹ snd p = snd q ⟹ p = q
  by (simp add: Pair-fst-snd-eq)
```

13.3.3 *split* and *curry*

```
lemma split-conv [simp, code]: split f (a, b) = f a b
  by (simp add: split-def)

lemma curry-conv [simp, code]: curry f a b = f (a, b)
  by (simp add: curry-def)

lemmas split = split-conv  — for backwards compatibility

lemma splitI: f a b ⟹ split f (a, b)
  by (rule split-conv [THEN iffD2])

lemma splitD: split f (a, b) ⟹ f a b
  by (rule split-conv [THEN iffD1])

lemma curryI [intro!]: f (a, b) ⟹ curry f a b
  by (simp add: curry-def)
```

lemma *curryD* [*dest*!]: *curry f a b* $\Longrightarrow$ *f* (*a*, *b*)
by (*simp add*: *curry-def*)

lemma *curryE*: *curry f a b* $\Longrightarrow$ (*f* (*a*, *b*) $\Longrightarrow$ *Q*) $\Longrightarrow$ *Q*
by (*simp add*: *curry-def*)

lemma *curry-split* [*simp*]: *curry* (*split f*) = *f*
by (*simp add*: *curry-def split-def*)

lemma *split-curry* [*simp*]: *split* (*curry f*) = *f*
by (*simp add*: *curry-def split-def*)

lemma *split-Pair* [*simp*]: (λ(*x*, *y*). (*x*, *y*)) = *id*
by (*simp add*: *split-def id-def*)

lemma *split-eta*: (λ(*x*, *y*). *f* (*x*, *y*)) = *f*
— Subsumes the old *split-Pair* when *f* is the identity function.
by (*rule ext*) *auto*

lemma *split-comp*: *split* (*f* ∘ *g*) *x* = *f* (*g* (*fst x*)) (*snd x*)
by (*cases x*) *simp*

lemma *split-twice*: *split f* (*split g p*) = *split* (λ*x y*. *split f* (*g x y*)) *p*
unfolding *split-def* **..**

lemma *split-paired-The*: (*THE x*. *P x*) = (*THE* (*a*, *b*). *P* (*a*, *b*))
— Can't be added to simpset: loops!
by (*simp add*: *split-eta*)

lemma *The-split*: *The* (*split P*) = (*THE xy*. *P* (*fst xy*) (*snd xy*))
by (*simp add*: *split-def*)

lemma *split-weak-cong*: *p* = *q* $\Longrightarrow$ *split c p* = *split c q*
— Prevents simplification of *c*: much faster
by (*erule arg-cong*)

lemma *cond-split-eta*: (!!*x y*. *f x y* = *g* (*x*, *y*)) ==> (%(*x*, *y*). *f x y*) = *g*
by (*simp add*: *split-eta*)

Simplification procedure for *cond-split-eta*. Using *split-eta* as a rewrite rule is not general enough, and using *cond-split-eta* directly would render some existing proofs very inefficient; similarly for *split-beta*.

ML ⟪

```
local
  val cond-split-eta-ss = HOL-basic-ss addsimps [thm cond-split-eta]
  fun  Pair-pat k 0 (Bound m) = (m = k)
  |    Pair-pat k i (Const (Pair,  -) $ Bound m $ t) = i > 0 andalso
```

```
                m = k+i andalso Pair-pat k (i-1) t
  |   Pair-pat - - - = false;
  fun no-args k i (Abs (-, -, t)) = no-args (k+1) i t
  |   no-args k i (t $ u) = no-args k i t andalso no-args k i u
  |   no-args k i (Bound m) = m < k orelse m > k+i
  |   no-args - - - = true;
  fun split-pat tp i (Abs (-,-,t)) = if tp 0 i t then SOME (i,t) else NONE
  |   split-pat tp i (Const (split, -) $ Abs (-, -, t)) = split-pat tp (i+1) t
  |   split-pat tp i - = NONE;
  fun metaeq ss lhs rhs = mk-meta-eq (Goal.prove (Simplifier.the-context ss) [] []
        (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs,rhs)))
        (K (simp-tac (Simplifier.inherit-context ss cond-split-eta-ss) 1)));

  fun beta-term-pat k i (Abs (-, -, t)) = beta-term-pat (k+1) i t
  |   beta-term-pat k i (t $ u) = Pair-pat k i (t $ u) orelse
                  (beta-term-pat k i t andalso beta-term-pat k i u)
  |   beta-term-pat k i t = no-args k i t;
  fun  eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
  |    eta-term-pat - - - = false;
  fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
  |   subst arg k i (t $ u) = if Pair-pat k i (t $ u) then incr-boundvars k arg
                       else (subst arg k i t $ subst arg k i u)
  |   subst arg k i t = t;
  fun beta-proc ss (s as Const (split, -) $ Abs (-, -, t) $ arg) =
        (case split-pat beta-term-pat 1 t of
        SOME (i,f) => SOME (metaeq ss s (subst arg 0 i f))
        | NONE => NONE)
  |   beta-proc - - = NONE;
  fun eta-proc ss (s as Const (split, -) $ Abs (-, -, t)) =
        (case split-pat eta-term-pat 1 t of
          SOME (-,ft) => SOME (metaeq ss s (let val (f $ arg) = ft in f end))
        | NONE => NONE)
  |   eta-proc - - = NONE;
in
  val split-beta-proc = Simplifier.simproc (the-context ()) split-beta [split f z] (K
beta-proc);
  val split-eta-proc = Simplifier.simproc (the-context ()) split-eta [split f] (K eta-proc);
end;

Addsimprocs [split-beta-proc, split-eta-proc];
⟫
```

lemma *split-beta* [*mono*]: (%(*x*, *y*). *P x y*) *z* = *P* (*fst z*) (*snd z*)
by (*subst surjective-pairing*, *rule split-conv*)

lemma *split-split* [*noatp*]: *R*(*split c p*) = (*ALL x y*. *p* = (*x*, *y*) −−> *R*(*c x y*))
— For use with *split* and the Simplifier.
by (*insert surj-pair* [*of p*], *clarify*, *simp*)

split-split could be declared as [*split*] done after the Splitter has been speeded

up significantly; precompute the constants involved and don't do anything unless the current goal contains one of those constants.

lemma *split-split-asm* [*noatp*]: *R* (*split c p*) = (~(*EX x y. p* = (*x*, *y*) & (~*R* (*c x y*))))
by (*subst split-split*, *simp*)

split used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *split* as rewrite.

lemma *splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *split c p*
  apply (*simp only*: *split-tupled-all*)
  apply (*simp* (*no-asm-simp*))
  done

lemma *splitI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *split c p x*
  apply (*simp only*: *split-tupled-all*)
  apply (*simp* (*no-asm-simp*))
  done

lemma *splitE*: *split c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
  by (*induct p*) (*auto simp add*: *split-def*)

lemma *splitE'*: *split c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
  by (*induct p*) (*auto simp add*: *split-def*)

lemma *splitE2*:
  [| *Q* (*split P z*); !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R* |] ==> *R*
proof −
  assume *q*: *Q* (*split P z*)
  assume *r*: !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R*
  show *R*
    apply (*rule r surjective-pairing*)+
    apply (*rule split-beta* [*THEN subst*], *rule q*)
    done
qed

lemma *splitD'*: *split R* (*a*,*b*) *c* ==> *R a b c*
  by *simp*

lemma *mem-splitI*: *z*: *c a b* ==> *z*: *split c* (*a*, *b*)
  by *simp*

lemma *mem-splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *z*: *c a b* |] ==> *z*: *split c p*
by (*simp only*: *split-tupled-all*, *simp*)

lemma *mem-splitE*:
  assumes *major*: *z*: *split c p*

and *cases*: !!*x y*. [| *p* = (*x*,*y*); *z*: *c x y* |] ==> *Q*
shows *Q*
by (*rule major* [*unfolded split-def*] *cases surjective-pairing*)+

declare *mem-splitI2* [*intro*!] *mem-splitI* [*intro*!] *splitI2'* [*intro*!] *splitI2* [*intro*!] *splitI* [*intro*!]
declare *mem-splitE* [*elim*!] *splitE'* [*elim*!] *splitE* [*elim*!]

ML ⟪
local (∗ *filtering with exists-p-split is an essential optimization* ∗)
fun exists-p-split (*Const* (*split*,-) $ - $ (*Const* (*Pair*,-)$-$-)) = *true*
| *exists-p-split* (*t* $ *u*) = *exists-p-split t orelse exists-p-split u*
| *exists-p-split* (*Abs* (-, -, *t*)) = *exists-p-split t*
| *exists-p-split* - = *false*;
val ss = *HOL-basic-ss addsimps* [*thm split-conv*];
in
val split-conv-tac = *SUBGOAL* (*fn* (*t*, *i*) =>
if exists-p-split t then safe-full-simp-tac ss i else no-tac);
end;
⟫

declaration ⟪ *fn* - =>
Classical.map-cs (*fn cs* => *cs addSbefore* (*split-conv-tac*, *split-conv-tac*))
⟫

lemma *split-eta-SetCompr* [*simp*,*noatp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P* (*x*, *y*)) = *P*
by (*rule ext*) *fast*

lemma *split-eta-SetCompr2* [*simp*,*noatp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P x y*) = *split P*
by (*rule ext*) *fast*

lemma *split-part* [*simp*]: (%(*a*,*b*). *P* & *Q a b*) = (%*ab*. *P* & *split Q ab*)
— Allows simplifications of nested splits in case of independent predicates.
by (*rule ext*) *blast*

lemma *split-comp-eq*:
fixes *f* :: *'a* => *'b* => *'c* **and** *g* :: *'d* => *'a*
shows (%*u*. *f* (*g* (*fst u*)) (*snd u*)) = (*split* (%*x*. *f* (*g x*)))
by (*rule ext*) *auto*

lemma *pair-imageI* [*intro*]: (*a*, *b*) : *A* ==> *f a b* : (%(*a*, *b*). *f a b*) ' *A*
apply (*rule-tac x* = (*a*, *b*) **in** *image-eqI*)
apply *auto*
done

lemma *The-split-eq* [*simp*]: (*THE* (*x'*,*y'*). *x* = *x'* & *y* = *y'*) = (*x*, *y*)
by *blast*

Setup of internal *split-rule*.

definition
internal-split :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a* × '*b* ⇒ '*c*
where
internal-split == *split*

lemma *internal-split-conv*: *internal-split c* (*a*, *b*) = *c a b*
by (*simp only*: *internal-split-def split-conv*)

hide *const internal-split*

use *Tools/split-rule.ML*
setup *SplitRule.setup*

lemmas *prod-caseI* = *prod.cases* [*THEN iffD2*, *standard*]

lemma *prod-caseI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *prod-case c p*
by *auto*

lemma *prod-caseI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *prod-case c p x*
by (*auto simp*: *split-tupled-all*)

lemma *prod-caseE*: *prod-case c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
by (*induct p*) *auto*

lemma *prod-caseE'*: *prod-case c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
by (*induct p*) *auto*

lemma *prod-case-unfold*: *prod-case* = (%*c p*. *c* (*fst p*) (*snd p*))
by (*simp add*: *expand-fun-eq*)

declare *prod-caseI2'* [*intro*!] *prod-caseI2* [*intro*!] *prod-caseI* [*intro*!]
declare *prod-caseE'* [*elim*!] *prod-caseE* [*elim*!]

lemma *prod-case-split*:
prod-case = *split*
by (*auto simp add*: *expand-fun-eq*)

lemma *prod-case-beta*:
prod-case f p = *f* (*fst p*) (*snd p*)
unfolding *prod-case-split split-beta* **..**

13.4 Further cases/induct rules for tuples

lemma *prod-cases3* [*cases type*]:
obtains (*fields*) *a b c* **where** $y = (a, b, c)$
by (*cases y*, *case-tac b*) *blast*

lemma *prod-induct3* [*case-names fields*, *induct type*]:
$(!!a\ b\ c.\ P\ (a, b, c)) ==> P\ x$
by (*cases x*) *blast*

lemma *prod-cases4* [*cases type*]:
obtains (*fields*) *a b c d* **where** $y = (a, b, c, d)$
by (*cases y*, *case-tac c*) *blast*

lemma *prod-induct4* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d.\ P\ (a, b, c, d)) ==> P\ x$
by (*cases x*) *blast*

lemma *prod-cases5* [*cases type*]:
obtains (*fields*) *a b c d e* **where** $y = (a, b, c, d, e)$
by (*cases y*, *case-tac d*) *blast*

lemma *prod-induct5* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e.\ P\ (a, b, c, d, e)) ==> P\ x$
by (*cases x*) *blast*

lemma *prod-cases6* [*cases type*]:
obtains (*fields*) *a b c d e f* **where** $y = (a, b, c, d, e, f)$
by (*cases y*, *case-tac e*) *blast*

lemma *prod-induct6* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f.\ P\ (a, b, c, d, e, f)) ==> P\ x$
by (*cases x*) *blast*

lemma *prod-cases7* [*cases type*]:
obtains (*fields*) *a b c d e f g* **where** $y = (a, b, c, d, e, f, g)$
by (*cases y*, *case-tac f*) *blast*

lemma *prod-induct7* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f\ g.\ P\ (a, b, c, d, e, f, g)) ==> P\ x$
by (*cases x*) *blast*

13.4.1 Derived operations

The composition-uncurry combinator.

notation *fcomp* (**infixl** *o>* *60*)

definition
scomp :: $('a \Rightarrow 'b \times 'c) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (**infixl** *o→* *60*)

where
  $f\ o{\rightarrow}\ g = (\lambda x.\ split\ g\ (f\ x))$

lemma *scomp-apply*: $(f\ o{\rightarrow}\ g)\ x = split\ g\ (f\ x)$
  by (*simp add*: *scomp-def*)

lemma *Pair-scomp*: $Pair\ x\ o{\rightarrow}\ f = f\ x$
  by (*simp add*: *expand-fun-eq scomp-apply*)

lemma *scomp-Pair*: $x\ o{\rightarrow}\ Pair = x$
  by (*simp add*: *expand-fun-eq scomp-apply*)

lemma *scomp-scomp*: $(f\ o{\rightarrow}\ g)\ o{\rightarrow}\ h = f\ o{\rightarrow}\ (\lambda x.\ g\ x\ o{\rightarrow}\ h)$
  by (*simp add*: *expand-fun-eq split-twice scomp-def*)

lemma *scomp-fcomp*: $(f\ o{\rightarrow}\ g)\ o{>}\ h = f\ o{\rightarrow}\ (\lambda x.\ g\ x\ o{>}\ h)$
  by (*simp add*: *expand-fun-eq scomp-apply fcomp-def split-def*)

lemma *fcomp-scomp*: $(f\ o{>}\ g)\ o{\rightarrow}\ h = f\ o{>}\ (g\ o{\rightarrow}\ h)$
  by (*simp add*: *expand-fun-eq scomp-apply fcomp-apply*)

no-notation *fcomp* (**infixl** $o{>}$ *60*)
no-notation *scomp* (**infixl** $o{\rightarrow}$ *60*)

prod-fun — action of the product functor upon functions.

definition *prod-fun* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'd$ **where**
  [*code del*]: $prod\text{-}fun\ f\ g = (\lambda(x,\ y).\ (f\ x,\ g\ y))$

lemma *prod-fun* [*simp*, *code*]: $prod\text{-}fun\ f\ g\ (a,\ b) = (f\ a,\ g\ b)$
  by (*simp add*: *prod-fun-def*)

lemma *prod-fun-compose*: *prod-fun* (*f1 o f2*) (*g1 o g2*) = (*prod-fun f1 g1 o prod-fun f2 g2*)
  by (*rule ext*) *auto*

lemma *prod-fun-ident* [*simp*]: *prod-fun* (%*x*. *x*) (%*y*. *y*) = (%*z*. *z*)
  by (*rule ext*) *auto*

lemma *prod-fun-imageI* [*intro*]: $(a,\ b) : r \Longrightarrow (f\ a,\ g\ b) : prod\text{-}fun\ f\ g\ `\ r$
  apply (*rule image-eqI*)
  apply (*rule prod-fun* [*symmetric*], *assumption*)
  done

lemma *prod-fun-imageE* [*elim*!]:
  assumes *major*: *c*: (*prod-fun f g*)‘*r*
    and *cases*: !!*x y*. [| *c*=(*f*(*x*),*g*(*y*)); (*x*,*y*):*r* |] ==> *P*
  shows *P*
  apply (*rule major* [*THEN imageE*])
  apply (*rule-tac p* = *x* **in** *PairE*)

apply (*rule cases*)
apply (*blast intro*: *prod-fun*)
apply *blast*
done

definition
apfst :: ('*a* ⇒ '*c*) ⇒ '*a* × '*b* ⇒ '*c* × '*b*
where
[*code del*]: *apfst f* = *prod-fun f id*

definition
apsnd :: ('*b* ⇒ '*c*) ⇒ '*a* × '*b* ⇒ '*a* × '*c*
where
[*code del*]: *apsnd f* = *prod-fun id f*

lemma *apfst-conv* [*simp*, *code*]:
apfst f (*x*, *y*) = (*f x*, *y*)
by (*simp add*: *apfst-def*)

lemma *upd-snd-conv* [*simp*, *code*]:
apsnd f (*x*, *y*) = (*x*, *f y*)
by (*simp add*: *apsnd-def*)

Disjoint union of a family of sets – Sigma.

definition *Sigma* :: ['*a set*, '*a* => '*b set*] => ('*a* × '*b*) *set* **where**
Sigma-def: *Sigma A B* == *UN x*:*A*. *UN y*:*B x*. {*Pair x y*}

abbreviation
Times :: ['*a set*, '*b set*] => ('*a* * '*b*) *set*
(**infixr** <*> *80*) **where**
A <*> *B* == *Sigma A* (%-. *B*)

notation (*xsymbols*)
Times (**infixr** × *80*)

notation (*HTML* **output**)
Times (**infixr** × *80*)

syntax
@*Sigma* ::[*pttrn*, '*a set*, '*b set*] => ('*a* * '*b*) *set* ((*3SIGMA* -:-./ -) [*0*, *0*, *10*] *10*)

translations
SIGMA x:*A*. *B* == *Product-Type.Sigma A* (%*x*. *B*)

lemma *SigmaI* [*intro*!]: [| *a*:*A*; *b*:*B*(*a*) |] ==> (*a*,*b*) : *Sigma A B*
by (*unfold Sigma-def*) *blast*

lemma *SigmaE* [*elim*!]:
[| *c*: *Sigma A B*;

!!x y.[| x:A; y:$B(x)$; c=(x,y) |] ==> P
|] ==> P
— The general elimination rule.
by (*unfold Sigma-def*) *blast*

Elimination of $(a,\ b) \in A \times B$ – introduces no eigenvariables.

lemma *SigmaD1*: $(a,\ b)$: *Sigma A B* ==> a : A
by *blast*

lemma *SigmaD2*: $(a,\ b)$: *Sigma A B* ==> b : B a
by *blast*

lemma *SigmaE2*:
[| $(a,\ b)$: *Sigma A B*;
[| a:A; b:$B(a)$ |] ==> P
|] ==> P
by *blast*

lemma *Sigma-cong*:
$[\![A = B;\ !!x.\ x \in B \Longrightarrow C\ x = D\ x]\!]$
$\Longrightarrow$ (*SIGMA* x: A. $C\ x$) = (*SIGMA* x: B. $D\ x$)
by *auto*

lemma *Sigma-mono*: [| A <= C; !!x. x:A ==> $B\ x$ <= $D\ x$ |] ==> *Sigma A B* <= *Sigma C D*
by *blast*

lemma *Sigma-empty1* [*simp*]: *Sigma* {} B = {}
by *blast*

lemma *Sigma-empty2* [*simp*]: A <*> {} = {}
by *blast*

lemma *UNIV-Times-UNIV* [*simp*]: *UNIV* <*> *UNIV* = *UNIV*
by *auto*

lemma *Compl-Times-UNIV1* [*simp*]: − (*UNIV* <*> A) = *UNIV* <*> (−A)
by *auto*

lemma *Compl-Times-UNIV2* [*simp*]: − (A <*> *UNIV*) = (−A) <*> *UNIV*
by *auto*

lemma *mem-Sigma-iff* [*iff*]: ((a,b): *Sigma A B*) = (a:A & b:$B(a)$)
by *blast*

lemma *Times-subset-cancel2*: x:C ==> (A <*> C <= B <*> C) = (A <= B)
by *blast*

lemma *Times-eq-cancel2*: x:C ==> (A <*> C = B <*> C) = (A = B)

by (*blast elim*: *equalityE*)

lemma *SetCompr-Sigma-eq*:
Collect (*split* (%*x y*. *P x* & *Q x y*)) = (*SIGMA x*:*Collect P*. *Collect* (*Q x*))
by *blast*

lemma *Collect-split* [*simp*]: {(*a*,*b*). *P a* & *Q b*} = *Collect P* <*> *Collect Q*
by *blast*

lemma *UN-Times-distrib*:
(*UN* (*a*,*b*):(*A* <*> *B*). *E a* <*> *F b*) = (*UNION A E*) <*> (*UNION B F*)
— Suggested by Pierre Chartier
by *blast*

lemma *split-paired-Ball-Sigma* [*simp*,*noatp*]:
(*ALL z*: *Sigma A B*. *P z*) = (*ALL x*:*A*. *ALL y*: *B x*. *P*(*x*,*y*))
by *blast*

lemma *split-paired-Bex-Sigma* [*simp*,*noatp*]:
(*EX z*: *Sigma A B*. *P z*) = (*EX x*:*A*. *EX y*: *B x*. *P*(*x*,*y*))
by *blast*

lemma *Sigma-Un-distrib1*: (*SIGMA i*:*I Un J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) *Un* (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Un-distrib2*: (*SIGMA i*:*I*. *A*(*i*) *Un B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) *Un* (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Int-distrib1*: (*SIGMA i*:*I Int J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) *Int* (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Int-distrib2*: (*SIGMA i*:*I*. *A*(*i*) *Int B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) *Int* (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Diff-distrib1*: (*SIGMA i*:*I* − *J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) − (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Diff-distrib2*: (*SIGMA i*:*I*. *A*(*i*) − *B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) − (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Union*: *Sigma* (*Union X*) *B* = (*UN A*:*X*. *Sigma A B*)
by *blast*

Non-dependent versions are needed to avoid the need for higher-order match-

ing, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: $(A\ Un\ B) <*> C = (A <*> C)\ Un\ (B <*> C)$
by *blast*

lemma *Times-Int-distrib1*: $(A\ Int\ B) <*> C = (A <*> C)\ Int\ (B <*> C)$
by *blast*

lemma *Times-Diff-distrib1*: $(A - B) <*> C = (A <*> C) - (B <*> C)$
by *blast*

lemma *insert-times-insert*[*simp*]:
$insert\ a\ A \times insert\ b\ B =$
$insert\ (a,b)\ (A \times insert\ b\ B \cup insert\ a\ A \times B)$
by *blast*

13.4.2 Code generator setup

instance $*$ :: $(eq,\ eq)\ eq$ **..**

lemma [*code*]:
eq-class.eq $(x1::'a::eq,\ y1::'b::eq)\ (x2,\ y2) \longleftrightarrow x1 = x2 \wedge y1 = y2$ **by** (*simp add*: *eq*)

lemma *split-case-cert*:
assumes $CASE \equiv split\ f$
shows $CASE\ (a,\ b) \equiv f\ a\ b$
using *assms* **by** *simp*

setup ⟪
Code.add-case @{*thm split-case-cert*}
⟫

code-type $*$
(*SML* **infix** *2* $*$)
(*OCaml* **infix** *2* $*$)
(*Haskell* !((-),/ (-)))

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

code-const *eq-class.eq* :: $'a::eq \times 'b::eq \Rightarrow 'a \times 'b \Rightarrow bool$
(*Haskell* **infixl** *4* ==)

code-const *Pair*
(*SML* !((-),/ (-)))
(*OCaml* !((-),/ (-)))
(*Haskell* !((-),/ (-)))

code-const *fst* **and** *snd*

(*Haskell fst* **and** *snd*)

types-code
  *** ((- **/ -))
attach (*term-of*) ⟪
fun term-of-id-42 aF aT bF bT (*x*, *y*) = *HOLogic.pair-const aT bT* $ *aF x* $ *bF y*;
⟫
attach (*test*) ⟪
fun gen-id-42 aG aT bG bT i =
  let
    val (*x*, *t*) = *aG i*;
    val (*y*, *u*) = *bG i*
  in ((*x*, *y*), *fn* () => *HOLogic.pair-const aT bT* $ *t* () $ *u* ()) *end*;
⟫

consts-code
  Pair ((-,/ -))

setup ⟪

let

fun strip-abs-split 0 t = ([], *t*)
  | *strip-abs-split i* (*Abs* (*s*, *T*, *t*)) =
      let
        val s' = *Codegen.new-name t s*;
        val v = *Free* (*s'*, *T*)
      in apfst (*cons v*) (*strip-abs-split* (*i*−*1*) (*subst-bound* (*v*, *t*))) *end*
  | *strip-abs-split i* (*u as Const* (*split*, -) $ *t*) = (*case strip-abs-split* (*i*+*1*) *t of*
        (*v* :: *v'* :: *vs*, *u*) => (*HOLogic.mk-prod* (*v*, *v'*) :: *vs*, *u*)
      | - => ([], *u*))
  | *strip-abs-split i t* =
      strip-abs-split i (*Abs* (*x*, *hd* (*binder-types* (*fastype-of t*)), *t* $ *Bound 0*));

fun let-codegen thy defs dep thyname brack t gr = (*case strip-comb t of*
    (*t1 as Const* (*Let*, -), *t2* :: *t3* :: *ts*) =>
    let
      fun dest-let (*l as Const* (*Let*, -) $ *t* $ *u*) =
          (*case strip-abs-split 1 u of*
            ([*p*], *u'*) => *apfst* (*cons* (*p*, *t*)) (*dest-let u'*)
          | - => ([], *l*))
        | *dest-let t* = ([], *t*);
      fun mk-code (*l*, *r*) *gr* =
        let
          val (*pl*, *gr1*) = *Codegen.invoke-codegen thy defs dep thyname false l gr*;
          val (*pr*, *gr2*) = *Codegen.invoke-codegen thy defs dep thyname false r gr1*;
        in ((*pl*, *pr*), *gr2*) *end*
    in case dest-let (*t1* $ *t2* $ *t3*) *of*

```
      ([], -) => NONE
    | (ps, u) =>
        let
          val (qs, gr1) = fold-map mk-code ps gr;
         val (pu, gr2) = Codegen.invoke-codegen thy defs dep thyname false u gr1;
          val (pargs, gr3) = fold-map
            (Codegen.invoke-codegen thy defs dep thyname true) ts gr2
        in
          SOME (Codegen.mk-app brack
            (Pretty.blk (0, [Codegen.str let , Pretty.blk (0, List.concat
               (separate [Codegen.str ;, Pretty.brk 1] (map (fn (pl, pr) =>
                 [Pretty.block [Codegen.str val , pl, Codegen.str =,
                    Pretty.brk 1, pr]]) qs))),
              Pretty.brk 1, Codegen.str in , pu,
              Pretty.brk 1, Codegen.str end])) pargs, gr3)
        end
  end
| - => NONE);

fun split-codegen thy defs dep thyname brack t gr = (case strip-comb t of
    (t1 as Const (split, -), t2 :: ts) =>
      let
        val ([p], u) = strip-abs-split 1 (t1 $ t2);
        val (q, gr1) = Codegen.invoke-codegen thy defs dep thyname false p gr;
        val (pu, gr2) = Codegen.invoke-codegen thy defs dep thyname false u gr1;
        val (pargs, gr3) = fold-map
          (Codegen.invoke-codegen thy defs dep thyname true) ts gr2
      in
        SOME (Codegen.mk-app brack
          (Pretty.block [Codegen.str (fn , q, Codegen.str =>,
            Pretty.brk 1, pu, Codegen.str )]) pargs, gr2)
      end
  | - => NONE);

in

  Codegen.add-codegen let-codegen let-codegen
  #> Codegen.add-codegen split-codegen split-codegen

end
⟩⟩
```

13.5 Legacy bindings

ML ⟨⟨

```
val Collect-split = thm Collect-split;
val Compl-Times-UNIV1 = thm Compl-Times-UNIV1;
val Compl-Times-UNIV2 = thm Compl-Times-UNIV2;
val PairE = thm PairE;
```

val Pair-Rep-inject = thm Pair-Rep-inject;
val Pair-def = thm Pair-def;
val Pair-eq = @{thm prod.inject};
val Pair-fst-snd-eq = thm Pair-fst-snd-eq;
val ProdI = thm ProdI;
val SetCompr-Sigma-eq = thm SetCompr-Sigma-eq;
val SigmaD1 = thm SigmaD1;
val SigmaD2 = thm SigmaD2;
val SigmaE = thm SigmaE;
val SigmaE2 = thm SigmaE2;
val SigmaI = thm SigmaI;
val Sigma-Diff-distrib1 = thm Sigma-Diff-distrib1;
val Sigma-Diff-distrib2 = thm Sigma-Diff-distrib2;
val Sigma-Int-distrib1 = thm Sigma-Int-distrib1;
val Sigma-Int-distrib2 = thm Sigma-Int-distrib2;
val Sigma-Un-distrib1 = thm Sigma-Un-distrib1;
val Sigma-Un-distrib2 = thm Sigma-Un-distrib2;
val Sigma-Union = thm Sigma-Union;
val Sigma-def = thm Sigma-def;
val Sigma-empty1 = thm Sigma-empty1;
val Sigma-empty2 = thm Sigma-empty2;
val Sigma-mono = thm Sigma-mono;
val The-split = thm The-split;
val The-split-eq = thm The-split-eq;
val The-split-eq = thm The-split-eq;
val Times-Diff-distrib1 = thm Times-Diff-distrib1;
val Times-Int-distrib1 = thm Times-Int-distrib1;
val Times-Un-distrib1 = thm Times-Un-distrib1;
val Times-eq-cancel2 = thm Times-eq-cancel2;
val Times-subset-cancel2 = thm Times-subset-cancel2;
val UNIV-Times-UNIV = thm UNIV-Times-UNIV;
val UN-Times-distrib = thm UN-Times-distrib;
val Unity-def = thm Unity-def;
val cond-split-eta = thm cond-split-eta;
val fst-conv = thm fst-conv;
val fst-def = thm fst-def;
val fst-eqD = thm fst-eqD;
val inj-on-Abs-Prod = thm inj-on-Abs-Prod;
val mem-Sigma-iff = thm mem-Sigma-iff;
val mem-splitE = thm mem-splitE;
val mem-splitI = thm mem-splitI;
val mem-splitI2 = thm mem-splitI2;
val prod-eqI = thm prod-eqI;
val prod-fun = thm prod-fun;
val prod-fun-compose = thm prod-fun-compose;
val prod-fun-def = thm prod-fun-def;
val prod-fun-ident = thm prod-fun-ident;
val prod-fun-imageE = thm prod-fun-imageE;
val prod-fun-imageI = thm prod-fun-imageI;

val prod-induct = thm prod.induct;
val snd-conv = thm snd-conv;
val snd-def = thm snd-def;
val snd-eqD = thm snd-eqD;
val split = thm split;
val splitD = thm splitD;
val splitD' = thm splitD';
val splitE = thm splitE;
val splitE' = thm splitE';
val splitE2 = thm splitE2;
val splitI = thm splitI;
val splitI2 = thm splitI2;
val splitI2' = thm splitI2';
val split-beta = thm split-beta;
val split-conv = thm split-conv;
val split-def = thm split-def;
val split-eta = thm split-eta;
val split-eta-SetCompr = thm split-eta-SetCompr;
val split-eta-SetCompr2 = thm split-eta-SetCompr2;
val split-paired-All = thm split-paired-All;
val split-paired-Ball-Sigma = thm split-paired-Ball-Sigma;
val split-paired-Bex-Sigma = thm split-paired-Bex-Sigma;
val split-paired-Ex = thm split-paired-Ex;
val split-paired-The = thm split-paired-The;
val split-paired-all = thm split-paired-all;
val split-part = thm split-part;
val split-split = thm split-split;
val split-split-asm = thm split-split-asm;
val split-tupled-all = thms split-tupled-all;
val split-weak-cong = thm split-weak-cong;
val surj-pair = thm surj-pair;
val surjective-pairing = thm surjective-pairing;
val unit-abs-eta-conv = thm unit-abs-eta-conv;
val unit-all-eq1 = thm unit-all-eq1;
val unit-all-eq2 = thm unit-all-eq2;
val unit-eq = thm unit-eq;
⟩⟩

13.6 Further inductive packages

use *Tools/inductive-realizer.ML*
setup *InductiveRealizer.setup*

use *Tools/inductive-set-package.ML*
setup *InductiveSetPackage.setup*

use *Tools/datatype-realizer.ML*
setup *DatatypeRealizer.setup*

end

14 Datatype: Analogues of the Cartesian Product and Disjoint Sum for Datatypes

theory *Datatype*
imports *Nat Product-Type*
begin

typedef (*Node*)
(*'a*,*'b*) *node* = {*p*. *EX f x k*. *p* = (*f*::*nat*=>*'b*+*nat*, *x*::*'a*+*nat*) & *f k* = *Inr 0*}
— it is a subtype of (*nat*=>*'b*+*nat*) * (*'a*+*nat*)
by *auto*

Datatypes will be represented by sets of type *node*

types *'a item* = (*'a*, *unit*) *node set*
('*a*, '*b*) *dtree* = (*'a*, *'b*) *node set*

consts
Push :: [(*'b* + *nat*), *nat* => (*'b* + *nat*)] => (*nat* => (*'b* + *nat*))

Push-Node :: [(*'b* + *nat*), (*'a*, *'b*) *node*] => (*'a*, *'b*) *node*
ndepth :: (*'a*, *'b*) *node* => *nat*

Atom :: (*'a* + *nat*) => (*'a*, *'b*) *dtree*
Leaf :: *'a* => (*'a*, *'b*) *dtree*
Numb :: *nat* => (*'a*, *'b*) *dtree*
Scons :: [(*'a*, *'b*) *dtree*, (*'a*, *'b*) *dtree*] => (*'a*, *'b*) *dtree*
In0 :: (*'a*, *'b*) *dtree* => (*'a*, *'b*) *dtree*
In1 :: (*'a*, *'b*) *dtree* => (*'a*, *'b*) *dtree*
Lim :: (*'b* => (*'a*, *'b*) *dtree*) => (*'a*, *'b*) *dtree*

ntrunc :: [*nat*, (*'a*, *'b*) *dtree*] => (*'a*, *'b*) *dtree*

uprod :: [(*'a*, *'b*) *dtree set*, (*'a*, *'b*) *dtree set*]=> (*'a*, *'b*) *dtree set*
usum :: [(*'a*, *'b*) *dtree set*, (*'a*, *'b*) *dtree set*]=> (*'a*, *'b*) *dtree set*

Split :: [[(*'a*, *'b*) *dtree*, (*'a*, *'b*) *dtree*]=>*'c*, (*'a*, *'b*) *dtree*] => *'c*
Case :: [[(*'a*, *'b*) *dtree*]=>*'c*, [(*'a*, *'b*) *dtree*]=>*'c*, (*'a*, *'b*) *dtree*] => *'c*

dprod :: [((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*, ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*]
=> ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*
dsum :: [((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*, ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*]
=> ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*

defs

Push-Node-def: *Push-Node* == (%*n x*. *Abs-Node* (*apfst* (*Push n*) (*Rep-Node x*)))

Push-def: *Push* == (%*b h*. *nat-case b h*)

Atom-def: *Atom* == (%*x*. {*Abs-Node*((%*k*. *Inr 0*, *x*))})
Scons-def: *Scons M N* == (*Push-Node* (*Inr 1*) ‘ *M*) *Un* (*Push-Node* (*Inr* (*Suc 1*)) ‘ *N*)

Leaf-def: *Leaf* == *Atom o Inl*
Numb-def: *Numb* == *Atom o Inr*

In0-def: *In0*(*M*) == *Scons* (*Numb 0*) *M*
In1-def: *In1*(*M*) == *Scons* (*Numb 1*) *M*

Lim-def: *Lim f* == *Union* {*z*. *? x*. *z* = *Push-Node* (*Inl x*) ‘ (*f x*)}

ndepth-def: *ndepth*(*n*) == (%(*f*,*x*). *LEAST k*. *f k* = *Inr 0*) (*Rep-Node n*)
ntrunc-def: *ntrunc k N* == {*n*. *n*:*N* & *ndepth*(*n*)<*k*}

uprod-def: *uprod A B* == *UN x*:*A*. *UN y*:*B*. { *Scons x y* }
usum-def: *usum A B* == *In0*‘*A Un In1*‘*B*

Split-def: *Split c M* == *THE u*. *EX x y*. *M* = *Scons x y* & *u* = *c x y*

Case-def: *Case c d M* == *THE u*. (*EX x* . *M* = *In0*(*x*) & *u* = *c*(*x*))
| (*EX y* . *M* = *In1*(*y*) & *u* = *d*(*y*))

dprod-def: *dprod r s* == *UN* (*x*,*x*′):*r*. *UN* (*y*,*y*′):*s*. {(*Scons x y*, *Scons x*′ *y*′)}

dsum-def: *dsum r s* == (*UN* (*x*,*x*′):*r*. {(*In0*(*x*),*In0*(*x*′))}) *Un*
(*UN* (*y*,*y*′):*s*. {(*In1*(*y*),*In1*(*y*′))})

lemma *apfst-convE*:
[| *q* = *apfst f p*; !!*x y*. [| *p* = (*x*,*y*); *q* = (*f*(*x*),*y*) |] ==> *R* |] ==> *R*
by (*force simp add*: *apfst-def*)

lemma *Push-inject1*: *Push i f* = *Push j g* ==> *i*=*j*
apply (*simp add*: *Push-def expand-fun-eq*)
apply (*drule-tac x*=*0* **in** *spec*, *simp*)
done

lemma *Push-inject2*: *Push i f* = *Push j g* ==> *f*=*g*
apply (*auto simp add*: *Push-def expand-fun-eq*)
apply (*drule-tac x*=*Suc x* **in** *spec*, *simp*)
done

lemma *Push-inject*:
[| *Push i f* =*Push j g*; [| *i*=*j*; *f*=*g* |] ==> *P* |] ==> *P*
by (*blast dest*: *Push-inject1 Push-inject2*)

lemma *Push-neq-K0*: *Push* (*Inr* (*Suc k*)) *f* = (%*z*. *Inr 0*) ==> *P*
by (*auto simp add*: *Push-def expand-fun-eq split*: *nat.split-asm*)

lemmas *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*, *standard*]

lemma *Node-K0-I*: (%*k*. *Inr 0*, *a*) : *Node*
by (*simp add*: *Node-def*)

lemma *Node-Push-I*: *p*: *Node* ==> *apfst* (*Push i*) *p* : *Node*
apply (*simp add*: *Node-def Push-def*)
apply (*fast intro*!: *apfst-conv nat-case-Suc* [*THEN trans*])
done

14.1 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N* ≠ *Atom*(*a*)
apply (*simp add*: *Atom-def Scons-def Push-Node-def One-nat-def*)
apply (*blast intro*: *Node-K0-I Rep-Node* [*THEN Node-Push-I*]
dest!: *Abs-Node-inj*
elim!: *apfst-convE sym* [*THEN Push-neq-K0*])
done

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*, *standard*]

lemma *inj-Atom*: *inj*(*Atom*)
apply (*simp add*: *Atom-def*)
apply (*blast intro*!: *inj-onI Node-K0-I dest*!: *Abs-Node-inj*)
done
lemmas *Atom-inject* = *inj-Atom* [*THEN injD*, *standard*]

lemma *Atom-Atom-eq* [*iff*]: (*Atom*(*a*)=*Atom*(*b*)) = (*a*=*b*)
by (*blast dest*!: *Atom-inject*)

lemma *inj-Leaf*: *inj*(*Leaf*)
apply (*simp add*: *Leaf-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inl-inject*])
done

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*, *standard*]

lemma *inj-Numb*: *inj*(*Numb*)
apply (*simp add*: *Numb-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inr-inject*])
done

lemmas *Numb-inject* [*dest*!] = *inj-Numb* [*THEN injD*, *standard*]

lemma *Push-Node-inject*:
[| *Push-Node i m* =*Push-Node j n*; [| *i*=*j*; *m*=*n* |] ==> *P*
|] ==> *P*
apply (*simp add*: *Push-Node-def*)
apply (*erule Abs-Node-inj* [*THEN apfst-convE*])
apply (*rule Rep-Node* [*THEN Node-Push-I*])+
apply (*erule sym* [*THEN apfst-convE*])
apply (*blast intro*: *Rep-Node-inject* [*THEN iffD1*] *trans sym elim*!: *Push-inject*)
done

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M' N'* ==> *M*<=*M'*
apply (*simp add*: *Scons-def One-nat-def*)
apply (*blast dest*!: *Push-Node-inject*)
done

lemma *Scons-inject-lemma2*: *Scons M N <= Scons M' N' ==> N<=N'*
apply (*simp add*: *Scons-def One-nat-def*)
apply (*blast dest*!: *Push-Node-inject*)
done

lemma *Scons-inject1*: *Scons M N = Scons M' N' ==> M=M'*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma1*)
done

lemma *Scons-inject2*: *Scons M N = Scons M' N' ==> N=N'*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma2*)
done

lemma *Scons-inject*:
[| *Scons M N = Scons M' N'*; [| *M=M'*; *N=N'* |] ==> *P* |] ==> *P*
by (*iprover dest*: *Scons-inject1 Scons-inject2*)

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N = Scons M' N'*) = (*M=M' & N=N'*)
by (*blast elim*!: *Scons-inject*)

lemma *Scons-not-Leaf* [*iff*]: *Scons M N ≠ Leaf(a)*
by (*simp add*: *Leaf-def o-def Scons-not-Atom*)

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*, *standard*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N ≠ Numb(k)*
by (*simp add*: *Numb-def o-def Scons-not-Atom*)

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*, *standard*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf(a) ≠ Numb(k)*
by (*simp add*: *Leaf-def Numb-def*)

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*, *standard*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr* 0, *x*)) = 0
by (*simp add*: *ndepth-def Node-K0-I* [*THEN Abs-Node-inverse*] *Least-equality*)

lemma *ndepth-Push-Node-aux*:
nat-case (*Inr* (*Suc i*)) *f k* = *Inr* 0 −−> *Suc*(*LEAST x*. *f x* = *Inr* 0) <= *k*
apply (*induct-tac k*, *auto*)
apply (*erule Least-le*)
done

lemma *ndepth-Push-Node*:
ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
apply (*insert Rep-Node* [*of n*, *unfolded Node-def*])
apply (*auto simp add*: *ndepth-def Push-Node-def*
Rep-Node [*THEN Node-Push-I*, *THEN Abs-Node-inverse*])
apply (*rule Least-equality*)
apply (*auto simp add*: *Push-def ndepth-Push-Node-aux*)
apply (*erule LeastI*)
done

lemma *ntrunc-0* [*simp*]: *ntrunc 0 M* = {}
by (*simp add*: *ntrunc-def*)

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
by (*auto simp add*: *Atom-def ntrunc-def ndepth-K0*)

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
by (*simp add*: *Leaf-def o-def ntrunc-Atom*)

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
by (*simp add*: *Numb-def o-def ntrunc-Atom*)

lemma *ntrunc-Scons* [*simp*]:
ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)
by (*auto simp add*: *Scons-def ntrunc-def One-nat-def ndepth-Push-Node*)

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
apply (*simp add*: *In0-def*)
apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*)

M)
by (*simp add*: *In0-def*)

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
apply (*simp add*: *In1-def*)
apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
by (*simp add*: *In1-def*)

14.2 Set Constructions

lemma *uprodI* [*intro*!]: [| *M*:*A*; *N*:*B* |] ==> *Scons M N* : *uprod A B*
by (*simp add*: *uprod-def*)

lemma *uprodE* [*elim*!]:
[| *c* : *uprod A B*;
!!*x y*. [| *x*:*A*; *y*:*B*; *c* = *Scons x y* |] ==> *P*
|] ==> *P*
by (*auto simp add*: *uprod-def*)

lemma *uprodE2*: [| *Scons M N* : *uprod A B*; [| *M*:*A*; *N*:*B* |] ==> *P* |] ==> *P*
by (*auto simp add*: *uprod-def*)

lemma *usum-In0I* [*intro*]: *M*:*A* ==> *In0*(*M*) : *usum A B*
by (*simp add*: *usum-def*)

lemma *usum-In1I* [*intro*]: *N*:*B* ==> *In1*(*N*) : *usum A B*
by (*simp add*: *usum-def*)

lemma *usumE* [*elim*!]:
[| *u* : *usum A B*;
!!*x*. [| *x*:*A*; *u*=*In0*(*x*) |] ==> *P*;
!!*y*. [| *y*:*B*; *u*=*In1*(*y*) |] ==> *P*
|] ==> *P*
by (*auto simp add*: *usum-def*)

lemma *In0-not-In1* [*iff*]: *In0*(*M*) ≠ *In1*(*N*)

by (*auto simp add*: *In0-def In1-def One-nat-def*)

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*, *standard*]

lemma *In0-inject*: *In0*(*M*) = *In0*(*N*) ==> *M*=*N*
by (*simp add*: *In0-def*)

lemma *In1-inject*: *In1*(*M*) = *In1*(*N*) ==> *M*=*N*
by (*simp add*: *In1-def*)

lemma *In0-eq* [*iff*]: (*In0 M* = *In0 N*) = (*M*=*N*)
by (*blast dest*!: *In0-inject*)

lemma *In1-eq* [*iff*]: (*In1 M* = *In1 N*) = (*M*=*N*)
by (*blast dest*!: *In1-inject*)

lemma *inj-In0*: *inj In0*
by (*blast intro*!: *inj-onI*)

lemma *inj-In1*: *inj In1*
by (*blast intro*!: *inj-onI*)

lemma *Lim-inject*: *Lim f* = *Lim g* ==> *f* = *g*
apply (*simp add*: *Lim-def*)
apply (*rule ext*)
apply (*blast elim*!: *Push-Node-inject*)
done

lemma *ntrunc-subsetI*: *ntrunc k M* <= *M*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-subsetD*: (!!*k*. *ntrunc k M* <= *N*) ==> *M*<=*N*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-equality*: (!!*k*. *ntrunc k M* = *ntrunc k N*) ==> *M*=*N*
apply (*rule equalityI*)
apply (*rule-tac* [!] *ntrunc-subsetD*)
apply (*rule-tac* [!] *ntrunc-subsetI* [*THEN* [*2*] *subset-trans*], *auto*)
done

lemma *ntrunc-o-equality*:
[| !!*k*. (*ntrunc*(*k*) *o h1*) = (*ntrunc*(*k*) *o h2*) |] ==> *h1*=*h2*

apply (*rule ntrunc-equality* [*THEN ext*])
apply (*simp add*: *expand-fun-eq*)
done

lemma *uprod-mono*: [| *A*<=*A'*; *B*<=*B'* |] ==> *uprod A B* <= *uprod A' B'*
by (*simp add*: *uprod-def*, *blast*)

lemma *usum-mono*: [| *A*<=*A'*; *B*<=*B'* |] ==> *usum A B* <= *usum A' B'*
by (*simp add*: *usum-def*, *blast*)

lemma *Scons-mono*: [| *M*<=*M'*; *N*<=*N'* |] ==> *Scons M N* <= *Scons M' N'*
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-mono*: *M*<=*N* ==> *In0*(*M*) <= *In0*(*N*)
by (*simp add*: *In0-def subset-refl Scons-mono*)

lemma *In1-mono*: *M*<=*N* ==> *In1*(*M*) <= *In1*(*N*)
by (*simp add*: *In1-def subset-refl Scons-mono*)

lemma *Split* [*simp*]: *Split c* (*Scons M N*) = *c M N*
by (*simp add*: *Split-def*)

lemma *Case-In0* [*simp*]: *Case c d* (*In0 M*) = *c*(*M*)
by (*simp add*: *Case-def*)

lemma *Case-In1* [*simp*]: *Case c d* (*In1 N*) = *d*(*N*)
by (*simp add*: *Case-def*)

lemma *ntrunc-UN1*: *ntrunc k* (*UN x*. *f*(*x*)) = (*UN x*. *ntrunc k* (*f x*))
by (*simp add*: *ntrunc-def*, *blast*)

lemma *Scons-UN1-x*: *Scons* (*UN x*. *f x*) *M* = (*UN x*. *Scons* (*f x*) *M*)
by (*simp add*: *Scons-def*, *blast*)

lemma *Scons-UN1-y*: *Scons M* (*UN x*. *f x*) = (*UN x*. *Scons M* (*f x*))
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-UN1*: *In0*(*UN x*. *f*(*x*)) = (*UN x*. *In0*(*f*(*x*)))
by (*simp add*: *In0-def Scons-UN1-y*)

lemma *In1-UN1*: $In1(UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
by (*simp add*: *In1-def Scons-UN1-y*)

lemma *dprodI* [*intro*!]:
$[|\ (M,M'):r;\ \ (N,N'):s\ |] ==> (Scons\ M\ N,\ Scons\ M'\ N') : dprod\ r\ s$
by (*auto simp add*: *dprod-def*)

lemma *dprodE* [*elim*!]:
$[|\ c : dprod\ r\ s;$
$\quad !!x\ y\ x'\ y'.\ [|\ (x,x') : r;\ \ (y,y') : s;$
$\qquad\qquad c = (Scons\ x\ y,\ Scons\ x'\ y')\ |] ==> P$
$\ |] ==> P$
by (*auto simp add*: *dprod-def*)

lemma *dsum-In0I* [*intro*]: $(M,M'):r ==> (In0(M),\ In0(M')) : dsum\ r\ s$
by (*auto simp add*: *dsum-def*)

lemma *dsum-In1I* [*intro*]: $(N,N'):s ==> (In1(N),\ In1(N')) : dsum\ r\ s$
by (*auto simp add*: *dsum-def*)

lemma *dsumE* [*elim*!]:
$[|\ w : dsum\ r\ s;$
$\quad !!x\ x'.\ [|\ (x,x') : r;\ \ w = (In0(x),\ In0(x'))\ |] ==> P;$
$\quad !!y\ y'.\ [|\ (y,y') : s;\ \ w = (In1(y),\ In1(y'))\ |] ==> P$
$\ |] ==> P$
by (*auto simp add*: *dsum-def*)

lemma *dprod-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dprod\ r\ s <= dprod\ r'\ s'$
by *blast*

lemma *dsum-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dsum\ r\ s <= dsum\ r'\ s'$
by *blast*

lemma *dprod-Sigma*: $(dprod\ (A <*> B)\ (C <*> D)) <= (uprod\ A\ C) <*> (uprod\ B\ D)$

by *blast*

lemmas *dprod-subset-Sigma = subset-trans [OF dprod-mono dprod-Sigma, standard]*

lemma *dprod-subset-Sigma2*:
(*dprod* (*Sigma A B*) (*Sigma C D*)) <=
Sigma (*uprod A C*) (*Split* (%*x y*. *uprod* (*B x*) (*D y*)))
by *auto*

lemma *dsum-Sigma*: (*dsum* (*A* <*∗*> *B*) (*C* <*∗*> *D*)) <= (*usum A C*) <*∗*> (*usum B D*)
by *blast*

lemmas *dsum-subset-Sigma = subset-trans [OF dsum-mono dsum-Sigma, standard]*

hides popular names

hide (**open**) *type node item*
hide (**open**) *const Push Node Atom Leaf Numb Lim Split Case*

15 Datatypes

15.1 Representing sums

rep-datatype (*sum*) *Inl Inr*
proof −
fix *P*
fix *s* :: *'a* + *'b*
assume *x*: ⋀*x*::*'a*. *P* (*Inl x*) **and** *y*: ⋀*y*::*'b*. *P* (*Inr y*)
then show *P s* **by** (*auto intro*: *sumE* [*of s*])
qed *simp-all*

lemma *sum-case-KK*[*simp*]: *sum-case* (%*x*. *a*) (%*x*. *a*) = (%*x*. *a*)
by (*rule ext*) (*simp split*: *sum.split*)

lemma *surjective-sum*: *sum-case* (%*x*::*'a*. *f* (*Inl x*)) (%*y*::*'b*. *f* (*Inr y*)) *s* = *f*(*s*)
apply (*rule-tac s = s* **in** *sumE*)
apply (*erule ssubst*)
apply (*rule sum.cases*(*1*))
apply (*erule ssubst*)
apply (*rule sum.cases*(*2*))
done

lemma *sum-case-weak-cong*: *s = t ==> sum-case f g s = sum-case f g t*
— Prevents simplification of *f* and *g*: much faster.
by *simp*

lemma *sum-case-inject*:
sum-case f1 f2 = sum-case g1 g2 ==> (f1 = g1 ==> f2 = g2 ==> P) ==> P
proof −
assume *a*: *sum-case f1 f2 = sum-case g1 g2*
assume *r*: *f1 = g1 ==> f2 = g2 ==> P*
show *P*
apply (*rule r*)
apply (*rule ext*)
apply (*cut-tac x = Inl x* **in** *a* [*THEN fun-cong*], *simp*)
apply (*rule ext*)
apply (*cut-tac x = Inr x* **in** *a* [*THEN fun-cong*], *simp*)
done
qed

constdefs
Suml :: ('*a* => '*c*) => '*a* + '*b* => '*c*
Suml == (%*f*. *sum-case f undefined*)

Sumr :: ('*b* => '*c*) => '*a* + '*b* => '*c*
Sumr == *sum-case undefined*

lemma *Suml-inject*: *Suml f = Suml g ==> f = g*
by (*unfold Suml-def*) (*erule sum-case-inject*)

lemma *Sumr-inject*: *Sumr f = Sumr g ==> f = g*
by (*unfold Sumr-def*) (*erule sum-case-inject*)

primrec *Projl* :: '*a* + '*b* => '*a*
where *Projl-Inl*: *Projl* (*Inl x*) = *x*

primrec *Projr* :: '*a* + '*b* => '*b*
where *Projr-Inr*: *Projr* (*Inr x*) = *x*

hide (**open**) *const Suml Sumr Projl Projr*

end

16 Power: Exponentiation

theory *Power*
imports *Nat*
begin

class *power* =
fixes *power* :: '*a* ⇒ *nat* ⇒ '*a* (**infixr** ^ *80*)

16.1 Powers for Arbitrary Monoids

class *recpower* = *monoid-mult* + *power* +
  assumes *power-0* [*simp*]: *a ^ 0 = 1*
  assumes *power-Suc* [*simp*]: *a ^ Suc n = a * (a ^ n)*

lemma *power-0-Suc* [*simp*]: *(0::'a::{recpower,semiring-0}) ^ (Suc n) = 0*
  by *simp*

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*: *0^n = (if n=0 then 1 else (0::'a::{recpower,semiring-0}))*
  by (*induct n*) *simp-all*

lemma *power-one* [*simp*]: *1^n = (1::'a::recpower)*
  by (*induct n*) *simp-all*

lemma *power-one-right* [*simp*]: *(a::'a::recpower) ^ 1 = a*
  unfolding *One-nat-def* **by** *simp*

lemma *power-commutes*: *(a::'a::recpower) ^ n * a = a * a ^ n*
  by (*induct n*) (*simp-all add*: *mult-assoc*)

lemma *power-Suc2*: *(a::'a::recpower) ^ Suc n = a ^ n * a*
  by (*simp add*: *power-commutes*)

lemma *power-add*: *(a::'a::recpower) ^ (m+n) = (a^m) * (a^n)*
  by (*induct m*) (*simp-all add*: *mult-ac*)

lemma *power-mult*: *(a::'a::recpower) ^ (m*n) = (a^m) ^ n*
  by (*induct n*) (*simp-all add*: *power-add*)

lemma *power-mult-distrib*: *((a::'a::{recpower,comm-monoid-mult}) * b) ^ n = (a^n) * (b^n)*
  by (*induct n*) (*simp-all add*: *mult-ac*)

lemma *zero-less-power*[*simp*]:
    0 < (a::'a::{ordered-semidom,recpower}) ==> 0 < a^n
by (*induct n*) (*simp-all add*: *mult-pos-pos*)

lemma *zero-le-power*[*simp*]:
    0 ≤ (a::'a::{ordered-semidom,recpower}) ==> 0 ≤ a^n
by (*induct n*) (*simp-all add*: *mult-nonneg-nonneg*)

lemma *one-le-power*[*simp*]:
    1 ≤ (a::'a::{ordered-semidom,recpower}) ==> 1 ≤ a^n
apply (*induct n*)
apply *simp-all*
apply (*rule order-trans* [*OF - mult-mono* [*of 1 - 1*]])
apply (*simp-all add*: *order-trans* [*OF zero-le-one*])
done

lemma *gt1-imp-ge0*: *1 < a ==> 0 ≤* (*a*::*'a*::*ordered-semidom*)
by (*simp add*: *order-trans* [*OF zero-le-one order-less-imp-le*])

lemma *power-gt1-lemma*:
assumes *gt1*: *1 <* (*a*::*'a*::{*ordered-semidom*,*recpower*})
shows *1 < a * a^n*
proof −
have *1*1 < a*1* **using** *gt1* **by** *simp*
also have ... ≤ *a * a^n* **using** *gt1*
by (*simp only*: *mult-mono gt1-imp-ge0 one-le-power order-less-imp-le*
zero-le-one order-refl)
finally show *?thesis* **by** *simp*
qed

lemma *one-less-power*[*simp*]:
⟦*1 <* (*a*::*'a*::{*ordered-semidom*,*recpower*}); *0 < n*⟧ ⟹ *1 < a ^ n*
by (*cases n*, *simp-all add*: *power-gt1-lemma*)

lemma *power-gt1*:
1 < (*a*::*'a*::{*ordered-semidom*,*recpower*}) ==> *1 < a ^* (*Suc n*)
by (*simp add*: *power-gt1-lemma*)

lemma *power-le-imp-le-exp*:
assumes *gt1*: (*1*::*'a*::{*recpower*,*ordered-semidom*}) *< a*
shows !!*n*. *a^m ≤ a^n ==> m ≤ n*
proof (*induct m*)
case *0*
show *?case* **by** *simp*
next
case (*Suc m*)
show *?case*
proof (*cases n*)
case *0*
from *prems* **have** *a * a^m ≤ 1* **by** *simp*
with *gt1* **show** *?thesis*
by (*force simp only*: *power-gt1-lemma*
linorder-not-less [*symmetric*])
next
case (*Suc n*)
from *prems* **show** *?thesis*
by (*force dest*: *mult-left-le-imp-le*
simp add: *order-less-trans* [*OF zero-less-one gt1*])
qed
qed

Surely we can strengthen this? It holds for *0<a<1* too.

lemma *power-inject-exp* [*simp*]:
1 < (*a*::*'a*::{*ordered-semidom*,*recpower*}) ==> (*a^m = a^n*) = (*m=n*)

by (*force simp add*: *order-antisym power-le-imp-le-exp*)

Can relax the first premise to $(0::'a) < a$ in the case of the natural numbers.

lemma *power-less-imp-less-exp*:
$[|\ (1::'a::\{recpower,ordered\text{-}semidom\}) < a;\ a\hat{}m < a\hat{}n\ |] ==> m < n$
by (*simp add*: *order-less-le* [*of m n*] *order-less-le* [*of a^m a^n*]
power-le-imp-le-exp)

lemma *power-mono*:
$[|a \leq b;\ (0::'a::\{recpower,ordered\text{-}semidom\}) \leq a|] ==> a\hat{}n \leq b\hat{}n$
apply (*induct n*)
apply *simp-all*
apply (*auto intro*: *mult-mono order-trans* [*of 0 a b*])
done

lemma *power-strict-mono* [*rule-format*]:
$[|a < b;\ (0::'a::\{recpower,ordered\text{-}semidom\}) \leq a|]$
$==> 0 < n \longrightarrow a\hat{}n < b\hat{}n$
apply (*induct n*)
apply (*auto simp add*: *mult-strict-mono order-le-less-trans* [*of 0 a b*])
done

lemma *power-eq-0-iff* [*simp*]:
$(a\hat{}n = 0) \longleftrightarrow$
$(a = (0::'a::\{mult\text{-}zero,zero\text{-}neq\text{-}one,no\text{-}zero\text{-}divisors,recpower\})\ \&\ n\neq 0)$
apply (*induct n*)
apply (*auto simp add*: *no-zero-divisors*)
done

lemma *field-power-not-zero*:
$a \neq (0::'a::\{ring\text{-}1\text{-}no\text{-}zero\text{-}divisors,recpower\}) ==> a\hat{}n \neq 0$
by *force*

lemma *nonzero-power-inverse*:
fixes $a :: 'a::\{division\text{-}ring,recpower\}$
shows $a \neq 0 ==> inverse\ (a\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
apply (*induct n*)
apply (*auto simp add*: *nonzero-inverse-mult-distrib power-commutes*)
done

Perhaps these should be simprules.

lemma *power-inverse*:
fixes $a :: 'a::\{division\text{-}ring,division\text{-}by\text{-}zero,recpower\}$
shows $inverse\ (a\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
apply (*cases a = 0*)
apply (*simp add*: *power-0-left*)
apply (*simp add*: *nonzero-power-inverse*)

done

lemma *power-one-over*: *1 / (a::'a::{field,division-by-zero,recpower}) ^n =*
(1 / a) ^n
apply (*simp add*: *divide-inverse*)
apply (*rule power-inverse*)
done

lemma *nonzero-power-divide*:
b ≠ 0 ==> (a/b) ^ n = ((a::'a::{field,recpower}) ^ n) / (b ^ n)
by (*simp add*: *divide-inverse power-mult-distrib nonzero-power-inverse*)

lemma *power-divide*:
(a/b) ^ n = ((a::'a::{field,division-by-zero,recpower}) ^ n / b ^ n)
apply (*case-tac b=0*, *simp add*: *power-0-left*)
apply (*rule nonzero-power-divide*)
apply *assumption*
done

lemma *power-abs*: *abs(a ^ n) = abs(a::'a::{ordered-idom,recpower}) ^ n*
apply (*induct n*)
apply (*auto simp add*: *abs-mult*)
done

lemma *abs-power-minus* [*simp*]:
fixes *a*:: *'a::{ordered-idom,recpower}* **shows** *abs((−a) ^ n) = abs(a ^ n)*
by (*simp add*: *abs-minus-cancel power-abs*)

lemma *zero-less-power-abs-iff* [*simp*,*noatp*]:
(0 < (abs a)^n) = (a ≠ (0::'a::{ordered-idom,recpower}) | n=0)
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
show *?case* **by** (*auto simp add*: *prems zero-less-mult-iff*)
qed

lemma *zero-le-power-abs* [*simp*]:
(0::'a::{ordered-idom,recpower}) ≤ (abs a)^n
by (*rule zero-le-power* [*OF abs-ge-zero*])

lemma *power-minus*: *(−a) ^ n = (− 1)^n * (a::'a::{ring-1,recpower}) ^ n*
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **then show** *?case*
by (*simp del*: *power-Suc add*: *power-Suc2 mult-assoc*)
qed

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*:
[|(0::'a::{ordered-semidom,recpower}) < a; a < 1|]
==> a * a^n < a^n
apply (*induct n*)
apply (*auto simp add*: *mult-strict-left-mono*)
done

lemma *power-strict-decreasing*:
[|n < N; 0 < a; a < (1::'a::{ordered-semidom,recpower})|]
==> a^N < a^n
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-Suc-less less-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac a * a^m < 1 * a^n, simp*)
apply (*rule mult-strict-mono*)
apply (*auto simp add*: *order-less-imp-le*)
done

Proof resembles that of *power-strict-decreasing*

lemma *power-decreasing*:
[|n ≤ N; 0 ≤ a; a ≤ (1::'a::{ordered-semidom,recpower})|]
==> a^N ≤ a^n
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *le-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac a * a^m ≤ 1 * a^n, simp*)
apply (*rule mult-mono*)
apply *auto*
done

lemma *power-Suc-less-one*:
[| 0 < a; a < (1::'a::{ordered-semidom,recpower}) |] ==> a ^ Suc n < 1
apply (*insert power-strict-decreasing* [*of 0 Suc n a*], *simp*)
done

Proof again resembles that of *power-strict-decreasing*

lemma *power-increasing*:
[|n ≤ N; (1::'a::{ordered-semidom,recpower}) ≤ a|] ==> a^n ≤ a^N
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *le-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac 1 * a^n ≤ a * a^m, simp*)
apply (*rule mult-mono*)
apply (*auto simp add*: *order-trans* [*OF zero-le-one*])
done

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:
(*1*::*'a*::{*ordered-semidom*,*recpower*}) < *a* ==> *a* ^ *n* < *a* * *a* ^ *n*
apply (*induct n*)
apply (*auto simp add*: *mult-strict-left-mono order-less-trans* [*OF zero-less-one*])
done

lemma *power-strict-increasing*:
[|*n* < *N*; (*1*::*'a*::{*ordered-semidom*,*recpower*}) < *a*|] ==> *a* ^ *n* < *a* ^ *N*
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-less-power-Suc less-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac 1* * *a* ^ *n* < *a* * *a* ^ *m*, *simp*)
apply (*rule mult-strict-mono*)
apply (*auto simp add*: *order-less-trans* [*OF zero-less-one*] *order-less-imp-le*)
done

lemma *power-increasing-iff* [*simp*]:
1 < (*b*::*'a*::{*ordered-semidom*,*recpower*}) ==> (*b* ^ *x* ≤ *b* ^ *y*) = (*x* ≤ *y*)
by (*blast intro*: *power-le-imp-le-exp power-increasing order-less-imp-le*)

lemma *power-strict-increasing-iff* [*simp*]:
1 < (*b*::*'a*::{*ordered-semidom*,*recpower*}) ==> (*b* ^ *x* < *b* ^ *y*) = (*x* < *y*)
by (*blast intro*: *power-less-imp-less-exp power-strict-increasing*)

lemma *power-le-imp-le-base*:
assumes *le*: *a* ^ *Suc n* ≤ *b* ^ *Suc n*
and *ynonneg*: (*0*::*'a*::{*ordered-semidom*,*recpower*}) ≤ *b*
shows *a* ≤ *b*
proof (*rule ccontr*)
assume ~ *a* ≤ *b*
then have *b* < *a* **by** (*simp only*: *linorder-not-le*)
then have *b* ^ *Suc n* < *a* ^ *Suc n*
by (*simp only*: *prems power-strict-mono*)
from *le* **and** *this* **show** *False*
by (*simp add*: *linorder-not-less* [*symmetric*])
qed

lemma *power-less-imp-less-base*:
fixes *a b* :: *'a*::{*ordered-semidom*,*recpower*}
assumes *less*: *a* ^ *n* < *b* ^ *n*
assumes *nonneg*: *0* ≤ *b*
shows *a* < *b*
proof (*rule contrapos-pp* [*OF less*])
assume ~ *a* < *b*
hence *b* ≤ *a* **by** (*simp only*: *linorder-not-less*)
hence *b* ^ *n* ≤ *a* ^ *n* **using** *nonneg* **by** (*rule power-mono*)
thus ~ *a* ^ *n* < *b* ^ *n* **by** (*simp only*: *linorder-not-less*)

qed

lemma *power-inject-base*:
[| *a ^ Suc n = b ^ Suc n*; *0 ≤ a*; *0 ≤ b* |]
==> *a = (b::'a::{ordered-semidom,recpower})*
by (*blast intro*: *power-le-imp-le-base order-antisym order-eq-refl sym*)

lemma *power-eq-imp-eq-base*:
fixes *a b :: 'a::{ordered-semidom,recpower}*
shows ⟦*a ^ n = b ^ n*; *0 ≤ a*; *0 ≤ b*; *0 < n*⟧ ⟹ *a = b*
by (*cases n*, *simp-all del*: *power-Suc*, *rule power-inject-base*)

The divides relation

lemma *le-imp-power-dvd*:
fixes *a :: 'a::{comm-semiring-1,recpower}*
assumes *m ≤ n* **shows** *a^m dvd a^n*
proof
have *a^n = a^(m + (n − m))*
using ‹*m ≤ n*› **by** *simp*
also have *... = a^m * a^(n − m)*
by (*rule power-add*)
finally show *a^n = a^m * a^(n − m)* **.**
qed

lemma *power-le-dvd*:
fixes *a b :: 'a::{comm-semiring-1,recpower}*
shows *a^n dvd b* ⟹ *m ≤ n* ⟹ *a^m dvd b*
by (*rule dvd-trans* [*OF le-imp-power-dvd*])

lemma *dvd-power-same*:
(*x::'a::{comm-semiring-1,recpower}*) *dvd y* ⟹ *x^n dvd y^n*
by (*induct n*) (*auto simp add*: *mult-dvd-mono*)

lemma *dvd-power-le*:
(*x::'a::{comm-semiring-1,recpower}*) *dvd y* ⟹ *m >= n* ⟹ *x^n dvd y^m*
by(*rule power-le-dvd*[*OF dvd-power-same*])

lemma *dvd-power* [*simp*]:
n > 0 | (*x::'a::{comm-semiring-1,recpower}*) *= 1* ⟹ *x dvd x^n*
apply (*erule disjE*)
apply (*subgoal-tac x ^ n = x^(Suc (n − 1))*)
apply (*erule ssubst*)
apply (*subst power-Suc*)
apply *auto*
done

16.2 Exponentiation for the Natural Numbers

instantiation *nat* :: *recpower*
begin

primrec *power-nat* **where**
  $p \hat{} 0 = (1::nat)$
  | $p \hat{} (Suc\ n) = (p::nat) * (p \hat{} n)$

instance proof
  fix *z n* :: *nat*
  show $z\hat{}0 = 1$ **by** *simp*
  show $z\hat{}(Suc\ n) = z * (z\hat{}n)$ **by** *simp*
qed

declare *power-nat.simps* [*simp del*]

end

lemma *of-nat-power*:
  of-nat $(m \hat{} n) = (of\text{-}nat\ m::'a::\{semiring\text{-}1,recpower\}) \hat{} n$
by (*induct n*, *simp-all add*: *of-nat-mult*)

lemma *nat-one-le-power* [*simp*]: $Suc\ 0 \leq i \Longrightarrow Suc\ 0 \leq i\hat{}n$
by (*rule one-le-power* [*of i n*, *unfolded One-nat-def*])

lemma *nat-zero-less-power-iff* [*simp*]: $(x\hat{}n > 0) = (x > (0::nat) \mid n=0)$
by (*induct n*, *auto*)

lemma *nat-power-eq-Suc-0-iff* [*simp*]:
  $((x::nat)\hat{}m = Suc\ 0) = (m = 0 \mid x = Suc\ 0)$
by (*induct-tac m*, *auto*)

lemma *power-Suc-0*[*simp*]: $(Suc\ 0)\hat{}n = Suc\ 0$
by *simp*

Valid for the naturals, but what if $0<i<1$? Premises cannot be weakened: consider the case where $i = (0::'a)$, $m = (1::'a)$ and $n = (0::'a)$.

lemma *nat-power-less-imp-less*:
  assumes *nonneg*: $0 < (i::nat)$
  assumes *less*: $i\hat{}m < i\hat{}n$
  shows $m < n$
proof (*cases* $i = 1$)
  case *True* **with** *less power-one* [**where** $'a = nat$] **show** *?thesis* **by** *simp*
next
  case *False* **with** *nonneg* **have** $1 < i$ **by** *auto*
  from *power-strict-increasing-iff* [*OF this*] *less* **show** *?thesis* **..**
qed

lemma *power-diff*:

assumes *nz*: *a* ~= *0*
shows *n* <= *m* ==> (*a*::'*a*::{*recpower*, *field*}) ^ (*m*−*n*) = (*a*^*m*) / (*a*^*n*)
by (*induct m n rule*: *diff-induct*)
(*simp-all add*: *nonzero-mult-divide-cancel-left nz*)

end

17 Finite-Set: Finite sets

theory *Finite-Set*
imports *Nat Product-Type Power*
begin

17.1 Definition and basic properties

inductive *finite* :: '*a set* => *bool*
where
emptyI [*simp*, *intro*!]: *finite* {}
| *insertI* [*simp*, *intro*!]: *finite A* ==> *finite* (*insert a A*)

lemma *ex-new-if-finite*: — does not depend on def of finite at all
assumes ¬ *finite* (*UNIV* :: '*a set*) **and** *finite A*
shows ∃ *a*::'*a*. *a* ∉ *A*
proof −
from *assms* **have** *A* ≠ *UNIV* **by** *blast*
thus *?thesis* **by** *blast*
qed

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
finite F ==>
P {} ==> (!!*x F*. *finite F* ==> *x* ∉ *F* ==> *P F* ==> *P* (*insert x F*)) ==>
P F
— Discharging *x* ∉ *F* entails extra work.
proof −
assume *P* {} **and**
insert: !!*x F*. *finite F* ==> *x* ∉ *F* ==> *P F* ==> *P* (*insert x F*)
assume *finite F*
thus *P F*
proof *induct*
show *P* {} **by** *fact*
fix *x F* **assume** *F*: *finite F* **and** *P*: *P F*
show *P* (*insert x F*)
proof *cases*
assume *x* ∈ *F*
hence *insert x F* = *F* **by** (*rule insert-absorb*)
with *P* **show** *?thesis* **by** (*simp only*:)
next
assume *x* ∉ *F*

```
    from F this P show ?thesis by (rule insert)
   qed
  qed
qed

lemma finite-ne-induct[case-names singleton insert, consumes 2]:
assumes fin: finite F shows F ≠ {} ⟹
⟦ ⋀x. P{x};
  ⋀x F. ⟦ finite F; F ≠ {}; x ∉ F; P F ⟧ ⟹ P (insert x F) ⟧
⟹ P F
using fin
proof induct
  case empty thus ?case by simp
next
  case (insert x F)
  show ?case
  proof cases
    assume F = {}
    thus ?thesis using ‹P {x}› by simp
  next
    assume F ≠ {}
    thus ?thesis using insert by blast
  qed
qed

lemma finite-subset-induct [consumes 2, case-names empty insert]:
  assumes finite F and F ⊆ A
    and empty: P {}
    and insert: !!a F. finite F ==> a ∈ A ==> a ∉ F ==> P F ==> P (insert
a F)
  shows P F
proof −
  from ‹finite F› and ‹F ⊆ A›
  show ?thesis
  proof induct
    show P {} by fact
  next
    fix x F
    assume finite F and x ∉ F and
      P: F ⊆ A ==> P F and i: insert x F ⊆ A
    show P (insert x F)
    proof (rule insert)
      from i show x ∈ A by blast
      from i have F ⊆ A by blast
      with P show P F .
      show finite F by fact
      show x ∉ F by fact
    qed
  qed
```

qed

A finite choice principle. Does not need the SOME choice operator.

lemma *finite-set-choice*:
finite A $\Longrightarrow$ *ALL x:A. (EX y. P x y)* $\Longrightarrow$ *EX f. ALL x:A. P x (f x)*
proof (*induct set*: *finite*)
case *empty* **thus** *?case* **by** *simp*
next
case (*insert a A*)
then obtain *f b* **where** *f*: *ALL x:A. P x (f x)* **and** *ab*: *P a b* **by** *auto*
show *?case* (**is** *EX f. ?P f*)
proof
show *?P*(%*x. if x = a then b else f x*) **using** *f ab* **by** *auto*
qed
qed

Finite sets are the images of initial segments of natural numbers:

lemma *finite-imp-nat-seg-image-inj-on*:
assumes *fin*: *finite A*
shows $\exists$ (*n*::*nat*) *f. A = f ‘ {i. i<n} & inj-on f {i. i<n}*
using *fin*
proof *induct*
case *empty*
show *?case*
proof show $\exists f.$ *{} = f ‘ {i*::*nat. i < 0} & inj-on f {i. i<0}* **by** *simp*
qed
next
case (*insert a A*)
have *notinA*: *a* $\notin$ *A* **by** *fact*
from *insert.hyps* **obtain** *n f*
where *A = f ‘ {i*::*nat. i < n} inj-on f {i. i < n}* **by** *blast*
hence *insert a A = f*(*n*:=*a*) *‘ {i. i < Suc n}*
inj-on (*f*(*n*:=*a*)) *{i. i < Suc n}* **using** *notinA*
by (*auto simp add*: *image-def Ball-def inj-on-def less-Suc-eq*)
thus *?case* **by** *blast*
qed

lemma *nat-seg-image-imp-finite*:
!!*f A. A = f ‘ {i*::*nat. i<n}* $\Longrightarrow$ *finite A*
proof (*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
let *?B = f ‘ {i. i < n}*
have *finB*: *finite ?B* **by**(*rule Suc.hyps*[*OF refl*])
show *?case*
proof *cases*
assume $\exists$ *k<n. f n = f k*
hence *A = ?B* **using** *Suc.prems* **by**(*auto simp*:*less-Suc-eq*)

```
    thus ?thesis using finB by simp
  next
    assume ¬(∃ k<n. f n = f k)
    hence A = insert (f n) ?B using Suc.prems by(auto simp:less-Suc-eq)
    thus ?thesis using finB by simp
  qed
qed

lemma finite-conv-nat-seg-image:
  finite A = (∃ (n::nat) f. A = f ` {i::nat. i<n})
by(blast intro: nat-seg-image-imp-finite dest: finite-imp-nat-seg-image-inj-on)

lemma finite-Collect-less-nat[iff]: finite{n::nat. n<k}
by(fastsimp simp: finite-conv-nat-seg-image)
```

17.1.1 Finiteness and set theoretic constructions

```
lemma finite-UnI: finite F ==> finite G ==> finite (F Un G)
by (induct set: finite) simp-all

lemma finite-subset: A ⊆ B ==> finite B ==> finite A
  — Every subset of a finite set is finite.
proof −
  assume finite B
  thus !!A. A ⊆ B ==> finite A
  proof induct
    case empty
    thus ?case by simp
  next
    case (insert x F A)
    have A: A ⊆ insert x F and r: A − {x} ⊆ F ==> finite (A − {x}) by fact+
    show finite A
    proof cases
      assume x: x ∈ A
      with A have A − {x} ⊆ F by (simp add: subset-insert-iff)
      with r have finite (A − {x}) .
      hence finite (insert x (A − {x})) ..
      also have insert x (A − {x}) = A using x by (rule insert-Diff)
      finally show ?thesis .
    next
      show A ⊆ F ==> ?thesis by fact
      assume x ∉ A
      with A show A ⊆ F by (simp add: subset-insert-iff)
    qed
  qed
qed

lemma finite-Un [iff]: finite (F Un G) = (finite F & finite G)
by (blast intro: finite-subset [of - X Un Y, standard] finite-UnI)
```

lemma *finite-Collect-disjI*[*simp*]:
finite{*x*. *P x* | *Q x*} = (*finite*{*x*. *P x*} & *finite*{*x*. *Q x*})
by(*simp add*:*Collect-disj-eq*)

lemma *finite-Int* [*simp*, *intro*]: *finite F* | *finite G* ==> *finite* (*F Int G*)
— The converse obviously fails.
by (*blast intro*: *finite-subset*)

lemma *finite-Collect-conjI* [*simp*, *intro*]:
finite{*x*. *P x*} | *finite*{*x*. *Q x*} ==> *finite*{*x*. *P x* & *Q x*}
— The converse obviously fails.
by(*simp add*:*Collect-conj-eq*)

lemma *finite-Collect-le-nat*[*iff*]: *finite*{*n*::*nat*. *n*<=*k*}
by(*simp add*: *le-eq-less-or-eq*)

lemma *finite-insert* [*simp*]: *finite* (*insert a A*) = *finite A*
apply (*subst insert-is-Un*)
apply (*simp only*: *finite-Un*, *blast*)
done

lemma *finite-Union*[*simp*, *intro*]:
⟦ *finite A*; !!*M*. *M* ∈ *A* ⟹ *finite M* ⟧ ⟹ *finite*(⋃ *A*)
by (*induct rule*:*finite-induct*) *simp-all*

lemma *finite-empty-induct*:
assumes *finite A*
and *P A*
and !!*a A*. *finite A* ==> *a*:*A* ==> *P A* ==> *P* (*A* − {*a*})
shows *P* {}
proof −
have *P* (*A* − *A*)
proof −
{
fix *c b* :: *'a set*
assume *c*: *finite c* **and** *b*: *finite b*
and *P1*: *P b* **and** *P2*: !!*x y*. *finite y* ==> *x* ∈ *y* ==> *P y* ==> *P* (*y* − {*x*})
have *c* ⊆ *b* ==> *P* (*b* − *c*)
using *c*
proof *induct*
case *empty*
from *P1* **show** *?case* **by** *simp*
next
case (*insert x F*)
have *P* (*b* − *F* − {*x*})
proof (*rule P2*)
from - *b* **show** *finite* (*b* − *F*) **by** (*rule finite-subset*) *blast*

```
        from insert show x ∈ b − F by simp
        from insert show P (b − F) by simp
      qed
      also have b − F − {x} = b − insert x F by (rule Diff-insert [symmetric])
      finally show ?case .
    qed
  }
  then show ?thesis by this (simp-all add: assms)
 qed
 then show ?thesis by simp
qed

lemma finite-Diff [simp]: finite A ==> finite (A − B)
by (rule Diff-subset [THEN finite-subset])

lemma finite-Diff2 [simp]:
 assumes finite B shows finite (A − B) = finite A
proof −
 have finite A ⟷ finite((A−B) Un (A Int B)) by(simp add: Un-Diff-Int)
 also have ... ⟷ finite(A−B) using ‹finite B› by(simp)
 finally show ?thesis ..
qed

lemma finite-compl[simp]:
 finite(A::'a set) ⟹ finite(−A) = finite(UNIV::'a set)
by(simp add:Compl-eq-Diff-UNIV)

lemma finite-Collect-not[simp]:
 finite{x::'a. P x} ⟹ finite{x. ~P x} = finite(UNIV::'a set)
by(simp add:Collect-neg-eq)

lemma finite-Diff-insert [iff]: finite (A − insert a B) = finite (A − B)
 apply (subst Diff-insert)
 apply (case-tac a : A − B)
  apply (rule finite-insert [symmetric, THEN trans])
  apply (subst insert-Diff, simp-all)
 done
```

Image and Inverse Image over Finite Sets

```
lemma finite-imageI[simp]: finite F ==> finite (h ‘ F)
 — The image of a finite set is finite.
 by (induct set: finite) simp-all

lemma finite-surj: finite A ==> B <= f ‘ A ==> finite B
 apply (frule finite-imageI)
 apply (erule finite-subset, assumption)
 done

lemma finite-range-imageI:
```

finite (*range g*) ==> *finite* (*range* (%*x*. *f* (*g x*)))
apply (*drule finite-imageI*, *simp add*: *range-composition*)
done

lemma *finite-imageD*: *finite* (*f*‘*A*) ==> *inj-on f A* ==> *finite A*
proof −
have *aux*: !!*A*. *finite* (*A* − {}) = *finite A* **by** *simp*
fix *B* :: *'a set*
assume *finite B*
thus !!*A*. *f*‘*A* = *B* ==> *inj-on f A* ==> *finite A*
apply *induct*
apply *simp*
apply (*subgoal-tac EX y*:*A*. *f y* = *x* & *F* = *f* ‘ (*A* − {*y*}))
apply *clarify*
apply (*simp* (*no-asm-use*) *add*: *inj-on-def*)
apply (*blast dest*!: *aux* [*THEN iffD1*], *atomize*)
apply (*erule-tac V* = *ALL A*. *?PP* (*A*) **in** *thin-rl*)
apply (*frule subsetD* [*OF equalityD2 insertI1*], *clarify*)
apply (*rule-tac x* = *xa* **in** *bexI*)
apply (*simp-all add*: *inj-on-image-set-diff*)
done
qed (*rule refl*)

lemma *inj-vimage-singleton*: *inj f* ==> *f*−‘{*a*} ⊆ {*THE x*. *f x* = *a*}
— The inverse image of a singleton under an injective function is included in a singleton.
apply (*auto simp add*: *inj-on-def*)
apply (*blast intro*: *the-equality* [*symmetric*])
done

lemma *finite-vimageI*: [|*finite F*; *inj h*|] ==> *finite* (*h* −‘ *F*)
— The inverse image of a finite set under an injective function is finite.
apply (*induct set*: *finite*)
apply *simp-all*
apply (*subst vimage-insert*)
apply (*simp add*: *finite-Un finite-subset* [*OF inj-vimage-singleton*])
done

The finite UNION of finite sets

lemma *finite-UN-I*: *finite A* ==> (!!*a*. *a*:*A* ==> *finite* (*B a*)) ==> *finite* (*UN a*:*A*. *B a*)
by (*induct set*: *finite*) *simp-all*

Strengthen RHS to (∀ *x*∈*A*. *finite* (*B x*)) ∧ *finite* {*x* ∈ *A*. *B x* ≠ {}}?
We'd need to prove *finite C* ⟹ ∀ *A B*. *UNION A B* ⊆ *C* ⟶ *finite* {*x* ∈ *A*. *B x* ≠ {}} by induction.

lemma *finite-UN* [*simp*]:
finite A ==> *finite* (*UNION A B*) = (*ALL x*:*A*. *finite* (*B x*))

by (*blast intro*: *finite-UN-I finite-subset*)

lemma *finite-Collect-bex*[*simp*]: *finite A* ⟹
finite{*x. EX y*:*A. Q x y*} = (*ALL y*:*A. finite*{*x. Q x y*})
apply(*subgoal-tac* {*x. EX y*:*A. Q x y*} = *UNION A* (%*y.* {*x. Q x y*}))
apply *auto*
done

lemma *finite-Collect-bounded-ex*[*simp*]: *finite*{*y. P y*} ⟹
finite{*x. EX y. P y* & *Q x y*} = (*ALL y. P y* ⟶ *finite*{*x. Q x y*})
apply(*subgoal-tac* {*x. EX y. P y* & *Q x y*} = *UNION* {*y. P y*} (%*y.* {*x. Q x y*}))
apply *auto*
done

lemma *finite-Plus*: [| *finite A*; *finite B* |] ==> *finite* (*A* <+> *B*)
by (*simp add*: *Plus-def*)

Sigma of finite sets

lemma *finite-SigmaI* [*simp*]:
finite A ==> (!!*a. a*:*A* ==> *finite* (*B a*)) ==> *finite* (*SIGMA a*:*A. B a*)
by (*unfold Sigma-def*) (*blast intro*!: *finite-UN-I*)

lemma *finite-cartesian-product*: [| *finite A*; *finite B* |] ==>
finite (*A* <*> *B*)
by (*rule finite-SigmaI*)

lemma *finite-Prod-UNIV*:
finite (*UNIV*::'*a set*) ==> *finite* (*UNIV*::'*b set*) ==> *finite* (*UNIV*::('*a* * '*b*) *set*)
apply (*subgoal-tac* (*UNIV*:: ('*a* * '*b*) *set*) = *Sigma UNIV* (%*x. UNIV*))
apply (*erule ssubst*)
apply (*erule finite-SigmaI*, *auto*)
done

lemma *finite-cartesian-productD1*:
[| *finite* (*A* <*> *B*); *B* ≠ {} |] ==> *finite A*
apply (*auto simp add*: *finite-conv-nat-seg-image*)
apply (*drule-tac x*=*n* **in** *spec*)
apply (*drule-tac x*=*fst o f* **in** *spec*)
apply (*auto simp add*: *o-def*)
prefer *2* **apply** (*force dest*!: *equalityD2*)
apply (*drule equalityD1*)
apply (*rename-tac y x*)
apply (*subgoal-tac* ∃*k. k*<*n* & *f k* = (*x*,*y*))
prefer *2* **apply** *force*
apply *clarify*
apply (*rule-tac x*=*k* **in** *image-eqI*, *auto*)
done

lemma *finite-cartesian-productD2*:
[| *finite* (*A* <*>* *B*); *A* ≠ {} |] ==> *finite B*
apply (*auto simp add*: *finite-conv-nat-seg-image*)
apply (*drule-tac x*=*n* **in** *spec*)
apply (*drule-tac x*=*snd o f* **in** *spec*)
apply (*auto simp add*: *o-def*)
prefer *2* **apply** (*force dest*!: *equalityD2*)
apply (*drule equalityD1*)
apply (*rename-tac x y*)
apply (*subgoal-tac* ∃*k*. *k*<*n* & *f k* = (*x*,*y*))
prefer *2* **apply** *force*
apply *clarify*
apply (*rule-tac x*=*k* **in** *image-eqI*, *auto*)
done

The powerset of a finite set

lemma *finite-Pow-iff* [*iff*]: *finite* (*Pow A*) = *finite A*
proof
assume *finite* (*Pow A*)
with - **have** *finite* ((%*x*. {*x*}) ‘ *A*) **by** (*rule finite-subset*) *blast*
thus *finite A* **by** (*rule finite-imageD* [*unfolded inj-on-def*]) *simp*
next
assume *finite A*
thus *finite* (*Pow A*)
by *induct* (*simp-all add*: *finite-UnI finite-imageI Pow-insert*)
qed

lemma *finite-Collect-subsets*[*simp*,*intro*]: *finite A* ⟹ *finite*{*B*. *B* ⊆ *A*}
by(*simp add*: *Pow-def*[*symmetric*])

lemma *finite-UnionD*: *finite*(⋃ *A*) ⟹ *finite A*
by(*blast intro*: *finite-subset*[*OF subset-Pow-Union*])

17.2 Class *finite*

setup ⟪ *Sign.add-path finite* ⟫ — FIXME: name tweaking
class *finite* =
assumes *finite-UNIV*: *finite* (*UNIV* :: *'a set*)
setup ⟪ *Sign.parent-path* ⟫
hide *const finite*

context *finite*
begin

lemma *finite* [*simp*]: *finite* (*A* :: *'a set*)
by (*rule subset-UNIV finite-UNIV finite-subset*)+

end

lemma *UNIV-unit* [*noatp*]:
UNIV = {()} **by** *auto*

instance *unit* :: *finite*
by *default* (*simp add*: *UNIV-unit*)

lemma *UNIV-bool* [*noatp*]:
UNIV = {*False*, *True*} **by** *auto*

instance *bool* :: *finite*
by *default* (*simp add*: *UNIV-bool*)

instance ∗ :: (*finite*, *finite*) *finite*
by *default* (*simp only*: *UNIV-Times-UNIV* [*symmetric*] *finite-cartesian-product finite*)

lemma *inj-graph*: *inj* (%*f*. {(*x*, *y*). *y* = *f x*})
by (*rule inj-onI*, *auto simp add*: *expand-set-eq expand-fun-eq*)

instance *fun* :: (*finite*, *finite*) *finite*
proof
show *finite* (*UNIV* :: ('*a* => '*b*) *set*)
proof (*rule finite-imageD*)
let *?graph* = %*f*::'*a* => '*b*. {(*x*, *y*). *y* = *f x*}
have *range ?graph* ⊆ *Pow UNIV* **by** *simp*
moreover have *finite* (*Pow* (*UNIV* :: ('*a* ∗ '*b*) *set*))
by (*simp only*: *finite-Pow-iff finite*)
ultimately show *finite* (*range ?graph*)
by (*rule finite-subset*)
show *inj ?graph* **by** (*rule inj-graph*)
qed
qed

instance + :: (*finite*, *finite*) *finite*
by *default* (*simp only*: *UNIV-Plus-UNIV* [*symmetric*] *finite-Plus finite*)

17.3 A fold functional for finite sets

The intended behaviour is *fold f z* {x_1, ..., x_n} = *f* x_1 (... (*f* x_n *z*)...) if *f* is "left-commutative":

locale *fun-left-comm* =
fixes *f* :: '*a* ⇒ '*b* ⇒ '*b*
assumes *fun-left-comm*: *f x* (*f y z*) = *f y* (*f x z*)
begin

On a functional level it looks much nicer:

lemma *fun-comp-comm*: *f x* ∘ *f y* = *f y* ∘ *f x*

by (*simp add*: *fun-left-comm expand-fun-eq*)

end

inductive *fold-graph* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool$
for $f :: 'a \Rightarrow 'b \Rightarrow 'b$ **and** $z :: 'b$ **where**
emptyI [*intro*]: *fold-graph f z* {} *z* |
insertI [*intro*]: $x \notin A \Longrightarrow$ *fold-graph f z A y*
$\Longrightarrow$ *fold-graph f z* (*insert x A*) (*f x y*)

inductive-cases *empty-fold-graphE* [*elim*!]: *fold-graph f z* {} *x*

definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b$ **where**
[*code del*]: *fold f z A* = (*THE y*. *fold-graph f z A y*)

A tempting alternative for the definiens is *if finite A then THE y*. *fold-graph f z A y else e*. It allows the removal of finiteness assumptions from the theorems *fold-comm*, *fold-reindex* and *fold-distrib*. The proofs become ugly. It is not worth the effort. (???)

lemma *Diff1-fold-graph*:
fold-graph f z (*A* − {*x*}) *y* $\Longrightarrow$ $x \in A \Longrightarrow$ *fold-graph f z A* (*f x y*)
by (*erule insert-Diff* [*THEN subst*], *rule fold-graph.intros*, *auto*)

lemma *fold-graph-imp-finite*: *fold-graph f z A x* $\Longrightarrow$ *finite A*
by (*induct set*: *fold-graph*) *auto*

lemma *finite-imp-fold-graph*: *finite A* $\Longrightarrow \exists x$. *fold-graph f z A x*
by (*induct set*: *finite*) *auto*

17.3.1 From *fold-graph* to *fold*

lemma *image-less-Suc*: *h* ‘ {*i*. *i* < *Suc m*} = *insert* (*h m*) (*h* ‘ {*i*. *i* < *m*})
by (*auto simp add*: *less-Suc-eq*)

lemma *insert-image-inj-on-eq*:
[|*insert* (*h m*) *A* = *h* ‘ {*i*. *i* < *Suc m*}; *h m* ∉ *A*;
inj-on h {*i*. *i* < *Suc m*}|]
==> *A* = *h* ‘ {*i*. *i* < *m*}
apply (*auto simp add*: *image-less-Suc inj-on-def*)
apply (*blast intro*: *less-trans*)
done

lemma *insert-inj-onE*:
assumes *aA*: *insert a A* = *h*‘{*i*::*nat*. *i*<*n*} **and** *anot*: *a* ∉ *A*
and *inj-on*: *inj-on h* {*i*::*nat*. *i*<*n*}
shows ∃*hm m*. *inj-on hm* {*i*::*nat*. *i*<*m*} & *A* = *hm* ‘ {*i*. *i*<*m*} & *m* < *n*
proof (*cases n*)
case *0* **thus** *?thesis* **using** *aA* **by** *auto*
next

```
  case (Suc m)
  have nSuc: n = Suc m by fact
  have mlessn: m<n by (simp add: nSuc)
   from aA obtain k where hkeq: h k = a and klessn: k<n by (blast elim!:
equalityE)
  let ?hm = Fun.swap k m h
  have inj-hm: inj-on ?hm {i. i < n} using klessn mlessn
    by (simp add: inj-on-swap-iff inj-on)
  show ?thesis
  proof (intro exI conjI)
    show inj-on ?hm {i. i < m} using inj-hm
      by (auto simp add: nSuc less-Suc-eq intro: subset-inj-on)
    show m<n by (rule mlessn)
    show A = ?hm ' {i. i < m}
    proof (rule insert-image-inj-on-eq)
      show inj-on (Fun.swap k m h) {i. i < Suc m} using inj-hm nSuc by simp
      show ?hm m ∉ A by (simp add: swap-def hkeq anot)
      show insert (?hm m) A = ?hm ' {i. i < Suc m}
        using aA hkeq nSuc klessn
        by (auto simp add: swap-def image-less-Suc fun-upd-image
                  less-Suc-eq inj-on-image-set-diff [OF inj-on])
    qed
  qed
qed

context fun-left-comm
begin

lemma fold-graph-determ-aux:
  A = h'{i::nat. i<n} ⟹ inj-on h {i. i<n}
   ⟹ fold-graph f z A x ⟹ fold-graph f z A x'
   ⟹ x' = x
proof (induct n arbitrary: A x x' h rule: less-induct)
  case (less n)
  have IH: ⋀m h A x x'. m < n ⟹ A = h ' {i. i<m}
     ⟹ inj-on h {i. i<m} ⟹ fold-graph f z A x
     ⟹ fold-graph f z A x' ⟹ x' = x by fact
  have Afoldx: fold-graph f z A x and Afoldx': fold-graph f z A x'
    and A: A = h'{i. i<n} and injh: inj-on h {i. i<n} by fact+
  show ?case
  proof (rule fold-graph.cases [OF Afoldx])
    assume A = {} and x = z
    with Afoldx' show x' = x by auto
  next
    fix B b u
    assume AbB: A = insert b B and x: x = f b u
      and notinB: b ∉ B and Bu: fold-graph f z B u
    show x'=x
    proof (rule fold-graph.cases [OF Afoldx'])
```

```
    assume A = {} and x' = z
    with AbB show x' = x by blast
  next
    fix C c v
    assume AcC: A = insert c C and x': x' = f c v
      and notinC: c ∉ C and Cv: fold-graph f z C v
    from A AbB have Beq: insert b B = h`{i. i<n} by simp
    from insert-inj-onE [OF Beq notinB injh]
    obtain hB mB where inj-onB: inj-on hB {i. i < mB}
      and Beq: B = hB ` {i. i < mB} and lessB: mB < n by auto
    from A AcC have Ceq: insert c C = h`{i. i<n} by simp
    from insert-inj-onE [OF Ceq notinC injh]
    obtain hC mC where inj-onC: inj-on hC {i. i < mC}
      and Ceq: C = hC ` {i. i < mC} and lessC: mC < n by auto
    show x'=x
    proof cases
      assume b=c
      then moreover have B = C using AbB AcC notinB notinC by auto
      ultimately show ?thesis  using Bu Cv x x' IH [OF lessC Ceq inj-onC]
        by auto
    next
      assume diff: b ≠ c
      let ?D = B - {c}
      have B: B = insert c ?D and C: C = insert b ?D
        using AbB AcC notinB notinC diff by(blast elim!:equalityE)+
      have finite A by(rule fold-graph-imp-finite [OF Afoldx])
      with AbB have finite ?D by simp
      then obtain d where Dfoldd: fold-graph f z ?D d
        using finite-imp-fold-graph by iprover
      moreover have cinB: c ∈ B using B by auto
      ultimately have fold-graph f z B (f c d) by(rule Diff1-fold-graph)
      hence f c d = u by (rule IH [OF lessB Beq inj-onB Bu])
      moreover have f b d = v
      proof (rule IH[OF lessC Ceq inj-onC Cv])
        show fold-graph f z C (f b d) using C notinB Dfoldd by fastsimp
      qed
      ultimately show ?thesis
        using fun-left-comm [of c b] x x' by (auto simp add: o-def)
    qed
  qed
qed

lemma fold-graph-determ:
  fold-graph f z A x ⟹ fold-graph f z A y ⟹ y = x
apply (frule fold-graph-imp-finite [THEN finite-imp-nat-seg-image-inj-on])
apply (blast intro: fold-graph-determ-aux [rule-format])
done
```

lemma *fold-equality*:
  fold-graph f z A y $\Longrightarrow$ *fold f z A* = *y*
by (*unfold fold-def*) (*blast intro*: *fold-graph-determ*)

The base case for *fold*:

lemma (**in** −) *fold-empty* [*simp*]: *fold f z* {} = *z*
by (*unfold fold-def*) *blast*

The various recursion equations for *fold*:

lemma *fold-insert-aux*: *x* $\notin$ *A*
  $\Longrightarrow$ *fold-graph f z* (*insert x A*) *v* $\longleftrightarrow$
    ($\exists$ *y*. *fold-graph f z A y* $\wedge$ *v* = *f x y*)
apply *auto*
apply (*rule-tac A1* = *A* **and** *f1* = *f* **in** *finite-imp-fold-graph* [*THEN exE*])
 apply (*fastsimp dest*: *fold-graph-imp-finite*)
apply (*blast intro*: *fold-graph-determ*)
done

lemma *fold-insert* [*simp*]:
  finite A ==> *x* $\notin$ *A* ==> *fold f z* (*insert x A*) = *f x* (*fold f z A*)
apply (*simp add*: *fold-def fold-insert-aux*)
apply (*rule the-equality*)
 apply (*auto intro*: *finite-imp-fold-graph*
      cong add: *conj-cong simp add*: *fold-def*[*symmetric*] *fold-equality*)
done

lemma *fold-fun-comm*:
  finite A $\Longrightarrow$ *f x* (*fold f z A*) = *fold f* (*f x z*) *A*
proof (*induct rule*: *finite-induct*)
  case *empty* **then show** *?case* **by** *simp*
next
  case (*insert y A*) **then show** *?case*
    by (*simp add*: *fun-left-comm*[*of x*])
qed

lemma *fold-insert2*:
  finite A $\Longrightarrow$ *x* $\notin$ *A* $\Longrightarrow$ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
by (*simp add*: *fold-insert fold-fun-comm*)

lemma *fold-rec*:
assumes *finite A* **and** *x* $\in$ *A*
shows *fold f z A* = *f x* (*fold f z* (*A* − {*x*}))
proof −
  have *A*: *A* = *insert x* (*A* − {*x*}) **using** ‹*x* $\in$ *A*› **by** *blast*
  then have *fold f z A* = *fold f z* (*insert x* (*A* − {*x*})) **by** *simp*
  also have ... = *f x* (*fold f z* (*A* − {*x*}))
    by (*rule fold-insert*) (*simp add*: ‹*finite A*›)+
  finally show *?thesis* **.**
qed

lemma *fold-insert-remove*:
assumes *finite A*
shows *fold f z* (*insert x A*) = *f x* (*fold f z* (*A* − {*x*}))
proof −
from ‹*finite A*› **have** *finite* (*insert x A*) **by** *auto*
moreover have *x* ∈ *insert x A* **by** *auto*
ultimately have *fold f z* (*insert x A*) = *f x* (*fold f z* (*insert x A* − {*x*}))
by (*rule fold-rec*)
then show *?thesis* **by** *simp*
qed

end

A simplified version for idempotent functions:

locale *fun-left-comm-idem* = *fun-left-comm* +
assumes *fun-left-idem*: *f x* (*f x z*) = *f x z*
begin

The nice version:

lemma *fun-comp-idem* : *f x o f x* = *f x*
by (*simp add*: *fun-left-idem expand-fun-eq*)

lemma *fold-insert-idem*:
assumes *fin*: *finite A*
shows *fold f z* (*insert x A*) = *f x* (*fold f z A*)
proof *cases*
assume *x* ∈ *A*
then obtain *B* **where** *A* = *insert x B* **and** *x* ∉ *B* **by** (*rule set-insert*)
then show *?thesis* **using** *assms* **by** (*simp add*:*fun-left-idem*)
next
assume *x* ∉ *A* **then show** *?thesis* **using** *assms* **by** *simp*
qed

declare *fold-insert*[*simp del*] *fold-insert-idem*[*simp*]

lemma *fold-insert-idem2*:
finite A ⟹ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
by(*simp add*:*fold-fun-comm*)

end

17.3.2 The derived combinator *fold-image*

definition *fold-image* :: ('*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*b* ⇒ '*a set* ⇒ '*b*
where *fold-image f g* = *fold* (%*x y*. *f* (*g x*) *y*)

lemma *fold-image-empty*[*simp*]: *fold-image f g z* {} = *z*
by(*simp add*:*fold-image-def*)

context *ab-semigroup-mult*
begin

lemma *fold-image-insert*[*simp*]:
assumes *finite A* **and** $a \notin A$
shows *fold-image times g z* (*insert a A*) = *g a* ∗ (*fold-image times g z A*)
proof −
interpret *I*: *fun-left-comm* %*x y*. (*g x*) ∗ *y*
by *unfold-locales* (*simp add*: *mult-ac*)
show *?thesis* **using** *assms* **by**(*simp add*:*fold-image-def I*.*fold-insert*)
qed

lemma *fold-image-reindex*:
assumes *fin*: *finite A*
shows *inj-on h A* ⟹ *fold-image times g z* (*h*‘*A*) = *fold-image times* (*g*∘*h*) *z A*
using *fin* **apply** *induct*
apply *simp*
apply *simp*
done

lemma *fold-image-cong*:
finite A ⟹
(!!*x*. *x*:*A* ==> *g x* = *h x*) ==> *fold-image times g z A* = *fold-image times h z A*
apply (*subgoal-tac ALL C*. *C* <= *A* −−> (*ALL x*:*C*. *g x* = *h x*) −−> *fold-image times g z C* = *fold-image times h z C*)
apply *simp*
apply (*erule finite-induct*, *simp*)
apply (*simp add*: *subset-insert-iff*, *clarify*)
apply (*subgoal-tac finite C*)
prefer *2* **apply** (*blast dest*: *finite-subset* [*COMP swap-prems-rl*])
apply (*subgoal-tac C* = *insert x* (*C* − {*x*}))
prefer *2* **apply** *blast*
apply (*erule ssubst*)
apply (*drule spec*)
apply (*erule* (*1*) *notE impE*)
apply (*simp add*: *Ball-def del*: *insert-Diff-single*)
done

end

context *comm-monoid-mult*
begin

lemma *fold-image-Un-Int*:

finite A ==> finite B ==>
*fold-image times g 1 A * fold-image times g 1 B =*
*fold-image times g 1 (A Un B) * fold-image times g 1 (A Int B)*
by (*induct set*: *finite*)
(*auto simp add*: *mult-ac insert-absorb Int-insert-left*)

corollary *fold-Un-disjoint*:
finite A ==> finite B ==> A Int B = {} ==>
fold-image times g 1 (A Un B) =
*fold-image times g 1 A * fold-image times g 1 B*
by (*simp add*: *fold-image-Un-Int*)

lemma *fold-image-UN-disjoint*:
⟦ *finite I*; *ALL i:I. finite (A i)*;
ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {} ⟧
⟹ *fold-image times g 1 (UNION I A) =*
fold-image times (%i. fold-image times g 1 (A i)) 1 I
apply (*induct set*: *finite*, *simp*, *atomize*)
apply (*subgoal-tac ALL i:F. x ≠ i*)
prefer *2* **apply** *blast*
apply (*subgoal-tac A x Int UNION F A = {}*)
prefer *2* **apply** *blast*
apply (*simp add*: *fold-Un-disjoint*)
done

lemma *fold-image-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
fold-image times (%x. fold-image times (g x) 1 (B x)) 1 A =
fold-image times (split g) 1 (SIGMA x:A. B x)
apply (*subst Sigma-def*)
apply (*subst fold-image-UN-disjoint*, *assumption*, *simp*)
apply *blast*
apply (*erule fold-image-cong*)
apply (*subst fold-image-UN-disjoint*, *simp*, *simp*)
apply *blast*
apply *simp*
done

lemma *fold-image-distrib*: *finite A* ⟹
*fold-image times (%x. g x * h x) 1 A =*
*fold-image times g 1 A * fold-image times h 1 A*
by (*erule finite-induct*) (*simp-all add*: *mult-ac*)

lemma *fold-image-related*:
assumes *Re*: *R e e*
and *Rop*: ∀ *x1 y1 x2 y2*. *R x1 x2* ∧ *R y1 y2* ⟶ *R* (*x1 * y1*) (*x2 * y2*)
and *fS*: *finite S* **and** *Rfg*: ∀ *x*∈*S*. *R* (*h x*) (*g x*)
shows *R* (*fold-image* (*op **) *h e S*) (*fold-image* (*op **) *g e S*)
using *fS* **by** (*rule finite-subset-induct*) (*insert assms*, *auto*)

lemma *fold-image-eq-general*:
assumes *fS*: *finite S*
and *h*: $\forall\, y{\in}S'.\ \exists!x.\ x{\in}\ S \wedge h(x) = y$
and *f12*: $\forall\, x{\in}S.\ h\ x \in S' \wedge f2(h\ x) = f1\ x$
shows *fold-image* $(op\ *)$ *f1 e S* = *fold-image* $(op\ *)$ *f2 e S'*
proof−
from *h f12* **have** *hS*: *h* ' *S* = *S'* **by** *auto*
{fix *x y* **assume** *H*: $x \in S\ y \in S\ h\ x = h\ y$
from *f12 h H* **have** $x = y$ **by** *auto* **}**
hence *hinj*: *inj-on h S* **unfolding** *inj-on-def Ex1-def* **by** *blast*
from *f12* **have** *th*: $\bigwedge x.\ x \in S \Longrightarrow (f2 \circ h)\ x = f1\ x$ **by** *auto*
from *hS* **have** *fold-image* $(op\ *)$ *f2 e S'* = *fold-image* $(op\ *)$ *f2 e* (*h* ' *S*) **by** *simp*
also have ... = *fold-image* $(op\ *)$ (*f2 o h*) *e S*
using *fold-image-reindex*[*OF fS hinj*, *of f2 e*] **.**
also have ... = *fold-image* $(op\ *)$ *f1 e S* **using** *th fold-image-cong*[*OF fS*, *of f2 o h f1 e*]
by *blast*
finally show *?thesis* **..**
qed

lemma *fold-image-eq-general-inverses*:
assumes *fS*: *finite S*
and *kh*: $\bigwedge y.\ y \in T \Longrightarrow k\ y \in S \wedge h\ (k\ y) = y$
and *hk*: $\bigwedge x.\ x \in S \Longrightarrow h\ x \in T \wedge k\ (h\ x) = x\ \wedge\ g\ (h\ x) = f\ x$
shows *fold-image* $(op\ *)$ *f e S* = *fold-image* $(op\ *)$ *g e T*

apply (*rule fold-image-eq-general*[*OF fS*, *of T h g f e*])
apply (*rule ballI*)
apply (*frule kh*)
apply (*rule ex1I*[])
apply *blast*
apply *clarsimp*
apply (*drule hk*) **apply** *simp*
apply (*rule sym*)
apply (*erule conjunct1*[*OF conjunct2*[*OF hk*]])
apply (*rule ballI*)
apply (*drule hk*)
apply *blast*
done

end

17.4 Generalized summation over a set

interpretation *comm-monoid-add*: *comm-monoid-mult 0*::'*a*::*comm-monoid-add op* +
proof qed (*auto intro*: *add-assoc add-commute*)

definition *setsum* :: ('*a* => '*b*) => '*a set* => '*b*::*comm-monoid-add*
where *setsum f A* == *if finite A then fold-image* (*op* +) *f 0 A else 0*

abbreviation
Setsum (∑ - [*1000*] *999*) **where**
∑ *A* == *setsum* (%*x*. *x*) *A*

Now: lot's of fancy syntax. First, *setsum* (λ*x*. *e*) *A* is written ∑ *x*∈*A*. *e*.

syntax
-setsum :: *pttrn* => '*a set* => '*b* => '*b*::*comm-monoid-add* ((*3SUM* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-setsum :: *pttrn* => '*a set* => '*b* => '*b*::*comm-monoid-add* ((*3*∑ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setsum :: *pttrn* => '*a set* => '*b* => '*b*::*comm-monoid-add* ((*3*∑ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM i:*A*. *b* == *CONST setsum* (%*i*. *b*) *A*
∑ *i*∈*A*. *b* == *CONST setsum* (%*i*. *b*) *A*

Instead of ∑ *x*∈{*x*. *P*}. *e* we introduce the shorter ∑ *x*|*P*. *e*.

syntax
-qsetsum :: *pttrn* ⇒ *bool* ⇒ '*a* ⇒ '*a* ((*3SUM* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ '*a* ⇒ '*a* ((*3*∑ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ '*a* ⇒ '*a* ((*3*∑ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations
SUM x|*P*. *t* => *CONST setsum* (%*x*. *t*) {*x*. *P*}
∑ *x*|*P*. *t* => *CONST setsum* (%*x*. *t*) {*x*. *P*}

print-translation ⟪⟪
let
fun setsum-tr' [*Abs*(*x*,*Tx*,*t*), *Const* (*Collect*,-) $ *Abs*(*y*,*Ty*,*P*)] =
if x<>*y then raise Match*
else let val x' = *Syntax.mark-bound x*
val t' = *subst-bound*(*x'*,*t*)
val P' = *subst-bound*(*x'*,*P*)
in Syntax.const -qsetsum $ *Syntax.mark-bound x* $ *P'* $ *t' end*
in [(*setsum*, *setsum-tr'*)] *end*
⟫⟫

lemma *setsum-empty* [*simp*]: *setsum f* {} = *0*
by (*simp add*: *setsum-def*)

lemma *setsum-insert* [*simp*]:
  finite F ==> a ∉ F ==> setsum f (insert a F) = f a + setsum f F
by (*simp add*: *setsum-def*)

lemma *setsum-infinite* [*simp*]: ~ *finite A ==> setsum f A = 0*
by (*simp add*: *setsum-def*)

lemma *setsum-reindex*:
    inj-on f B ==> setsum h (f ‘ B) = setsum (h ∘ f) B
by(*auto simp add*: *setsum-def comm-monoid-add.fold-image-reindex dest*!:*finite-imageD*)

lemma *setsum-reindex-id*:
    inj-on f B ==> setsum f B = setsum id (f ‘ B)
by (*auto simp add*: *setsum-reindex*)

lemma *setsum-reindex-nonzero*:
  assumes *fS*: *finite S*
  and *nz*: ⋀ *x y*. *x ∈ S ⟹ y ∈ S ⟹ x ≠ y ⟹ f x = f y ⟹ h (f x) = 0*
  shows *setsum h (f ‘ S) = setsum (h o f) S*
using *nz*
proof(*induct rule*: *finite-induct*[*OF fS*])
  case *1* **thus** *?case* **by** *simp*
next
  case (*2 x F*)
  {assume *fxF*: *f x ∈ f ‘ F* **hence** ∃ *y ∈ F . f y = f x* **by** *auto*
    then obtain *y* **where** *y*: *y ∈ F f x = f y* **by** *auto*
    from *2.hyps y* **have** *xy*: *x ≠ y* **by** *auto*

    from *2.prems*[*of x y*] *2.hyps xy y* **have** *h0*: *h (f x) = 0* **by** *simp*
    have *setsum h (f ‘ insert x F) = setsum h (f ‘ F)* **using** *fxF* **by** *auto*
    also have *... = setsum (h o f) (insert x F)*
      unfolding *setsum-insert*[*OF ‹finite F› ‹x∉F›*]
      using *h0*
      apply *simp*
      apply (*rule 2.hyps*(*3*))
      apply (*rule-tac y=y* **in** *2.prems*)
      apply *simp-all*
      done
    finally have *?case* **.}**
  moreover
  {assume *fxF*: *f x ∉ f ‘ F*
    have *setsum h (f ‘ insert x F) = h (f x) + setsum h (f ‘ F)*
      using *fxF 2.hyps* **by** *simp*
    also have *... = setsum (h o f) (insert x F)*
      unfolding *setsum-insert*[*OF ‹finite F› ‹x∉F›*]
      apply *simp*
      apply (*rule cong*[*OF refl*[*of op + (h (f x))*]])
      apply (*rule 2.hyps*(*3*))
      apply (*rule-tac y=y* **in** *2.prems*)

```
    apply simp-all
    done
  finally have ?case .}
 ultimately show ?case by blast
qed

lemma setsum-cong:
  A = B ==> (!!x. x:B ==> f x = g x) ==> setsum f A = setsum g B
by(fastsimp simp: setsum-def intro: comm-monoid-add.fold-image-cong)

lemma strong-setsum-cong[cong]:
  A = B ==> (!!x. x:B =simp=> f x = g x)
  ==> setsum (%x. f x) A = setsum (%x. g x) B
by(fastsimp simp: simp-implies-def setsum-def intro: comm-monoid-add.fold-image-cong)

lemma setsum-cong2: ⟦⋀x. x ∈ A ⟹ f x = g x⟧ ⟹ setsum f A = setsum g A
by (rule setsum-cong[OF refl], auto)

lemma setsum-reindex-cong:
  [|inj-on f A; B = f ' A; !!a. a:A ⟹ g a = h (f a)|]
   ==> setsum h B = setsum g A
by (simp add: setsum-reindex cong: setsum-cong)


lemma setsum-0[simp]: setsum (%i. 0) A = 0
apply (clarsimp simp: setsum-def)
apply (erule finite-induct, auto)
done

lemma setsum-0': ALL a:A. f a = 0 ==> setsum f A = 0
by(simp add:setsum-cong)

lemma setsum-Un-Int: finite A ==> finite B ==>
  setsum g (A Un B) + setsum g (A Int B) = setsum g A + setsum g B
  — The reversed orientation looks more natural, but LOOPS as a simprule!
by(simp add: setsum-def comm-monoid-add.fold-image-Un-Int [symmetric])

lemma setsum-Un-disjoint: finite A ==> finite B
  ==> A Int B = {} ==> setsum g (A Un B) = setsum g A + setsum g B
by (subst setsum-Un-Int [symmetric], auto)

lemma setsum-mono-zero-left:
  assumes fT: finite T and ST: S ⊆ T
  and z: ∀ i ∈ T - S. f i = 0
  shows setsum f S = setsum f T
proof-
  have eq: T = S ∪ (T - S) using ST by blast
  have d: S ∩ (T - S) = {} using ST by blast
  from fT ST have f: finite S finite (T - S) by (auto intro: finite-subset)
```

show *?thesis*
by (*simp add*: *setsum-Un-disjoint*[*OF f d*, *unfolded eq*[*symmetric*]] *setsum-0'*[*OF z*])
qed

lemma *setsum-mono-zero-right*:
finite T $\Longrightarrow$ *S* $\subseteq$ *T* $\Longrightarrow$ $\forall$ *i* $\in$ *T* $-$ *S*. *f i* = *0* $\Longrightarrow$ *setsum f T* = *setsum f S*
by(*blast intro*!: *setsum-mono-zero-left*[*symmetric*])

lemma *setsum-mono-zero-cong-left*:
assumes *fT*: *finite T* **and** *ST*: *S* $\subseteq$ *T*
and *z*: $\forall$ *i* $\in$ *T* $-$ *S*. *g i* = *0*
and *fg*: $\bigwedge$*x*. *x* $\in$ *S* $\Longrightarrow$ *f x* = *g x*
shows *setsum f S* = *setsum g T*
proof−
have *eq*: *T* = *S* $\cup$ (*T* $-$ *S*) **using** *ST* **by** *blast*
have *d*: *S* $\cap$ (*T* $-$ *S*) = {} **using** *ST* **by** *blast*
from *fT ST* **have** *f*: *finite S finite* (*T* $-$ *S*) **by** (*auto intro*: *finite-subset*)
show *?thesis*
using *fg* **by** (*simp add*: *setsum-Un-disjoint*[*OF f d*, *unfolded eq*[*symmetric*]] *setsum-0'*[*OF z*])
qed

lemma *setsum-mono-zero-cong-right*:
assumes *fT*: *finite T* **and** *ST*: *S* $\subseteq$ *T*
and *z*: $\forall$ *i* $\in$ *T* $-$ *S*. *f i* = *0*
and *fg*: $\bigwedge$*x*. *x* $\in$ *S* $\Longrightarrow$ *f x* = *g x*
shows *setsum f T* = *setsum g S*
using *setsum-mono-zero-cong-left*[*OF fT ST z*] *fg*[*symmetric*] **by** *auto*

lemma *setsum-delta*:
assumes *fS*: *finite S*
shows *setsum* (*λk*. *if k=a then b k else 0*) *S* = (*if a* $\in$ *S then b a else 0*)
proof−
let *?f* = (*λk*. *if k=a then b k else 0*)
{**assume** *a*: *a* $\notin$ *S*
hence $\forall$ *k*$\in$ *S*. *?f k* = *0* **by** *simp*
hence *?thesis* **using** *a* **by** *simp*}
moreover
{**assume** *a*: *a* $\in$ *S*
let *?A* = *S* $-$ {*a*}
let *?B* = {*a*}
have *eq*: *S* = *?A* $\cup$ *?B* **using** *a* **by** *blast*
have *dj*: *?A* $\cap$ *?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 *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 *setsum-restrict-set*:
assumes *fA*: *finite A*
shows *setsum f* (*A* ∩ *B*) = *setsum* (λ*x*. *if x* ∈ *B then f x else 0*) *A*
proof−
from *fA* **have** *fab*: *finite* (*A* ∩ *B*) **by** *auto*
have *aba*: *A* ∩ *B* ⊆ *A* **by** *blast*
let *?g* = λ*x*. *if x* ∈ *A*∩*B then f x else 0*
from *setsum-mono-zero-left*[*OF fA aba*, *of ?g*]
show *?thesis* **by** *simp*
qed

lemma *setsum-cases*:
assumes *fA*: *finite A*
shows *setsum* (λ*x*. *if x* ∈ *B then f x else g x*) *A* =
setsum f (*A* ∩ *B*) + *setsum g* (*A* ∩ − *B*)
proof−
have *a*: *A* = *A* ∩ *B* ∪ *A* ∩ −*B* (*A* ∩ *B*) ∩ (*A* ∩ −*B*) = {}
by *blast*+
from *fA*
have *f*: *finite* (*A* ∩ *B*) *finite* (*A* ∩ −*B*) **by** *auto*
let *?g* = λ*x*. *if x* ∈ *B then f x else g x*
from *setsum-Un-disjoint*[*OF f a*(*2*), *of ?g*] *a*(*1*)
show *?thesis* **by** *simp*
qed

lemma *setsum-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. *i* ≠ *j* −−> *A i Int A j* = {}) ==>
setsum f (*UNION I A*) = (∑ *i*∈*I*. *setsum f* (*A i*))
by(*simp add*: *setsum-def comm-monoid-add.fold-image-UN-disjoint cong*: *setsum-cong*)

No need to assume that C is finite. If infinite, the rhs is directly 0, and $\bigcup C$ is also infinite, hence the lhs is also 0.

lemma *setsum-Union-disjoint*:
[| (*ALL A*:*C*. *finite A*);
(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* −−> *A Int B* = {}) |]
==> *setsum f* (*Union C*) = *setsum* (*setsum f*) *C*
apply (*cases finite C*)

prefer *2* **apply** (*force dest*: *finite-UnionD simp add*: *setsum-def*)
apply (*frule setsum-UN-disjoint* [*of C id f*])
apply (*unfold Union-def id-def*, *assumption+*)
done

lemma *setsum-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
$(\sum x{\in}A.\ (\sum y{\in}B\ x.\ f\ x\ y)) = (\sum (x,y){\in}(SIGMA\ x{:}A.\ B\ x).\ f\ x\ y)$
by(*simp add*:*setsum-def comm-monoid-add.fold-image-Sigma split-def cong*:*setsum-cong*)

Here we can eliminate the finiteness assumptions, by cases.

lemma *setsum-cartesian-product*:
$(\sum x{\in}A.\ (\sum y{\in}B.\ f\ x\ y)) = (\sum (x,y) \in A <*> B.\ f\ x\ y)$
apply (*cases finite A*)
apply (*cases finite B*)
apply (*simp add*: *setsum-Sigma*)
apply (*cases A={}*, *simp*)
apply (*simp*)
apply (*auto simp add*: *setsum-def*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *setsum-addf*: *setsum* (%*x*. *f x* + *g x*) *A* = (*setsum f A* + *setsum g A*)
by(*simp add*:*setsum-def comm-monoid-add.fold-image-distrib*)

17.4.1 Properties in more restricted classes of structures

lemma *setsum-SucD*: *setsum f A = Suc n ==> EX a:A. 0 < f a*
apply (*case-tac finite A*)
prefer *2* **apply** (*simp add*: *setsum-def*)
apply (*erule rev-mp*)
apply (*erule finite-induct*, *auto*)
done

lemma *setsum-eq-0-iff* [*simp*]:
finite F ==> (setsum f F = 0) = (ALL a:F. f a = (0::nat))
by (*induct set*: *finite*) *auto*

lemma *setsum-eq-Suc0-iff*: *finite A* $\Longrightarrow$
(*setsum f A = Suc 0*) = (*EX a:A. f a = Suc 0 & (ALL b:A. a≠b* $\longrightarrow$ *f b = 0*))
apply(*erule finite-induct*)
apply (*auto simp add*:*add-is-1*)
done

lemmas *setsum-eq-1-iff = setsum-eq-Suc0-iff*[*simplified One-nat-def*[*symmetric*]]

lemma *setsum-Un-nat*: *finite A ==> finite B ==>*
(*setsum f* (*A Un B*) :: *nat*) = *setsum f A* + *setsum f B* − *setsum f* (*A Int B*)
— For the natural numbers, we have subtraction.

by (*subst setsum-Un-Int* [*symmetric*], *auto simp add*: *algebra-simps*)

lemma *setsum-Un*: *finite A ==> finite B ==>*
(*setsum f* (*A Un B*) :: *'a* :: *ab-group-add*) =
setsum f A + setsum f B − setsum f (*A Int B*)
by (*subst setsum-Un-Int* [*symmetric*], *auto simp add*: *algebra-simps*)

lemma (**in** *comm-monoid-mult*) *fold-image-1*: *finite S* $\Longrightarrow$ $(\forall x{\in}S.\ f\ x = 1) \Longrightarrow$ *fold-image op ∗ f 1 S = 1*
apply (*induct set*: *finite*)
apply *simp* **by** (*auto simp add*: *fold-image-insert*)

lemma (**in** *comm-monoid-mult*) *fold-image-Un-one*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: $\forall x \in S{\cap}T.\ f\ x = 1$
shows *fold-image* (*op ∗*) *f 1* (*S ∪ T*) = *fold-image* (*op ∗*) *f 1 S ∗ fold-image* (*op ∗*) *f 1 T*
proof−
have *fold-image op ∗ f 1* (*S ∩ T*) = *1*
apply (*rule fold-image-1*)
using *fS fT I0* **by** *auto*
with *fold-image-Un-Int*[*OF fS fT*] **show** *?thesis* **by** *simp*
qed

lemma *setsum-eq-general-reverses*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *kh*: $\bigwedge y.\ y \in T \Longrightarrow k\ y \in S \wedge h\ (k\ y) = y$
and *hk*: $\bigwedge x.\ x \in S \Longrightarrow h\ x \in T \wedge k\ (h\ x) = x \wedge g\ (h\ x) = f\ x$
shows *setsum f S = setsum g T*
apply (*simp add*: *setsum-def fS fT*)
apply (*rule comm-monoid-add.fold-image-eq-general-inverses*[*OF fS*])
apply (*erule kh*)
apply (*erule hk*)
done

lemma *setsum-Un-zero*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: $\forall x \in S{\cap}T.\ f\ x = 0$
shows *setsum f* (*S ∪ T*) = *setsum f S* + *setsum f T*
using *fS fT*
apply (*simp add*: *setsum-def*)
apply (*rule comm-monoid-add.fold-image-Un-one*)
using *I0* **by** *auto*

lemma *setsum-UNION-zero*:
assumes *fS*: *finite S* **and** *fSS*: $\forall\ T \in S.$ *finite T*

and *f0*: $\bigwedge T1\ T2\ x.\ T1{\in}S \Longrightarrow T2{\in}S \Longrightarrow T1 \neq T2 \Longrightarrow x \in T1 \Longrightarrow x \in T2 \Longrightarrow f\ x = 0$
shows $setsum\ f\ (\bigcup S) = setsum\ (\lambda T.\ setsum\ f\ T)\ S$
using *fSS f0*
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **by** *simp*
next
case (*2 T F*)
then have *fTF*: *finite T* $\forall\ T{\in}F.$ *finite T finite F* **and** *TF*: $T \notin F$
and *H*: $setsum\ f\ (\bigcup F) = setsum\ (setsum\ f)\ F$ **by** (*auto simp add*: *finite-insert*)
from *fTF* **have** *fUF*: *finite* $(\bigcup F)$ **by** (*auto intro*: *finite-Union*)
from *2.prems TF fTF*
show *?case*
by (*auto simp add*: *H*[*symmetric*] *intro*: *setsum-Un-zero*[*OF fTF*(*1*) *fUF*, *of f*])
qed

lemma *setsum-diff1-nat*: $(setsum\ f\ (A - \{a\}) :: nat) =$
(*if a*:*A then setsum f A* $-$ *f a else setsum f A*)
apply (*case-tac finite A*)
prefer *2* **apply** (*simp add*: *setsum-def*)
apply (*erule finite-induct*)
apply (*auto simp add*: *insert-Diff-if*)
apply (*drule-tac a* $=$ *a* **in** *mk-disjoint-insert*, *auto*)
done

lemma *setsum-diff1*: *finite A* $\Longrightarrow$
$(setsum\ f\ (A - \{a\}) :: ('a{::}ab\text{-}group\text{-}add)) =$
(*if a*:*A then setsum f A* $-$ *f a else setsum f A*)
by (*erule finite-induct*) (*auto simp add*: *insert-Diff-if*)

lemma *setsum-diff1'*[*rule-format*]:
finite A $\Longrightarrow a \in A \longrightarrow (\sum\ x \in A.\ f\ x) = f\ a + (\sum\ x \in (A - \{a\}).\ f\ x)$
apply (*erule finite-induct*[**where** *F*=*A* **and** *P*=% *A*. $(a \in A \longrightarrow (\sum\ x \in A.\ f\ x) = f\ a + (\sum\ x \in (A - \{a\}).\ f\ x))$])
apply (*auto simp add*: *insert-Diff-if add-ac*)
done

lemma *setsum-diff-nat*:
assumes *finite B* **and** $B \subseteq A$
shows $(setsum\ f\ (A - B) :: nat) = (setsum\ f\ A) - (setsum\ f\ B)$
using *assms*
proof *induct*
show $setsum\ f\ (A - \{\}) = (setsum\ f\ A) - (setsum\ f\ \{\})$ **by** *simp*
next
fix *F x* **assume** *finF*: *finite F* **and** *xnotinF*: $x \notin F$

and *xFinA*: *insert x F* ⊆ *A*
and *IH*: *F* ⊆ *A* ⟹ *setsum f* (*A* − *F*) = *setsum f A* − *setsum f F*
from *xnotinF xFinA* **have** *xinAF*: *x* ∈ (*A* − *F*) **by** *simp*
from *xinAF* **have** *A*: *setsum f* ((*A* − *F*) − {*x*}) = *setsum f* (*A* − *F*) − *f x*
by (*simp add*: *setsum-diff1-nat*)
from *xFinA* **have** *F* ⊆ *A* **by** *simp*
with *IH* **have** *setsum f* (*A* − *F*) = *setsum f A* − *setsum f F* **by** *simp*
with *A* **have** *B*: *setsum f* ((*A* − *F*) − {*x*}) = *setsum f A* − *setsum f F* − *f x*
by *simp*
from *xnotinF* **have** *A* − *insert x F* = (*A* − *F*) − {*x*} **by** *auto*
with *B* **have** *C*: *setsum f* (*A* − *insert x F*) = *setsum f A* − *setsum f F* − *f x*
by *simp*
from *finF xnotinF* **have** *setsum f* (*insert x F*) = *setsum f F* + *f x* **by** *simp*
with *C* **have** *setsum f* (*A* − *insert x F*) = *setsum f A* − *setsum f* (*insert x F*)
by *simp*
thus *setsum f* (*A* − *insert x F*) = *setsum f A* − *setsum f* (*insert x F*) **by** *simp*
qed

lemma *setsum-diff*:
assumes *le*: *finite A B* ⊆ *A*
shows *setsum f* (*A* − *B*) = *setsum f A* − ((*setsum f B*)::('*a*::*ab-group-add*))
proof −
from *le* **have** *finiteB*: *finite B* **using** *finite-subset* **by** *auto*
show *?thesis* **using** *finiteB le*
proof *induct*
case *empty*
thus *?case* **by** *auto*
next
case (*insert x F*)
thus *?case* **using** *le finiteB*
by (*simp add*: *Diff-insert*[**where** *a*=*x* **and** *B*=*F*] *setsum-diff1 insert-absorb*)
qed
qed

lemma *setsum-mono*:
assumes *le*: ⋀*i*. *i*∈*K* ⟹ *f* (*i*::'*a*) ≤ ((*g i*)::('*b*::{*comm-monoid-add*, *pordered-ab-semigroup-add*}))
shows (∑ *i*∈*K*. *f i*) ≤ (∑ *i*∈*K*. *g i*)
proof (*cases finite K*)
case *True*
thus *?thesis* **using** *le*
proof *induct*
case *empty*
thus *?case* **by** *simp*
next
case *insert*
thus *?case* **using** *add-mono* **by** *fastsimp*
qed
next
case *False*

thus *?thesis*
by (*simp add*: *setsum-def*)
qed

lemma *setsum-strict-mono*:
fixes $f :: 'a \Rightarrow 'b$::{*pordered-cancel-ab-semigroup-add*,*comm-monoid-add*}
assumes *finite* A $A \neq \{\}$
and $!!x.\ x{:}A \Longrightarrow f\ x < g\ x$
shows *setsum* $f\ A <$ *setsum* $g\ A$
using *prems*
proof (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **by** (*auto simp*: *add-strict-mono*)
qed

lemma *setsum-negf*:
setsum $(\%x.\ -\ (f\ x)::'a::$*ab-group-add*$)\ A = -$ *setsum* $f\ A$
proof (*cases finite A*)
case *True* **thus** *?thesis* **by** (*induct set*: *finite*) *auto*
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-subtractf*:
setsum $(\%x.\ ((f\ x)::'a::$*ab-group-add*$)\ -\ g\ x)\ A =$
setsum $f\ A -$ *setsum* $g\ A$
proof (*cases finite A*)
case *True* **thus** *?thesis* **by** (*simp add*: *diff-minus setsum-addf setsum-negf*)
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-nonneg*:
assumes *nn*: $\forall x{\in}A.\ (0::'a::\{$*pordered-ab-semigroup-add*,*comm-monoid-add*$\}) \leq f\ x$
shows $0 \leq$ *setsum* $f\ A$
proof (*cases finite A*)
case *True* **thus** *?thesis* **using** *nn*
proof *induct*
case *empty* **then show** *?case* **by** *simp*
next
case (*insert x F*)
then have $0 + 0 \leq f\ x +$ *setsum* $f\ F$ **by** (*blast intro*: *add-mono*)
with *insert* **show** *?case* **by** *simp*
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-nonpos*:
assumes *np*: $\forall x{\in}A.\ f\,x \leq (0::'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add, comm\text{-}monoid\text{-}add\})$
shows $setsum\ f\ A \leq 0$
proof (*cases finite A*)
case *True* **thus** *?thesis* **using** *np*
proof *induct*
case *empty* **then show** *?case* **by** *simp*
next
case (*insert x F*)
then have $f\,x + setsum\ f\ F \leq 0 + 0$ **by** (*blast intro*: *add-mono*)
with *insert* **show** *?case* **by** *simp*
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-mono2*:
fixes $f :: 'a \Rightarrow 'b :: \{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, comm\text{-}monoid\text{-}add\}$
assumes *fin*: *finite B* **and** *sub*: $A \subseteq B$ **and** *nn*: $\bigwedge b.\ b \in B{-}A \Longrightarrow 0 \leq f\,b$
shows $setsum\ f\ A \leq setsum\ f\ B$
proof −
have $setsum\ f\ A \leq setsum\ f\ A + setsum\ f\ (B{-}A)$
by(*simp add*: *add-increasing2*[*OF setsum-nonneg*] *nn Ball-def*)
also have $\ldots = setsum\ f\ (A \cup (B{-}A))$ **using** *fin finite-subset*[*OF sub fin*]
by (*simp add*:*setsum-Un-disjoint del*:*Un-Diff-cancel*)
also have $A \cup (B{-}A) = B$ **using** *sub* **by** *blast*
finally show *?thesis* **.**
qed

lemma *setsum-mono3*: *finite B ==> A <= B ==>*
ALL x: *B − A*.
0 <= ((f x)::'a::{comm-monoid-add,pordered-ab-semigroup-add}) ==>
setsum f A <= setsum f B
apply (*subgoal-tac setsum f B = setsum f A + setsum f (B − A)*)
apply (*erule ssubst*)
apply (*subgoal-tac setsum f A + 0 <= setsum f A + setsum f (B − A)*)
apply *simp*
apply (*rule add-left-mono*)
apply (*erule setsum-nonneg*)
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*erule finite-subset*, *assumption*)
apply (*rule finite-subset*)
prefer *2*
apply *assumption*
apply *auto*
apply (*rule setsum-cong*)
apply *auto*
done

lemma *setsum-right-distrib*:
fixes $f :: 'a => ('b::semiring\text{-}0)$
shows $r * setsum\ f\ A = setsum\ (\%n.\ r * f\ n)\ A$
proof (*cases finite A*)
case *True*
thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*simp add*: *right-distrib*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-left-distrib*:
$setsum\ f\ A * (r::'a::semiring\text{-}0) = (\sum n{\in}A.\ f\ n * r)$
proof (*cases finite A*)
case *True*
then show *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*simp add*: *left-distrib*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-divide-distrib*:
$setsum\ f\ A\ /\ (r::'a::field) = (\sum n{\in}A.\ f\ n\ /\ r)$
proof (*cases finite A*)
case *True*
then show *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*simp add*: *add-divide-distrib*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-abs*[*iff*]:
fixes $f :: 'a => ('b::pordered\text{-}ab\text{-}group\text{-}add\text{-}abs)$
shows $abs\ (setsum\ f\ A) \leq setsum\ (\%i.\ abs(f\ i))\ A$
proof (*cases finite A*)
case *True*

thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*)
thus *?case* **by** (*auto intro*: *abs-triangle-ineq order-trans*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-abs-ge-zero*[*iff*]:
fixes *f* :: *'a* => (*'b*::*pordered-ab-group-add-abs*)
shows *0* ≤ *setsum* (%*i*. *abs*(*f i*)) *A*
proof (*cases finite A*)
case *True*
thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*auto simp*: *add-nonneg-nonneg*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *abs-setsum-abs*[*simp*]:
fixes *f* :: *'a* => (*'b*::*pordered-ab-group-add-abs*)
shows *abs* (∑ *a*∈*A*. *abs*(*f a*)) = (∑ *a*∈*A*. *abs*(*f a*))
proof (*cases finite A*)
case *True*
thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert a A*)
hence |∑ *a*∈*insert a A*. |*f a*|| = ||*f a*| + (∑ *a*∈*A*. |*f a*|)| **by** *simp*
also have ... = ||*f a*| + |∑ *a*∈*A*. |*f a*||| **using** *insert* **by** *simp*
also have ... = |*f a*| + |∑ *a*∈*A*. |*f a*||
by (*simp del*: *abs-of-nonneg*)
also have ... = (∑ *a*∈*insert a A*. |*f a*|) **using** *insert* **by** *simp*
finally show *?case* **.**
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

Commuting outer and inner summation

lemma *swap-inj-on*:

inj-on (%(*i*, *j*). (*j*, *i*)) (*A* × *B*)
by (*unfold inj-on-def*) *fast*

lemma *swap-product*:
(%(*i*, *j*). (*j*, *i*)) ' (*A* × *B*) = *B* × *A*
by (*simp add*: *split-def image-def*) *blast*

lemma *setsum-commute*:
(∑*i*∈*A*. ∑*j*∈*B*. *f i j*) = (∑*j*∈*B*. ∑*i*∈*A*. *f i j*)
proof (*simp add*: *setsum-cartesian-product*)
have (∑(*x*,*y*) ∈ *A* <*> *B*. *f x y*) =
(∑(*y*,*x*) ∈ (%(*i*, *j*). (*j*, *i*)) ' (*A* × *B*). *f x y*)
(**is** *?s* = -)
apply (*simp add*: *setsum-reindex* [**where** *f* = %(*i*, *j*). (*j*, *i*)] *swap-inj-on*)
apply (*simp add*: *split-def*)
done
also have ... = (∑(*y*,*x*)∈*B* × *A*. *f x y*)
(**is** - = *?t*)
apply (*simp add*: *swap-product*)
done
finally show *?s* = *?t* **.**
qed

lemma *setsum-product*:
fixes *f* :: *'a* => (*'b*::*semiring-0*)
shows *setsum f A* ∗ *setsum g B* = (∑*i*∈*A*. ∑*j*∈*B*. *f i* ∗ *g j*)
by (*simp add*: *setsum-right-distrib setsum-left-distrib*) (*rule setsum-commute*)

17.5 Generalized product over a set

definition *setprod* :: (*'a* => *'b*) => *'a set* => *'b*::*comm-monoid-mult*
where *setprod f A* == *if finite A then fold-image* (*op* ∗) *f 1 A else 1*

abbreviation
Setprod (∏- [*1000*] *999*) **where**
∏*A* == *setprod* (%*x*. *x*) *A*

syntax
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3PROD* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*∏-∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*∏-∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
PROD i:*A*. *b* == *CONST setprod* (%*i*. *b*) *A*

$\prod i{\in}A.\ b == CONST\ setprod\ (\%i.\ b)\ A$

Instead of $\prod x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\prod x|P.\ e$.

syntax
-qsetprod :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3PROD - |/ -./ -) [0,0,10] 10)
syntax (*xsymbols*)
-qsetprod :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∏ - | (-)./ -) [0,0,10] 10)
syntax (*HTML* **output**)
-qsetprod :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∏ - | (-)./ -) [0,0,10] 10)

translations
PROD x|P. t => CONST setprod (%x. t) {x. P}
∏ x|P. t => CONST setprod (%x. t) {x. P}

lemma *setprod-empty* [*simp*]: *setprod f {} = 1*
by (*auto simp add*: *setprod-def*)

lemma *setprod-insert* [*simp*]: [| *finite A*; *a ∉ A* |] ==>
*setprod f (insert a A) = f a * setprod f A*
by (*simp add*: *setprod-def*)

lemma *setprod-infinite* [*simp*]: *~ finite A ==> setprod f A = 1*
by (*simp add*: *setprod-def*)

lemma *setprod-reindex*:
inj-on f B ==> setprod h (f ' B) = setprod (h ∘ f) B
by(*auto simp*: *setprod-def fold-image-reindex dest*!:*finite-imageD*)

lemma *setprod-reindex-id*: *inj-on f B ==> setprod f B = setprod id (f ' B)*
by (*auto simp add*: *setprod-reindex*)

lemma *setprod-cong*:
A = B ==> (!!x. x:B ==> f x = g x) ==> setprod f A = setprod g B
by(*fastsimp simp*: *setprod-def intro*: *fold-image-cong*)

lemma *strong-setprod-cong*[*cong*]:
A = B ==> (!!x. x:B =simp=> f x = g x) ==> setprod f A = setprod g B
by(*fastsimp simp*: *simp-implies-def setprod-def intro*: *fold-image-cong*)

lemma *setprod-reindex-cong*: *inj-on f A ==>*
B = f ' A ==> g = h ∘ f ==> setprod h B = setprod g A
by (*frule setprod-reindex*, *simp*)

lemma *strong-setprod-reindex-cong*: **assumes** *i*: *inj-on f A*
and *B*: *B = f ' A* **and** *eq*: *⋀x. x ∈ A ⟹ g x = (h ∘ f) x*
shows *setprod h B = setprod g A*
proof−
have *setprod h B = setprod (h o f) A*

```
    by (simp add: B setprod-reindex[OF i, of h])
  then show ?thesis apply simp
    apply (rule setprod-cong)
    apply simp
    by (simp add: eq)
qed

lemma setprod-Un-one:
  assumes fS: finite S and fT: finite T
  and I0: ∀ x ∈ S∩T. f x = 1
  shows setprod f (S ∪ T) = setprod f S * setprod f T
  using fS fT
  apply (simp add: setprod-def)
  apply (rule fold-image-Un-one)
  using I0 by auto


lemma setprod-1: setprod (%i. 1) A = 1
apply (case-tac finite A)
apply (erule finite-induct, auto simp add: mult-ac)
done

lemma setprod-1': ALL a:F. f a = 1 ==> setprod f F = 1
apply (subgoal-tac setprod f F = setprod (%x. 1) F)
apply (erule ssubst, rule setprod-1)
apply (rule setprod-cong, auto)
done

lemma setprod-Un-Int: finite A ==> finite B
   ==> setprod g (A Un B) * setprod g (A Int B) = setprod g A * setprod g B
by(simp add: setprod-def fold-image-Un-Int[symmetric])

lemma setprod-Un-disjoint: finite A ==> finite B
  ==> A Int B = {} ==> setprod g (A Un B) = setprod g A * setprod g B
by (subst setprod-Un-Int [symmetric], auto)

lemma setprod-mono-one-left:
  assumes fT: finite T and ST: S ⊆ T
  and z: ∀ i ∈ T - S. f i = 1
  shows setprod f S = setprod f T
proof-
  have eq: T = S ∪ (T - S) using ST by blast
  have d: S ∩ (T - S) = {} using ST by blast
  from fT ST have f: finite S finite (T - S) by (auto intro: finite-subset)
  show ?thesis
  by (simp add: setprod-Un-disjoint[OF f d, unfolded eq[symmetric]] setprod-1'[OF
z])
qed
```

lemmas *setprod-mono-one-right* = *setprod-mono-one-left* [*THEN sym*]

lemma *setprod-delta*:
assumes *fS*: *finite S*
shows *setprod* (λ*k*. *if k=a then b k else 1*) *S* = (*if a* ∈ *S then b a else 1*)
proof−
let *?f* = (λ*k*. *if k=a then b k else 1*)
{assume *a*: *a* ∉ *S*
hence ∀ *k*∈ *S*. *?f k* = *1* **by** *simp*
hence *?thesis* **using** *a* **by** (*simp add*: *setprod-1 cong add*: *setprod-cong*) **}**
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 *fA1*: *setprod ?f ?A* = *1* **apply** (*rule setprod-1'*) **by** *auto*
have *setprod ?f ?A* ∗ *setprod ?f ?B* = *setprod ?f S*
using *setprod-Un-disjoint*[*OF fAB dj*, *of ?f*, *unfolded eq*[*symmetric*]]
by *simp*
then have *?thesis* **using** *a* **by** (*simp add*: *fA1 cong add*: *setprod-cong cong del*: *if-weak-cong*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *setprod-delta'*:
assumes *fS*: *finite S* **shows**
setprod (λ*k*. *if a* = *k then b k else 1*) *S* =
(*if a*∈ *S then b a else 1*)
using *setprod-delta*[*OF fS*, *of a b*, *symmetric*]
by (*auto intro*: *setprod-cong*)

lemma *setprod-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. *i* ≠ *j* −−> *A i Int A j* = {}) ==>
setprod f (*UNION I A*) = *setprod* (%*i*. *setprod f* (*A i*)) *I*
by(*simp add*: *setprod-def fold-image-UN-disjoint cong*: *setprod-cong*)

lemma *setprod-Union-disjoint*:
[| (*ALL A*:*C*. *finite A*);
(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* −−> *A Int B* = {}) |]
==> *setprod f* (*Union C*) = *setprod* (*setprod f*) *C*
apply (*cases finite C*)
prefer *2* **apply** (*force dest*: *finite-UnionD simp add*: *setprod-def*)
apply (*frule setprod-UN-disjoint* [*of C id f*])
apply (*unfold Union-def id-def*, *assumption*+)
done

lemma *setprod-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
(∏ x∈A. (∏ y∈ B x. f x y)) =
(∏ (x,y)∈(SIGMA x:A. B x). f x y)
by(*simp add*:*setprod-def fold-image-Sigma split-def cong*:*setprod-cong*)

Here we can eliminate the finiteness assumptions, by cases.

lemma *setprod-cartesian-product*:
(∏ x∈A. (∏ y∈ B. f x y)) = (∏ (x,y)∈(A <> B). f x y)*
apply (*cases finite A*)
apply (*cases finite B*)
apply (*simp add*: *setprod-Sigma*)
apply (*cases A={}*, *simp*)
apply (*simp add*: *setprod-1*)
apply (*auto simp add*: *setprod-def*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *setprod-timesf*:
*setprod (%x. f x * g x) A = (setprod f A * setprod g A)*
by(*simp add*:*setprod-def fold-image-distrib*)

17.5.1 Properties in more restricted classes of structures

lemma *setprod-eq-1-iff* [*simp*]:
finite F ==> (setprod f F = 1) = (ALL a:F. f a = (1::nat))
by (*induct set*: *finite*) *auto*

lemma *setprod-zero*:
finite A ==> EX x: A. f x = (0::'a::comm-semiring-1) ==> setprod f A = 0
apply (*induct set*: *finite*, *force*, *clarsimp*)
apply (*erule disjE*, *auto*)
done

lemma *setprod-nonneg* [*rule-format*]:
(ALL x: A. (0::'a::ordered-semidom) ≤ f x) --> 0 ≤ setprod f A
by (*cases finite A*, *induct set*: *finite*, *simp-all add*: *mult-nonneg-nonneg*)

lemma *setprod-pos* [*rule-format*]: *(ALL x: A. (0::'a::ordered-semidom) < f x)*
--> 0 < setprod f A
by (*cases finite A*, *induct set*: *finite*, *simp-all add*: *mult-pos-pos*)

lemma *setprod-zero-iff*[*simp*]: *finite A ==>*
(setprod f A = (0::'a::{comm-semiring-1,no-zero-divisors})) =
(EX x: A. f x = 0)
by (*erule finite-induct*, *auto simp*:*no-zero-divisors*)

lemma *setprod-pos-nat*:
finite S ==> (ALL x : S. f x > (0::nat)) ==> setprod f S > 0

using *setprod-zero-iff* **by**(*simp del*:*neq0-conv add*:*neq0-conv*[*symmetric*])

lemma *setprod-pos-nat-iff*[*simp*]:
finite S ==> (setprod f S > 0) = (ALL x : S. f x > (0::nat))
using *setprod-zero-iff* **by**(*simp del*:*neq0-conv add*:*neq0-conv*[*symmetric*])

lemma *setprod-Un*: *finite A ==> finite B ==> (ALL x*: *A Int B. f x ≠ 0) ==>*
(setprod f (A Un B) :: 'a ::{field})
*= setprod f A * setprod f B / setprod f (A Int B)*
by (*subst setprod-Un-Int* [*symmetric*], *auto*)

lemma *setprod-diff1*: *finite A ==> f a ≠ 0 ==>*
(setprod f (A − {a}) :: 'a :: {field}) =
(if a:*A then setprod f A / f a else setprod f A)*
by (*erule finite-induct*) (*auto simp add*: *insert-Diff-if*)

lemma *setprod-inversef*: *finite A ==>*
ALL x: *A. f x ≠ (0::'a::{field,division-by-zero}) ==>*
setprod (inverse ∘ f) A = inverse (setprod f A)
by (*erule finite-induct*) *auto*

lemma *setprod-dividef*:
[|*finite A*;
∀ *x* ∈ *A. g x ≠ (0::'a::{field,division-by-zero})*|]
==> setprod (%x. f x / g x) A = setprod f A / setprod g A
apply (*subgoal-tac*
*setprod (%x. f x / g x) A = setprod (%x. f x * (inverse ∘ g) x) A*)
apply (*erule ssubst*)
apply (*subst divide-inverse*)
apply (*subst setprod-timesf*)
apply (*subst setprod-inversef*, *assumption+*, *rule refl*)
apply (*rule setprod-cong*, *rule refl*)
apply (*subst divide-inverse*, *auto*)
done

lemma *setprod-dvd-setprod* [*rule-format*]:
(ALL x : A. f x dvd g x) ⟶ setprod f A dvd setprod g A
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply (*auto simp add*: *dvd-def*)
apply (*rule-tac x = k * ka* **in** *exI*)
apply (*simp add*: *algebra-simps*)
done

lemma *setprod-dvd-setprod-subset*:
finite B ⟹ A <= B ⟹ setprod f A dvd setprod f B
apply (*subgoal-tac setprod f B = setprod f A * setprod f (B − A)*)
apply (*unfold dvd-def*, *blast*)
apply (*subst setprod-Un-disjoint* [*symmetric*])

apply (*auto elim*: *finite-subset intro*: *setprod-cong*)
done

lemma *setprod-dvd-setprod-subset2*:
finite B $\Longrightarrow$ *A* <= *B* $\Longrightarrow$ *ALL x* : *A*. (*f x*::'*a*::*comm-semiring-1*) *dvd g x* $\Longrightarrow$
setprod f A dvd setprod g B
apply (*rule dvd-trans*)
apply (*rule setprod-dvd-setprod*, *erule* (*1*) *bspec*)
apply (*erule* (*1*) *setprod-dvd-setprod-subset*)
done

lemma *dvd-setprod*: *finite A* $\Longrightarrow$ *i*:*A* $\Longrightarrow$
(*f i* ::'*a*::*comm-semiring-1*) *dvd setprod f A*
by (*induct set*: *finite*) (*auto intro*: *dvd-mult*)

lemma *dvd-setsum* [*rule-format*]: (*ALL i* : *A*. *d dvd f i*) $\longrightarrow$
(*d*::'*a*::*comm-semiring-1*) *dvd* (*SUM x* : *A*. *f x*)
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply *auto*
done

17.6 Finite cardinality

This definition, although traditional, is ugly to work with: *card A* == *LEAST n*. *EX f*. *A* = {*f i* | *i*. *i* < *n*}. But now that we have *setsum* things are easy:

definition *card* :: '*a set* $\Rightarrow$ *nat*
where *card A* = *setsum* (λx. *1*) *A*

lemma *card-empty* [*simp*]: *card* {} = *0*
by (*simp add*: *card-def*)

lemma *card-infinite* [*simp*]: ~ *finite A* ==> *card A* = *0*
by (*simp add*: *card-def*)

lemma *card-eq-setsum*: *card A* = *setsum* (%*x*. *1*) *A*
by (*simp add*: *card-def*)

lemma *card-insert-disjoint* [*simp*]:
finite A ==> *x* $\notin$ *A* ==> *card* (*insert x A*) = *Suc*(*card A*)
by(*simp add*: *card-def*)

lemma *card-insert-if*:
finite A ==> *card* (*insert x A*) = (*if x*:*A then card A else Suc*(*card*(*A*)))
by (*simp add*: *insert-absorb*)

lemma *card-0-eq* [*simp*,*noatp*]: *finite A* ==> (*card A* = *0*) = (*A* = {})
apply *auto*

apply (*drule-tac a = x* **in** *mk-disjoint-insert*, *clarify*, *auto*)
done

lemma *card-eq-0-iff*: (*card A = 0*) = (*A = {}* | ~ *finite A*)
by *auto*

lemma *card-Suc-Diff1*: *finite A ==> x*: *A ==> Suc* (*card* (*A* − {*x*})) = *card A*
apply(*rule-tac t = A* **in** *insert-Diff* [*THEN subst*], *assumption*)
apply(*simp del*:*insert-Diff-single*)
done

lemma *card-Diff-singleton*:
finite A ==> x: *A ==> card* (*A* − {*x*}) = *card A* − *1*
by (*simp add*: *card-Suc-Diff1* [*symmetric*])

lemma *card-Diff-singleton-if*:
finite A ==> card (*A*−{*x*}) = (*if x* : *A then card A* − *1 else card A*)
by (*simp add*: *card-Diff-singleton*)

lemma *card-Diff-insert*[*simp*]:
assumes *finite A* **and** *a*:*A* **and** *a* ~: *B*
shows *card*(*A* − *insert a B*) = *card*(*A* − *B*) − *1*
proof −
have *A* − *insert a B* = (*A* − *B*) − {*a*} **using** *assms* **by** *blast*
then show *?thesis* **using** *assms* **by**(*simp add*:*card-Diff-singleton*)
qed

lemma *card-insert*: *finite A ==> card* (*insert x A*) = *Suc* (*card* (*A* − {*x*}))
by (*simp add*: *card-insert-if card-Suc-Diff1 del*:*card-Diff-insert*)

lemma *card-insert-le*: *finite A ==> card A <= card* (*insert x A*)
by (*simp add*: *card-insert-if*)

lemma *card-mono*: ⟦ *finite B*; *A* ⊆ *B* ⟧ ⟹ *card A* ≤ *card B*
by (*simp add*: *card-def setsum-mono2*)

lemma *card-seteq*: *finite B ==>* (!!*A*. *A <= B ==> card B <= card A ==> A* = *B*)
apply (*induct set*: *finite*, *simp*, *clarify*)
apply (*subgoal-tac finite A* & *A* − {*x*} <= *F*)
prefer *2* **apply** (*blast intro*: *finite-subset*, *atomize*)
apply (*drule-tac x = A* − {*x*} **in** *spec*)
apply (*simp add*: *card-Diff-singleton-if split add*: *split-if-asm*)
apply (*case-tac card A*, *auto*)
done

lemma *psubset-card-mono*: *finite B ==> A < B ==> card A < card B*
apply (*simp add*: *psubset-eq linorder-not-le* [*symmetric*])

apply (*blast dest*: *card-seteq*)
done

lemma *card-Un-Int*: *finite A* ==> *finite B*
==> *card A* + *card B* = *card* (*A Un B*) + *card* (*A Int B*)
by(*simp add*:*card-def setsum-Un-Int*)

lemma *card-Un-disjoint*: *finite A* ==> *finite B*
==> *A Int B* = {} ==> *card* (*A Un B*) = *card A* + *card B*
by (*simp add*: *card-Un-Int*)

lemma *card-Diff-subset*:
finite B ==> *B* <= *A* ==> *card* (*A* − *B*) = *card A* − *card B*
by(*simp add*:*card-def setsum-diff-nat*)

lemma *card-Diff1-less*: *finite A* ==> *x*: *A* ==> *card* (*A* − {*x*}) < *card A*
apply (*rule Suc-less-SucD*)
apply (*simp add*: *card-Suc-Diff1 del*:*card-Diff-insert*)
done

lemma *card-Diff2-less*:
finite A ==> *x*: *A* ==> *y*: *A* ==> *card* (*A* − {*x*} − {*y*}) < *card A*
apply (*case-tac x* = *y*)
apply (*simp add*: *card-Diff1-less del*:*card-Diff-insert*)
apply (*rule less-trans*)
prefer *2* **apply** (*auto intro*!: *card-Diff1-less simp del*:*card-Diff-insert*)
done

lemma *card-Diff1-le*: *finite A* ==> *card* (*A* − {*x*}) <= *card A*
apply (*case-tac x* : *A*)
apply (*simp-all add*: *card-Diff1-less less-imp-le*)
done

lemma *card-psubset*: *finite B* ==> *A* ⊆ *B* ==> *card A* < *card B* ==> *A* < *B*
by (*erule psubsetI*, *blast*)

lemma *insert-partition*:
⟦ *x* ∉ *F*; ∀ *c1* ∈ *insert x F*. ∀ *c2* ∈ *insert x F*. *c1* ≠ *c2* ⟶ *c1* ∩ *c2* = {} ⟧
⟹ *x* ∩ ⋃ *F* = {}
by *auto*

main cardinality theorem

lemma *card-partition* [*rule-format*]:
finite C ==>
finite (⋃ *C*) −−>
(∀ *c*∈*C*. *card c* = *k*) −−>
(∀ *c1* ∈ *C*. ∀ *c2* ∈ *C*. *c1* ≠ *c2* −−> *c1* ∩ *c2* = {}) −−>
k ∗ *card*(*C*) = *card* (⋃ *C*)
apply (*erule finite-induct*, *simp*)

apply (*simp add*: *card-insert-disjoint card-Un-disjoint insert-partition*
finite-subset [*of* - $\bigcup$ (*insert x F*)])
done

The form of a finite set of given cardinality

lemma *card-eq-SucD*:
assumes *card A = Suc k*
shows $\exists$ *b B. A = insert b B & b* $\notin$ *B & card B = k & (k=0* $\longrightarrow$ *B={})*
proof −
have *fin*: *finite A* **using** *assms* **by** (*auto intro*: *ccontr*)
moreover have *card A* $\neq$ *0* **using** *assms* **by** *auto*
ultimately obtain *b* **where** *b*: *b* $\in$ *A* **by** *auto*
show *?thesis*
proof (*intro exI conjI*)
show *A = insert b* (*A*−{*b*}) **using** *b* **by** *blast*
show *b* $\notin$ *A* − {*b*} **by** *blast*
show *card* (*A* − {*b*}) = *k* **and** *k = 0* $\longrightarrow$ *A* − {*b*} = {}
using *assms b fin* **by**(*fastsimp dest*:*mk-disjoint-insert*)+
qed
qed

lemma *card-Suc-eq*:
(*card A = Suc k*) =
(∃ *b B. A = insert b B & b* $\notin$ *B & card B = k & (k=0* $\longrightarrow$ *B={}*))
apply(*rule iffI*)
apply(*erule card-eq-SucD*)
apply(*auto*)
apply(*subst card-insert*)
apply(*auto intro*:*ccontr*)
done

lemma *setsum-constant* [*simp*]: ($\sum$ *x* ∈ *A. y*) = *of-nat*(*card A*) ∗ *y*
apply (*cases finite A*)
apply (*erule finite-induct*)
apply (*auto simp add*: *algebra-simps*)
done

lemma *setprod-constant*: *finite A* ==> ($\prod$ *x*∈ *A.* (*y*::′*a*::{*recpower*, *comm-monoid-mult*}))
= *y* ^(*card A*)
apply (*erule finite-induct*)
apply (*auto simp add*: *power-Suc*)
done

lemma *setprod-gen-delta*:
assumes *fS*: *finite S*
shows *setprod* (λ*k. if k=a then b k else c*) *S* = (*if a* ∈ *S then* (*b a* ::′*a*::{*comm-monoid-mult*,
recpower}) ∗ *c* ^ (*card S* − *1*) *else c* ^ *card S*)
proof−
let *?f* = (λ*k. if k=a then b k else c*)

{assume *a*: $a \notin S$
hence $\forall\ k{\in}\ S.\ ?f\ k = c$ **by** *simp*
hence *?thesis* **using** *a setprod-constant*[*OF fS*, *of c*] **by** (*simp add*: *setprod-1 cong add*: *setprod-cong*) **}**
moreover
{assume *a*: $a \in S$
let $?A = S - \{a\}$
let $?B = \{a\}$
have *eq*: $S = ?A \cup ?B$ **using** *a* **by** *blast*
have *dj*: $?A \cap ?B = \{\}$ **by** *simp*
from *fS* **have** *fAB*: *finite ?A finite ?B* **by** *auto*
have *fA0*:*setprod ?f* $?A$ = *setprod* $(\lambda i.\ c)$ $?A$
apply (*rule setprod-cong*) **by** *auto*
have *cA*: *card* $?A$ = *card* $S - 1$ **using** *fS a* **by** *auto*
have *fA1*: *setprod ?f* $?A = c\ \hat{}$ *card* $?A$ **unfolding** *fA0* **apply** (*rule setprod-constant*) **using** *fS* **by** *auto*
have *setprod ?f* $?A * $ *setprod ?f* $?B$ = *setprod ?f S*
using *setprod-Un-disjoint*[*OF fAB dj*, *of ?f*, *unfolded eq*[*symmetric*]]
by *simp*
then have *?thesis* **using** *a cA*
by (*simp add*: *fA1 ring-simps cong add*: *setprod-cong cong del*: *if-weak-cong*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *setsum-bounded*:
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow f\ i \leq (K::'a::\{semiring\text{-}1, pordered\text{-}ab\text{-}semigroup\text{-}add\})$
shows *setsum f A* $\leq$ *of-nat*(*card A*) $*$ *K*
proof (*cases finite A*)
case *True*
thus *?thesis* **using** *le setsum-mono*[**where** $K{=}A$ **and** $g = \%x.\ K$] **by** *simp*
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

17.6.1 Cardinality of unions

lemma *card-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. $i \neq j$ --> *A i Int A j* = {}) ==> *card* (*UNION I A*) = $(\sum i{\in}I.\ card(A\ i))$
apply (*simp add*: *card-def del*: *setsum-constant*)
apply (*subgoal-tac*
setsum (%*i*. *card* (*A i*)) *I* = *setsum* (%*i*. (*setsum* (%*x*. *1*) (*A i*))) *I*)
apply (*simp add*: *setsum-UN-disjoint del*: *setsum-constant*)
apply (*simp cong*: *setsum-cong*)
done

lemma *card-Union-disjoint*:

finite C ==> (*ALL A*:*C*. *finite A*) ==>
(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* --> *A Int B* = {})
==> *card* (*Union C*) = *setsum card C*
apply (*frule card-UN-disjoint* [*of C id*])
apply (*unfold Union-def id-def*, *assumption*+)
done

17.6.2 Cardinality of image

The image of a finite set can be expressed using *fold-image*.

lemma *image-eq-fold-image*:
finite A ==> *f* ‘ *A* = *fold-image* (*op Un*) (%*x*. {*f x*}) {} *A*
proof (*induct rule*: *finite-induct*)
case *empty* **then show** *?case* **by** *simp*
next
interpret *ab-semigroup-mult op Un*
proof qed *auto*
case *insert*
then show *?case* **by** *simp*
qed

lemma *card-image-le*: *finite A* ==> *card* (*f* ‘ *A*) <= *card A*
apply (*induct set*: *finite*)
apply *simp*
apply (*simp add*: *le-SucI finite-imageI card-insert-if*)
done

lemma *card-image*: *inj-on f A* ==> *card* (*f* ‘ *A*) = *card A*
by(*simp add*:*card-def setsum-reindex o-def del*:*setsum-constant*)

lemma *endo-inj-surj*: *finite A* ==> *f* ‘ *A* ⊆ *A* ==> *inj-on f A* ==> *f* ‘ *A* = *A*
by (*simp add*: *card-seteq card-image*)

lemma *eq-card-imp-inj-on*:
[| *finite A*; *card*(*f* ‘ *A*) = *card A* |] ==> *inj-on f A*
apply (*induct rule*:*finite-induct*)
apply *simp*
apply(*frule card-image-le*[**where** *f* = *f*])
apply(*simp add*:*card-insert-if split*:*if-splits*)
done

lemma *inj-on-iff-eq-card*:
finite A ==> *inj-on f A* = (*card*(*f* ‘ *A*) = *card A*)
by(*blast intro*: *card-image eq-card-imp-inj-on*)

lemma *card-inj-on-le*:
[|*inj-on f A*; *f* ‘ *A* ⊆ *B*; *finite B* |] ==> *card A* ≤ *card B*
apply (*subgoal-tac finite A*)

apply (*force intro*: *card-mono simp add*: *card-image* [*symmetric*])
apply (*blast intro*: *finite-imageD dest*: *finite-subset*)
done

lemma *card-bij-eq*:
[|*inj-on f A*; *f ' A* ⊆ *B*; *inj-on g B*; *g ' B* ⊆ *A*;
finite A; *finite B* |] ==> *card A* = *card B*
by (*auto intro*: *le-anti-sym card-inj-on-le*)

17.6.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
⟦ *finite A*; *ALL a*:*A*. *finite* (*B a*) ⟧
⟹ *card* (*SIGMA x*: *A*. *B x*) = (∑ *a*∈*A*. *card* (*B a*))
by(*simp add*:*card-def setsum-Sigma del*:*setsum-constant*)

lemma *card-cartesian-product*: *card* (*A* <*> *B*) = *card*(*A*) ∗ *card*(*B*)
apply (*cases finite A*)
apply (*cases finite B*)
apply (*auto simp add*: *card-eq-0-iff*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *card-cartesian-product-singleton*: *card*({*x*} <*> *A*) = *card*(*A*)
by (*simp add*: *card-cartesian-product*)

17.6.4 Cardinality of sums

lemma *card-Plus*:
assumes *finite A* **and** *finite B*
shows *card* (*A* <+> *B*) = *card A* + *card B*
proof −
have *Inl*`*A* ∩ *Inr*`*B* = {} **by** *fast*
with *assms* **show** *?thesis*
unfolding *Plus-def*
by (*simp add*: *card-Un-disjoint card-image*)
qed

17.6.5 Cardinality of the Powerset

lemma *card-Pow*: *finite A* ==> *card* (*Pow A*) = *Suc* (*Suc 0*) ^ *card A*
apply (*induct set*: *finite*)
apply (*simp-all add*: *Pow-insert*)
apply (*subst card-Un-disjoint*, *blast*)
apply (*blast intro*: *finite-imageI*, *blast*)
apply (*subgoal-tac inj-on* (*insert x*) (*Pow F*))
apply (*simp add*: *card-image Pow-insert*)
apply (*unfold inj-on-def*)
apply (*blast elim*!: *equalityE*)
done

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma *dvd-partition*:
finite (*Union C*) ==>
ALL c : *C*. *k dvd card c* ==>
(*ALL c1*: *C*. *ALL c2*: *C*. *c1* ≠ *c2* −−> *c1 Int c2* = {}) ==>
k dvd card (*Union C*)
apply(*frule finite-UnionD*)
apply(*rotate-tac* −*1*)
apply (*induct set*: *finite*, *simp-all*, *clarify*)
apply (*subst card-Un-disjoint*)
apply (*auto simp add*: *dvd-add disjoint-eq-subset-Compl*)
done

17.6.6 Relating injectivity and surjectivity

lemma *finite-surj-inj*: *finite*(*A*) ⟹ *A* <= *f*‘*A* ⟹ *inj-on f A*
apply(*rule eq-card-imp-inj-on*, *assumption*)
apply(*frule finite-imageI*)
apply(*drule* (*1*) *card-seteq*)
apply(*erule card-image-le*)
apply *simp*
done

lemma *finite-UNIV-surj-inj*: **fixes** *f* :: *'a* ⇒ *'a*
shows *finite*(*UNIV*:: *'a set*) ⟹ *surj f* ⟹ *inj f*
by (*blast intro*: *finite-surj-inj subset-UNIV dest*:*surj-range*)

lemma *finite-UNIV-inj-surj*: **fixes** *f* :: *'a* ⇒ *'a*
shows *finite*(*UNIV*:: *'a set*) ⟹ *inj f* ⟹ *surj f*
by(*fastsimp simp*:*surj-def dest*!: *endo-inj-surj*)

corollary *infinite-UNIV-nat*: ~*finite*(*UNIV*::*nat set*)
proof
assume *finite*(*UNIV*::*nat set*)
with *finite-UNIV-inj-surj*[*of Suc*]
show *False* **by** *simp* (*blast dest*: *Suc-neq-Zero surjD*)
qed

lemma *infinite-UNIV-char-0*:
¬ *finite* (*UNIV*::*'a*::*semiring-char-0 set*)
proof
assume *finite* (*UNIV*::*'a set*)
with *subset-UNIV* **have** *finite* (*range of-nat*::*'a set*)
by (*rule finite-subset*)
moreover have *inj* (*of-nat*::*nat* ⇒ *'a*)
by (*simp add*: *inj-on-def*)
ultimately have *finite* (*UNIV*::*nat set*)
by (*rule finite-imageD*)
then show *False*

by (*simp add*: *infinite-UNIV-nat*)
qed

17.7 A fold functional for non-empty sets

Does not require start value.

inductive
fold1Set :: ('*a* => '*a* => '*a*) => '*a set* => '*a* => *bool*
for *f* :: '*a* => '*a* => '*a*
where
fold1Set-insertI [*intro*]:
⟦ *fold-graph f a A x*; *a* ∉ *A* ⟧ ⟹ *fold1Set f* (*insert a A*) *x*

constdefs
fold1 :: ('*a* => '*a* => '*a*) => '*a set* => '*a*
fold1 f A == *THE x*. *fold1Set f A x*

lemma *fold1Set-nonempty*:
fold1Set f A x ⟹ *A* ≠ {}
by(*erule fold1Set.cases*, *simp-all*)

inductive-cases *empty-fold1SetE* [*elim*!]: *fold1Set f* {} *x*

inductive-cases *insert-fold1SetE* [*elim*!]: *fold1Set f* (*insert a X*) *x*

lemma *fold1Set-sing* [*iff*]: (*fold1Set f* {*a*} *b*) = (*a* = *b*)
by (*blast intro*: *fold-graph.intros elim*: *fold-graph.cases*)

lemma *fold1-singleton* [*simp*]: *fold1 f* {*a*} = *a*
by (*unfold fold1-def*) *blast*

lemma *finite-nonempty-imp-fold1Set*:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *EX x*. *fold1Set f A x*
apply (*induct A rule*: *finite-induct*)
apply (*auto dest*: *finite-imp-fold-graph* [*of* - *f*])
done

First, some lemmas about *fold-graph*.

context *ab-semigroup-mult*
begin

lemma *fun-left-comm*: *fun-left-comm*(*op* ∗)
by *unfold-locales* (*simp add*: *mult-ac*)

lemma *fold-graph-insert-swap*:
assumes *fold*: *fold-graph times* (*b*::'*a*) *A y* **and** *b* ∉ *A*
shows *fold-graph times z* (*insert b A*) (*z* ∗ *y*)
proof −

```
  interpret fun-left-comm op ∗::'a ⇒ 'a ⇒ 'a by (rule fun-left-comm)
from assms show ?thesis
proof (induct rule: fold-graph.induct)
  case emptyI thus ?case by (force simp add: fold-insert-aux mult-commute)
next
  case (insertI x A y)
    have fold-graph times z (insert x (insert b A)) (x ∗ (z ∗ y))
      using insertI by force  — how does id get unfolded?
    thus ?case by (simp add: insert-commute mult-ac)
qed
qed

lemma fold-graph-permute-diff:
assumes fold: fold-graph times b A x
shows !!a. ⟦a ∈ A; b ∉ A⟧ ⟹ fold-graph times a (insert b (A−{a})) x
using fold
proof (induct rule: fold-graph.induct)
  case emptyI thus ?case by simp
next
  case (insertI x A y)
  have a = x ∨ a ∈ A using insertI by simp
  thus ?case
  proof
    assume a = x
    with insertI show ?thesis
      by (simp add: id-def [symmetric], blast intro: fold-graph-insert-swap)
  next
    assume ainA: a ∈ A
    hence fold-graph times a (insert x (insert b (A − {a}))) (x ∗ y)
      using insertI by force
    moreover
    have insert x (insert b (A − {a})) = insert b (insert x A − {a})
      using ainA insertI by blast
    ultimately show ?thesis by simp
  qed
qed

lemma fold1-eq-fold:
assumes finite A a ∉ A shows fold1 times (insert a A) = fold times a A
proof −
  interpret fun-left-comm op ∗::'a ⇒ 'a ⇒ 'a by (rule fun-left-comm)
  from assms show ?thesis
apply (simp add: fold1-def fold-def)
apply (rule the-equality)
apply (best intro: fold-graph-determ theI dest: finite-imp-fold-graph [of - times])
apply (rule sym, clarify)
apply (case-tac Aa=A)
 apply (best intro: the-equality fold-graph-determ)
apply (subgoal-tac fold-graph times a A x)
```

```
 apply (best intro: the-equality fold-graph-determ)
apply (subgoal-tac insert aa (Aa − {a}) = A)
 prefer 2 apply (blast elim: equalityE)
apply (auto dest: fold-graph-permute-diff [where a=a])
done
qed

lemma nonempty-iff: (A ≠ {}) = (∃x B. A = insert x B & x ∉ B)
apply safe
 apply simp
 apply (drule-tac x=x in spec)
 apply (drule-tac x=A−{x} in spec, auto)
done

lemma fold1-insert:
  assumes nonempty: A ≠ {} and A: finite A x ∉ A
  shows fold1 times (insert x A) = x ∗ fold1 times A
proof −
  interpret fun-left-comm op ∗::'a ⇒ 'a ⇒ 'a by (rule fun-left-comm)
  from nonempty obtain a A' where A = insert a A' & a ~: A'
    by (auto simp add: nonempty-iff)
  with A show ?thesis
    by (simp add: insert-commute [of x] fold1-eq-fold eq-commute)
qed

end

context ab-semigroup-idem-mult
begin

lemma fun-left-comm-idem: fun-left-comm-idem(op ∗)
apply unfold-locales
 apply (simp add: mult-ac)
apply (simp add: mult-idem mult-assoc[symmetric])
done


lemma fold1-insert-idem [simp]:
  assumes nonempty: A ≠ {} and A: finite A
  shows fold1 times (insert x A) = x ∗ fold1 times A
proof −
  interpret fun-left-comm-idem op ∗::'a ⇒ 'a ⇒ 'a
    by (rule fun-left-comm-idem)
  from nonempty obtain a A' where A': A = insert a A' & a ~: A'
    by (auto simp add: nonempty-iff)
  show ?thesis
  proof cases
    assume a = x
    thus ?thesis
```

```
    proof cases
      assume A' = {}
      with prems show ?thesis by (simp add: mult-idem)
    next
      assume A' ≠ {}
      with prems show ?thesis
        by (simp add: fold1-insert mult-assoc [symmetric] mult-idem)
    qed
  next
    assume a ≠ x
    with prems show ?thesis
      by (simp add: insert-commute fold1-eq-fold fold-insert-idem)
  qed
qed

lemma hom-fold1-commute:
assumes hom: !!x y. h (x * y) = h x * h y
and N: finite N N ≠ {} shows h (fold1 times N) = fold1 times (h ' N)
using N proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case (insert n N)
  then have h (fold1 times (insert n N)) = h (n * fold1 times N) by simp
  also have ... = h n * h (fold1 times N) by(rule hom)
  also have h (fold1 times N) = fold1 times (h ' N) by(rule insert)
  also have times (h n) ... = fold1 times (insert (h n) (h ' N))
    using insert by(simp)
  also have insert (h n) (h ' N) = h ' insert n N by simp
  finally show ?case .
qed

end
```

Now the recursion rules for definitions:

```
lemma fold1-singleton-def: g = fold1 f ⟹ g {a} = a
by(simp add:fold1-singleton)

lemma (in ab-semigroup-mult) fold1-insert-def:
  ⟦ g = fold1 times; finite A; x ∉ A; A ≠ {} ⟧ ⟹ g (insert x A) = x * g A
by (simp add:fold1-insert)

lemma (in ab-semigroup-idem-mult) fold1-insert-idem-def:
  ⟦ g = fold1 times; finite A; A ≠ {} ⟧ ⟹ g (insert x A) = x * g A
by simp
```

17.7.1 Determinacy for *fold1Set*

```
declare
  empty-fold-graphE [rule del]  fold-graph.intros [rule del]
```

empty-fold1SetE [*rule del*] *insert-fold1SetE* [*rule del*]
— No more proofs involve these relations.

17.7.2 Lemmas about *fold1*

context *ab-semigroup-mult*
begin

lemma *fold1-Un*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹ *A Int B* = {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
using *A* **by** (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert mult-assoc*)

lemma *fold1-in*:
assumes *A*: *finite* (*A*) *A* ≠ {} **and** *elem*: ⋀*x y*. *x* ∗ *y* ∈ {*x*,*y*}
shows *fold1 times A* ∈ *A*
using *A*
proof (*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **using** *elem* **by** (*force simp add*:*fold1-insert*)
qed

end

lemma (**in** *ab-semigroup-idem-mult*) *fold1-Un2*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
using *A*
proof(*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **by** (*simp add*: *mult-assoc*)
qed

17.7.3 Fold1 in lattices with *inf* and *sup*

As an application of *fold1* we define infimum and supremum in (not necessarily complete!) lattices over (non-empty) sets by means of *fold1*.

context *lower-semilattice*
begin

lemma *ab-semigroup-idem-mult-inf*:
ab-semigroup-idem-mult inf
proof qed (*rule inf-assoc inf-commute inf-idem*)+

lemma *below-fold1-iff*:
assumes *finite A* $A \neq \{\}$
shows $x \leq$ *fold1 inf A* $\longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
proof −
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
show *?thesis* **using** *assms* **by** (*induct rule*: *finite-ne-induct*) *simp-all*
qed

lemma *fold1-belowI*:
assumes *finite A*
and $a \in A$
shows *fold1 inf A* $\leq a$
proof −
from *assms* **have** $A \neq \{\}$ **by** *auto*
from ‹*finite A*› ‹$A \neq \{\}$› ‹$a \in A$› **show** *?thesis*
proof (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
case (*insert x F*)
from *insert*(*5*) **have** $a = x \vee a \in F$ **by** *simp*
thus *?case*
proof
assume $a = x$ **thus** *?thesis* **using** *insert*
by (*simp add*: *mult-ac*)
next
assume $a \in F$
hence *bel*: *fold1 inf F* $\leq a$ **by** (*rule insert*)
have *inf* (*fold1 inf* (*insert x F*)) $a =$ *inf x* (*inf* (*fold1 inf F*) *a*)
using *insert* **by** (*simp add*: *mult-ac*)
also have *inf* (*fold1 inf F*) $a =$ *fold1 inf F*
using *bel* **by** (*auto intro*: *antisym*)
also have *inf x* ... $=$ *fold1 inf* (*insert x F*)
using *insert* **by** (*simp add*: *mult-ac*)
finally have *aux*: *inf* (*fold1 inf* (*insert x F*)) $a =$ *fold1 inf* (*insert x F*) **.**
moreover have *inf* (*fold1 inf* (*insert x F*)) $a \leq a$ **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
qed
qed

end

lemma (**in** *upper-semilattice*) *ab-semigroup-idem-mult-sup*:
ab-semigroup-idem-mult sup
by (*rule lower-semilattice.ab-semigroup-idem-mult-inf*)
(*rule dual-lattice*)

context *lattice*
begin

definition
Inf-fin :: *'a set* ⇒ *'a* ($\bigsqcap_{fin}$- [*900*] *900*)
where
Inf-fin = *fold1 inf*

definition
Sup-fin :: *'a set* ⇒ *'a* ($\bigsqcup_{fin}$- [*900*] *900*)
where
Sup-fin = *fold1 sup*

lemma *Inf-le-Sup* [*simp*]: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ $\bigsqcap_{fin} A \leq \bigsqcup_{fin} A$
apply(*unfold Sup-fin-def Inf-fin-def*)
apply(*subgoal-tac EX a*. *a*:*A*)
prefer *2* **apply** *blast*
apply(*erule exE*)
apply(*rule order-trans*)
apply(*erule* (*1*) *fold1-belowI*)
apply(*erule* (*1*) *lower-semilattice.fold1-belowI* [*OF dual-lattice*])
done

lemma *sup-Inf-absorb* [*simp*]:
finite A ⟹ *a* ∈ *A* ⟹ *sup a* ($\bigsqcap_{fin} A$) = *a*
apply(*subst sup-commute*)
apply(*simp add*: *Inf-fin-def sup-absorb2 fold1-belowI*)
done

lemma *inf-Sup-absorb* [*simp*]:
finite A ⟹ *a* ∈ *A* ⟹ *inf a* ($\bigsqcup_{fin} A$) = *a*
by (*simp add*: *Sup-fin-def inf-absorb1*
lower-semilattice.fold1-belowI [*OF dual-lattice*])

end

context *distrib-lattice*
begin

lemma *sup-Inf1-distrib*:
assumes *finite A*
and *A* ≠ {}
shows *sup x* ($\bigsqcap_{fin} A$) = $\bigsqcap_{fin}$ {*sup x a*|*a*. *a* ∈ *A*}
proof −
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
from *assms* **show** *?thesis*
by (*simp add*: *Inf-fin-def image-def*

hom-fold1-commute [**where** *h=sup x*, *OF sup-inf-distrib1*])
(*rule arg-cong* [**where** *f=fold1 inf*], *blast*)
qed

lemma *sup-Inf2-distrib*:
assumes *A*: *finite A A* $\neq$ {} **and** *B*: *finite B B* $\neq$ {}
shows *sup* ($\bigsqcap_{fin} A$) ($\bigsqcap_{fin} B$) = $\bigsqcap_{fin}$ {*sup a b*|*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*}
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by (*simp add*: *sup-Inf1-distrib* [*OF B*] *fold1-singleton-def* [*OF Inf-fin-def*])
next
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
case (*insert x A*)
have *finB*: *finite* {*sup x b* |*b*. *b* $\in$ *B*}
by(*rule finite-surj*[**where** *f* = *sup x*, *OF B*(*1*)], *auto*)
have *finAB*: *finite* {*sup a b* |*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*}
proof −
have {*sup a b* |*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*} = (*UN a*:*A*. *UN b*:*B*. {*sup a b*})
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: {*sup a b* |*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*} $\neq$ {} **using** *insert B* **by** *blast*
have *sup* ($\bigsqcap_{fin}$(*insert x A*)) ($\bigsqcap_{fin} B$) = *sup* (*inf x* ($\bigsqcap_{fin} A$)) ($\bigsqcap_{fin} B$)
using *insert* **by** (*simp add*: *fold1-insert-idem-def* [*OF Inf-fin-def*])
also have ... = *inf* (*sup x* ($\bigsqcap_{fin} B$)) (*sup* ($\bigsqcap_{fin} A$) ($\bigsqcap_{fin} B$)) **by**(*rule sup-inf-distrib2*)
also have ... = *inf* ($\bigsqcap_{fin}$ {*sup x b*|*b*. *b* $\in$ *B*}) ($\bigsqcap_{fin}$ {*sup a b*|*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*})
using *insert* **by**(*simp add*:*sup-Inf1-distrib*[*OF B*])
also have ... = $\bigsqcap_{fin}$({*sup x b* |*b*. *b* $\in$ *B*} $\cup$ {*sup a b* |*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*})
(**is** - = $\bigsqcap_{fin}$ *?M*)
using *B insert*
by (*simp add*: *Inf-fin-def fold1-Un2* [*OF finB - finAB ne*])
also have *?M* = {*sup a b* |*a b*. *a* $\in$ *insert x A* $\wedge$ *b* $\in$ *B*}
by *blast*
finally show *?case* **.**
qed

lemma *inf-Sup1-distrib*:
assumes *finite A* **and** *A* $\neq$ {}
shows *inf x* ($\bigsqcup_{fin} A$) = $\bigsqcup_{fin}$ {*inf x a*|*a*. *a* $\in$ *A*}
proof −
interpret *ab-semigroup-idem-mult sup*
by (*rule ab-semigroup-idem-mult-sup*)
from *assms* **show** *?thesis*
by (*simp add*: *Sup-fin-def image-def hom-fold1-commute* [**where** *h=inf x*, *OF inf-sup-distrib1*])
(*rule arg-cong* [**where** *f=fold1 sup*], *blast*)
qed

lemma *inf-Sup2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows *inf* $(\bigsqcup_{fin} A)$ $(\bigsqcup_{fin} B) = \bigsqcup_{fin}\{inf\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by(*simp add*: *inf-Sup1-distrib* [*OF B*] *fold1-singleton-def* [*OF Sup-fin-def*])
next
case (*insert x A*)
have *finB*: *finite* $\{inf\ x\ b\ | b.\ b \in B\}$
by(*rule finite-surj*[**where** $f = \%b.\ inf\ x\ b$, *OF B*(*1*)], *auto*)
have *finAB*: *finite* $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\}$
proof −
have $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\} = (UN\ a{:}A.\ UN\ b{:}B.\ \{inf\ a\ b\})$
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B* **by** *blast*
interpret *ab-semigroup-idem-mult sup*
by (*rule ab-semigroup-idem-mult-sup*)
have *inf* $(\bigsqcup_{fin}(insert\ x\ A))$ $(\bigsqcup_{fin} B) = inf\ (sup\ x\ (\bigsqcup_{fin} A))\ (\bigsqcup_{fin} B)$
using *insert* **by** (*simp add*: *fold1-insert-idem-def* [*OF Sup-fin-def*])
also have $\ldots = sup\ (inf\ x\ (\bigsqcup_{fin} B))\ (inf\ (\bigsqcup_{fin} A)\ (\bigsqcup_{fin} B))$ **by**(*rule inf-sup-distrib2*)
also have $\ldots = sup\ (\bigsqcup_{fin}\{inf\ x\ b | b.\ b \in B\})\ (\bigsqcup_{fin}\{inf\ a\ b | a\ b.\ a \in A \wedge b \in B\})$
using *insert* **by**(*simp add*:*inf-Sup1-distrib*[*OF B*])
also have $\ldots = \bigsqcup_{fin}(\{inf\ x\ b\ | b.\ b \in B\} \cup \{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\})$
(**is** - $= \bigsqcup_{fin} ?M$)
using *B insert*
by (*simp add*: *Sup-fin-def fold1-Un2* [*OF finB* - *finAB ne*])
also have $?M = \{inf\ a\ b\ | a\ b.\ a \in insert\ x\ A \wedge b \in B\}$
by *blast*
finally show *?case* **.**
qed

end

context *complete-lattice*
begin

Coincidence on finite sets in complete lattices:

lemma *Inf-fin-Inf*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcap_{fin} A = Inf\ A$
proof −
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
from *assms* **show** *?thesis*
unfolding *Inf-fin-def* **by** (*induct A set*: *finite*)

(*simp-all add*: *Inf-insert-simp*)
qed

lemma *Sup-fin-Sup*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcup_{fin} A = Sup\ A$
proof −
interpret *ab-semigroup-idem-mult sup*
by (*rule ab-semigroup-idem-mult-sup*)
from *assms* **show** *?thesis*
unfolding *Sup-fin-def* **by** (*induct A set*: *finite*)
(*simp-all add*: *Sup-insert-simp*)
qed

end

17.7.4 Fold1 in linear orders with *min* and *max*

As an application of *fold1* we define minimum and maximum in (not necessarily complete!) linear orders over (non-empty) sets by means of *fold1*.

context *linorder*
begin

lemma *ab-semigroup-idem-mult-min*:
ab-semigroup-idem-mult min
proof qed (*auto simp add*: *min-def*)

lemma *ab-semigroup-idem-mult-max*:
ab-semigroup-idem-mult max
proof qed (*auto simp add*: *max-def*)

lemma *min-lattice*:
lower-semilattice (*op* ≤) (*op* <) *min*
proof qed (*auto simp add*: *min-def*)

lemma *max-lattice*:
lower-semilattice (*op* ≥) (*op* >) *max*
proof qed (*auto simp add*: *max-def*)

lemma *dual-max*:
ord.max (*op* ≥) = *min*
by (*auto simp add*: *ord.max-def-raw min-def-raw expand-fun-eq*)

lemma *dual-min*:
ord.min (*op* ≥) = *max*
by (*auto simp add*: *ord.min-def-raw max-def-raw expand-fun-eq*)

lemma *strict-below-fold1-iff*:
assumes *finite A* **and** $A \neq \{\}$

shows $x < \mathit{fold1}\ \mathit{min}\ A \longleftrightarrow (\forall\, a{\in}A.\ x < a)$
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert*)
qed

lemma *fold1-below-iff*:
assumes *finite* A **and** $A \neq \{\}$
shows $\mathit{fold1}\ \mathit{min}\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert min-le-iff-disj*)
qed

lemma *fold1-strict-below-iff*:
assumes *finite* A **and** $A \neq \{\}$
shows $\mathit{fold1}\ \mathit{min}\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert min-less-iff-disj*)
qed

lemma *fold1-antimono*:
assumes $A \neq \{\}$ **and** $A \subseteq B$ **and** *finite* B
shows $\mathit{fold1}\ \mathit{min}\ B \leq \mathit{fold1}\ \mathit{min}\ A$
proof *cases*
assume $A = B$ **thus** *?thesis* **by** *simp*
next
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
assume $A \neq B$
have B: $B = A \cup (B{-}A)$ **using** ‹$A \subseteq B$› **by** *blast*
have $\mathit{fold1}\ \mathit{min}\ B = \mathit{fold1}\ \mathit{min}\ (A \cup (B{-}A))$ **by**(*subst* B)(*rule refl*)
also have $\ldots = \mathit{min}\ (\mathit{fold1}\ \mathit{min}\ A)\ (\mathit{fold1}\ \mathit{min}\ (B{-}A))$
proof −
have *finite* A **by**(*rule finite-subset*[*OF* ‹$A \subseteq B$› ‹*finite* B›])
moreover have *finite*$(B{-}A)$ **by**(*rule finite-Diff*[*OF* ‹*finite* B›])
moreover have $(B{-}A) \neq \{\}$ **using** *prems* **by** *blast*
moreover have $A\ \mathit{Int}\ (B{-}A) = \{\}$ **using** *prems* **by** *blast*
ultimately show *?thesis* **using** ‹$A \neq \{\}$› **by** (*rule-tac fold1-Un*)

qed
also have ... $\leq$ *fold1 min A* **by** (*simp add*: *min-le-iff-disj*)
finally show *?thesis* **.**
qed

definition
Min :: *'a set* $\Rightarrow$ *'a*
where
Min = *fold1 min*

definition
Max :: *'a set* $\Rightarrow$ *'a*
where
Max = *fold1 max*

lemmas *Min-singleton* [*simp*] = *fold1-singleton-def* [*OF Min-def*]
lemmas *Max-singleton* [*simp*] = *fold1-singleton-def* [*OF Max-def*]

lemma *Min-insert* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows *Min* (*insert x A*) = *min x* (*Min A*)
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis* **by** (*rule fold1-insert-idem-def* [*OF Min-def*])
qed

lemma *Max-insert* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows *Max* (*insert x A*) = *max x* (*Max A*)
proof −
interpret *ab-semigroup-idem-mult max*
by (*rule ab-semigroup-idem-mult-max*)
from *assms* **show** *?thesis* **by** (*rule fold1-insert-idem-def* [*OF Max-def*])
qed

lemma *Min-in* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows *Min A* $\in$ *A*
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms fold1-in* **show** *?thesis* **by** (*fastsimp simp*: *Min-def min-def*)
qed

lemma *Max-in* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows *Max A* $\in$ *A*
proof −

interpret *ab-semigroup-idem-mult max*
by (*rule ab-semigroup-idem-mult-max*)
from *assms fold1-in* [*of A*] **show** *?thesis* **by** (*fastsimp simp*: *Max-def max-def*)
qed

lemma *Min-Un*:
assumes *finite A* **and** $A \neq \{\}$ **and** *finite B* **and** $B \neq \{\}$
shows *Min* $(A \cup B)$ = *min* (*Min A*) (*Min B*)
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-Un2*)
qed

lemma *Max-Un*:
assumes *finite A* **and** $A \neq \{\}$ **and** *finite B* **and** $B \neq \{\}$
shows *Max* $(A \cup B)$ = *max* (*Max A*) (*Max B*)
proof −
interpret *ab-semigroup-idem-mult max*
by (*rule ab-semigroup-idem-mult-max*)
from *assms* **show** *?thesis*
by (*simp add*: *Max-def fold1-Un2*)
qed

lemma *hom-Min-commute*:
assumes $\bigwedge x\ y.\ h\ (min\ x\ y) = min\ (h\ x)\ (h\ y)$
and *finite N* **and** $N \neq \{\}$
shows *h* (*Min N*) = *Min* (*h* ‘ *N*)
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def hom-fold1-commute*)
qed

lemma *hom-Max-commute*:
assumes $\bigwedge x\ y.\ h\ (max\ x\ y) = max\ (h\ x)\ (h\ y)$
and *finite N* **and** $N \neq \{\}$
shows *h* (*Max N*) = *Max* (*h* ‘ *N*)
proof −
interpret *ab-semigroup-idem-mult max*
by (*rule ab-semigroup-idem-mult-max*)
from *assms* **show** *?thesis*
by (*simp add*: *Max-def hom-fold1-commute* [*of h*])
qed

lemma *Min-le* [*simp*]:
assumes *finite A* **and** $x \in A$

shows *Min A* ≤ *x*
proof −
interpret *lower-semilattice op* ≤ *op* < *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def fold1-belowI*)
qed

lemma *Max-ge* [*simp*]:
assumes *finite A* **and** *x* ∈ *A*
shows *x* ≤ *Max A*
proof −
interpret *lower-semilattice op* ≥ *op* > *max*
by (*rule max-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Max-def fold1-belowI*)
qed

lemma *Min-ge-iff* [*simp*, *noatp*]:
assumes *finite A* **and** *A* ≠ {}
shows *x* ≤ *Min A* ⟷ (∀ *a*∈*A*. *x* ≤ *a*)
proof −
interpret *lower-semilattice op* ≤ *op* < *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def below-fold1-iff*)
qed

lemma *Max-le-iff* [*simp*, *noatp*]:
assumes *finite A* **and** *A* ≠ {}
shows *Max A* ≤ *x* ⟷ (∀ *a*∈*A*. *a* ≤ *x*)
proof −
interpret *lower-semilattice op* ≥ *op* > *max*
by (*rule max-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Max-def below-fold1-iff*)
qed

lemma *Min-gr-iff* [*simp*, *noatp*]:
assumes *finite A* **and** *A* ≠ {}
shows *x* < *Min A* ⟷ (∀ *a*∈*A*. *x* < *a*)
proof −
interpret *lower-semilattice op* ≤ *op* < *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def strict-below-fold1-iff*)
qed

lemma *Max-less-iff* [*simp*, *noatp*]:
assumes *finite A* **and** *A* ≠ {}
shows *Max A* < *x* ⟷ (∀ *a*∈*A*. *a* < *x*)
proof −
note *Max* = *Max-def*
interpret *linorder op* ≥ *op* >

by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max strict-below-fold1-iff* [*folded dual-max*])
qed

lemma *Min-le-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
proof −
interpret *lower-semilattice op* $\leq$ *op* $<$ *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-below-iff*)
qed

lemma *Max-ge-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x \leq Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x \leq a)$
proof −
note *Max* = *Max-def*
interpret *linorder op* $\geq$ *op* $>$
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-below-iff* [*folded dual-max*])
qed

lemma *Min-less-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
proof −
interpret *lower-semilattice op* $\leq$ *op* $<$ *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-strict-below-iff*)
qed

lemma *Max-gr-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x < Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x < a)$
proof −
note *Max* = *Max-def*
interpret *linorder op* $\geq$ *op* $>$
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-strict-below-iff* [*folded dual-max*])
qed

lemma *Min-eqI*:
assumes *finite A*

assumes $\bigwedge y.\ y \in A \Longrightarrow y \geq x$
and $x \in A$
shows $Min\ A = x$
proof (*rule antisym*)
from ‹$x \in A$› **have** $A \neq \{\}$ **by** *auto*
with *assms* **show** $Min\ A \geq x$ **by** *simp*
next
from *assms* **show** $x \geq Min\ A$ **by** *simp*
qed

lemma *Max-eqI*:
assumes *finite A*
assumes $\bigwedge y.\ y \in A \Longrightarrow y \leq x$
and $x \in A$
shows $Max\ A = x$
proof (*rule antisym*)
from ‹$x \in A$› **have** $A \neq \{\}$ **by** *auto*
with *assms* **show** $Max\ A \leq x$ **by** *simp*
next
from *assms* **show** $x \leq Max\ A$ **by** *simp*
qed

lemma *Min-antimono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Min\ N \leq Min\ M$
proof −
interpret *distrib-lattice op* $\leq$ *op* $<$ *min max*
by (*rule distrib-lattice-min-max*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def fold1-antimono*)
qed

lemma *Max-mono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Max\ M \leq Max\ N$
proof −
note $Max = Max\text{-}def$
interpret *linorder op* $\geq$ *op* $>$
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-antimono* [*folded dual-max*])
qed

lemma *finite-linorder-induct*[*consumes 1*, *case-names empty insert*]:
finite A $\Longrightarrow$ *P* $\{\}$ $\Longrightarrow$
(!!*A b*. *finite A* $\Longrightarrow$ *ALL a*:*A*. *a* $<$ *b* $\Longrightarrow$ *P A* $\Longrightarrow$ *P*(*insert b A*))
$\Longrightarrow$ *P A*
proof (*induct A rule*: *measure-induct-rule*[**where** *f*=*card*])
fix *A* :: *'a set*
assume *IH*: !! *B*. *card B* $<$ *card A* $\Longrightarrow$ *finite B* $\Longrightarrow$ *P* $\{\}$ $\Longrightarrow$

```
            (!!A b. finite A ⟹ (∀ a∈A. a<b) ⟹ P A ⟹ P (insert b A))
             ⟹ P B
  and finite A and P {}
  and step: !!A b. ⟦finite A; ∀ a∈A. a < b; P A⟧ ⟹ P (insert b A)
  show P A
  proof (cases A = {})
    assume A = {} thus P A using ‹P {}› by simp
  next
    let ?B = A - {Max A} let ?A = insert (Max A) ?B
    assume A ≠ {}
    with ‹finite A› have Max A : A by auto
    hence A: ?A = A using insert-Diff-single insert-absorb by auto
    note card-Diff1-less[OF ‹finite A› ‹Max A : A›]
    moreover have finite ?B using ‹finite A› by simp
    ultimately have P ?B using ‹P {}› step IH by blast
    moreover have ∀ a∈?B. a < Max A
      using Max-ge [OF ‹finite A›] by fastsimp
    ultimately show P A
      using A insert-Diff-single step[OF ‹finite ?B›] by fastsimp
  qed
qed

end

context ordered-ab-semigroup-add
begin

lemma add-Min-commute:
  fixes k
  assumes finite N and N ≠ {}
  shows k + Min N = Min {k + m | m. m ∈ N}
proof -
  have ⋀x y. k + min x y = min (k + x) (k + y)
    by (simp add: min-def not-le)
      (blast intro: antisym less-imp-le add-left-mono)
  with assms show ?thesis
    using hom-Min-commute [of plus k N]
    by simp (blast intro: arg-cong [where f = Min])
qed

lemma add-Max-commute:
  fixes k
  assumes finite N and N ≠ {}
  shows k + Max N = Max {k + m | m. m ∈ N}
proof -
  have ⋀x y. k + max x y = max (k + x) (k + y)
    by (simp add: max-def not-le)
      (blast intro: antisym less-imp-le add-left-mono)
  with assms show ?thesis
```

using *hom-Max-commute* [*of plus k N*]
by *simp* (*blast intro*: *arg-cong* [**where** *f* = *Max*])
qed

end

end

18 Relation: Relations

theory *Relation*
imports *Datatype Finite-Set*
begin

18.1 Definitions

definition
converse :: ('*a* * '*b*) *set* => ('*b* * '*a*) *set*
((-^-1) [*1000*] *999*) **where**
r^-1 == {(*y*, *x*). (*x*, *y*) : *r*}

notation (*xsymbols*)
converse ((-$^{-1}$) [*1000*] *999*)

definition
rel-comp :: [('*b* * '*c*) *set*, ('*a* * '*b*) *set*] => ('*a* * '*c*) *set*
(**infixr** *O* *75*) **where**
r O s == {(*x*,*z*). *EX y*. (*x*, *y*) : *s* & (*y*, *z*) : *r*}

definition
Image :: [('*a* * '*b*) *set*, '*a set*] => '*b set*
(**infixl** `` *90*) **where**
r `` *s* == {*y*. *EX x*:*s*. (*x*,*y*):*r*}

definition
Id :: ('*a* * '*a*) *set* **where** — the identity relation
Id == {*p*. *EX x*. *p* = (*x*,*x*)}

definition
Id-on :: '*a set* => ('*a* * '*a*) *set* **where** — diagonal: identity over a set
Id-on A == $\bigcup$*x*∈*A*. {(*x*,*x*)}

definition
Domain :: ('*a* * '*b*) *set* => '*a set* **where**
Domain r == {*x*. *EX y*. (*x*,*y*):*r*}

definition
Range :: ('*a* * '*b*) *set* => '*b set* **where**

Range r == Domain(r^−1)

definition
Field :: *('a * 'a) set => 'a set* **where**
Field r == Domain r ∪ Range r

definition
refl-on :: *['a set, ('a * 'a) set] => bool* **where** — reflexivity over a set
refl-on A r == r ⊆ A × A & (ALL x: A. (x,x) : r)

abbreviation
refl :: *('a * 'a) set => bool* **where** — reflexivity over a type
refl == refl-on UNIV

definition
sym :: *('a * 'a) set => bool* **where** — symmetry predicate
sym r == ALL x y. (x,y): r −−> (y,x): r

definition
antisym :: *('a * 'a) set => bool* **where** — antisymmetry predicate
antisym r == ALL x y. (x,y):r −−> (y,x):r −−> x=y

definition
trans :: *('a * 'a) set => bool* **where** — transitivity predicate
trans r == (ALL x y z. (x,y):r −−> (y,z):r −−> (x,z):r)

definition
irrefl :: *('a * 'a) set => bool* **where**
irrefl r ≡ ∀ x. (x,x) ∉ r

definition
total-on :: *'a set => ('a * 'a) set => bool* **where**
total-on A r ≡ ∀ x∈A.∀ y∈A. x≠y ⟶ (x,y)∈r ∨ (y,x)∈r

abbreviation *total ≡ total-on UNIV*

definition
single-valued :: *('a * 'b) set => bool* **where**
single-valued r == ALL x y. (x,y):r −−> (ALL z. (x,z):r −−> y=z)

definition
inv-image :: *('b * 'b) set => ('a => 'b) => ('a * 'a) set* **where**
inv-image r f == {(x, y). (f x, f y) : r}

18.2 The identity relation

lemma *IdI* [*intro*]: *(a, a) : Id*
by (*simp add*: *Id-def*)

lemma *IdE* [*elim*!]: $p : Id \Longrightarrow (!!x.\ p = (x,\ x) \Longrightarrow P) \Longrightarrow P$
by (*unfold Id-def*) (*iprover elim*: *CollectE*)

lemma *pair-in-Id-conv* [*iff*]: $((a,\ b) : Id) = (a = b)$
by (*unfold Id-def*) *blast*

lemma *refl-Id*: *refl Id*
by (*simp add*: *refl-on-def*)

lemma *antisym-Id*: *antisym Id*
— A strange result, since *Id* is also symmetric.
by (*simp add*: *antisym-def*)

lemma *sym-Id*: *sym Id*
by (*simp add*: *sym-def*)

lemma *trans-Id*: *trans Id*
by (*simp add*: *trans-def*)

18.3 Diagonal: identity over a set

lemma *Id-on-empty* [*simp*]: *Id-on* $\{\} = \{\}$
by (*simp add*: *Id-on-def*)

lemma *Id-on-eqI*: $a = b \Longrightarrow a : A \Longrightarrow (a,\ b) :$ *Id-on* A
by (*simp add*: *Id-on-def*)

lemma *Id-onI* [*intro*!,*noatp*]: $a : A \Longrightarrow (a,\ a) :$ *Id-on* A
by (*rule Id-on-eqI*) (*rule refl*)

lemma *Id-onE* [*elim*!]:
$c :$ *Id-on* $A \Longrightarrow (!!x.\ x : A \Longrightarrow c = (x,\ x) \Longrightarrow P) \Longrightarrow P$
— The general elimination rule.
by (*unfold Id-on-def*) (*iprover elim*!: *UN-E singletonE*)

lemma *Id-on-iff*: $((x,\ y) :$ *Id-on* $A) = (x = y\ \&\ x : A)$
by *blast*

lemma *Id-on-subset-Times*: *Id-on* $A \subseteq A \times A$
by *blast*

18.4 Composition of two relations

lemma *rel-compI* [*intro*]:
$(a,\ b) : s \Longrightarrow (b,\ c) : r \Longrightarrow (a,\ c) : r\ O\ s$
by (*unfold rel-comp-def*) *blast*

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s \Longrightarrow$
$(!!x\ y\ z.\ xz = (x,\ z) \Longrightarrow (x,\ y) : s \Longrightarrow (y,\ z) : r \Longrightarrow P) \Longrightarrow P$
by (*unfold rel-comp-def*) (*iprover elim*!: *CollectE splitE exE conjE*)

lemma *rel-compEpair*:
$(a,\ c) : r\ O\ s \Longrightarrow (!!y.\ (a,\ y) : s \Longrightarrow (y,\ c) : r \Longrightarrow P) \Longrightarrow P$
by (*iprover elim*: *rel-compE Pair-inject ssubst*)

lemma *R-O-Id* [*simp*]: $R\ O\ Id = R$
by *fast*

lemma *Id-O-R* [*simp*]: $Id\ O\ R = R$
by *fast*

lemma *rel-comp-empty1*[*simp*]: $\{\}\ O\ R = \{\}$
by *blast*

lemma *rel-comp-empty2*[*simp*]: $R\ O\ \{\} = \{\}$
by *blast*

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
by *blast*

lemma *trans-O-subset*: *trans* $r \Longrightarrow r\ O\ r \subseteq r$
by (*unfold trans-def*) *blast*

lemma *rel-comp-mono*: $r' \subseteq r \Longrightarrow s' \subseteq s \Longrightarrow (r'\ O\ s') \subseteq (r\ O\ s)$
by *blast*

lemma *rel-comp-subset-Sigma*:
$s \subseteq A \times B \Longrightarrow r \subseteq B \times C \Longrightarrow (r\ O\ s) \subseteq A \times C$
by *blast*

lemma *rel-comp-distrib*[*simp*]: $R\ O\ (S \cup T) = (R\ O\ S) \cup (R\ O\ T)$
by *auto*

lemma *rel-comp-distrib2*[*simp*]: $(S \cup T)\ O\ R = (S\ O\ R) \cup (T\ O\ R)$
by *auto*

18.5 Reflexivity

lemma *refl-onI*: $r \subseteq A \times A \Longrightarrow (!!x.\ x : A \Longrightarrow (x,\ x) : r) \Longrightarrow$ *refl-on* $A\ r$
by (*unfold refl-on-def*) (*iprover intro*!: *ballI*)

lemma *refl-onD*: *refl-on* $A\ r \Longrightarrow a : A \Longrightarrow (a,\ a) : r$
by (*unfold refl-on-def*) *blast*

lemma *refl-onD1*: *refl-on* $A\ r \Longrightarrow (x,\ y) : r \Longrightarrow x : A$
by (*unfold refl-on-def*) *blast*

lemma *refl-onD2*: *refl-on* $A\ r \Longrightarrow (x,\ y) : r \Longrightarrow y : A$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-Int*: *refl-on A r ==> refl-on B s ==> refl-on* $(A \cap B)$ $(r \cap s)$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-Un*: *refl-on A r ==> refl-on B s ==> refl-on* $(A \cup B)$ $(r \cup s)$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-INTER*:
ALL x:S. refl-on (A x) (r x) ==> refl-on (INTER S A) (INTER S r)
by (*unfold refl-on-def*) *fast*

lemma *refl-on-UNION*:
ALL x:S. refl-on (A x) (r x) $\Longrightarrow$ *refl-on (UNION S A) (UNION S r)*
by (*unfold refl-on-def*) *blast*

lemma *refl-on-empty*[*simp*]: *refl-on* {} {}
by(*simp add*:*refl-on-def*)

lemma *refl-on-Id-on*: *refl-on A* (*Id-on A*)
by (*rule refl-onI* [*OF Id-on-subset-Times Id-onI*])

18.6 Antisymmetry

lemma *antisymI*:
(!!*x y*. (*x*, *y*) : *r* ==> (*y*, *x*) : *r* ==> *x*=*y*) ==> *antisym r*
by (*unfold antisym-def*) *iprover*

lemma *antisymD*: *antisym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r* ==> *a* = *b*
by (*unfold antisym-def*) *iprover*

lemma *antisym-subset*: $r \subseteq s$ ==> *antisym s* ==> *antisym r*
by (*unfold antisym-def*) *blast*

lemma *antisym-empty* [*simp*]: *antisym* {}
by (*unfold antisym-def*) *blast*

lemma *antisym-Id-on* [*simp*]: *antisym* (*Id-on A*)
by (*unfold antisym-def*) *blast*

18.7 Symmetry

lemma *symI*: (!!*a b*. (*a*, *b*) : *r* ==> (*b*, *a*) : *r*) ==> *sym r*
by (*unfold sym-def*) *iprover*

lemma *symD*: *sym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r*
by (*unfold sym-def*, *blast*)

lemma *sym-Int*: *sym r* ==> *sym s* ==> *sym* $(r \cap s)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-Un*: *sym r ==> sym s ==> sym* $(r \cup s)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-INTER*: *ALL x*:*S*. *sym* $(r\ x)$ *==> sym* $(INTER\ S\ r)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-UNION*: *ALL x*:*S*. *sym* $(r\ x)$ *==> sym* $(UNION\ S\ r)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-Id-on* [*simp*]: *sym* (*Id-on A*)
by (*rule symI*) *clarify*

18.8 Transitivity

lemma *transI*:
(!!*x y z*. $(x, y) : r$ *==>* $(y, z) : r$ *==>* $(x, z) : r$) *==> trans r*
by (*unfold trans-def*) *iprover*

lemma *transD*: *trans r ==>* $(a, b) : r$ *==>* $(b, c) : r$ *==>* $(a, c) : r$
by (*unfold trans-def*) *iprover*

lemma *trans-Int*: *trans r ==> trans s ==> trans* $(r \cap s)$
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-INTER*: *ALL x*:*S*. *trans* $(r\ x)$ *==> trans* $(INTER\ S\ r)$
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-Id-on* [*simp*]: *trans* (*Id-on A*)
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-diff-Id*: *trans r* $\Longrightarrow$ *antisym r* $\Longrightarrow$ *trans* $(r - Id)$
unfolding *antisym-def trans-def* **by** *blast*

18.9 Irreflexivity

lemma *irrefl-diff-Id*[*simp*]: *irrefl*$(r - Id)$
by(*simp add*:*irrefl-def*)

18.10 Totality

lemma *total-on-empty*[*simp*]: *total-on* $\{\}$ *r*
by(*simp add*:*total-on-def*)

lemma *total-on-diff-Id*[*simp*]: *total-on A* $(r - Id)$ = *total-on A r*
by(*simp add*: *total-on-def*)

18.11 Converse

lemma *converse-iff* [*iff*]: $((a,b): r\hat{}-1) = ((b,a) : r)$
by (*simp add*: *converse-def*)

lemma *converseI*[*sym*]: $(a,\ b) : r \Longrightarrow (b,\ a) : r\hat{}-1$
by (*simp add*: *converse-def*)

lemma *converseD*[*sym*]: $(a,b) : r\hat{}-1 \Longrightarrow (b,\ a) : r$
by (*simp add*: *converse-def*)

lemma *converseE* [*elim*!]:
$yx : r\hat{}-1 \Longrightarrow (!!x\ y.\ yx = (y,\ x) \Longrightarrow (x,\ y) : r \Longrightarrow P) \Longrightarrow P$
— More general than *converseD*, as it "splits" the member of the relation.
by (*unfold converse-def*) (*iprover elim*!: *CollectE splitE bexE*)

lemma *converse-converse* [*simp*]: $(r\hat{}-1)\hat{}-1 = r$
by (*unfold converse-def*) *blast*

lemma *converse-rel-comp*: $(r\ O\ s)\hat{}-1 = s\hat{}-1\ O\ r\hat{}-1$
by *blast*

lemma *converse-Int*: $(r \cap s)\hat{}-1 = r\hat{}-1 \cap s\hat{}-1$
by *blast*

lemma *converse-Un*: $(r \cup s)\hat{}-1 = r\hat{}-1 \cup s\hat{}-1$
by *blast*

lemma *converse-INTER*: $(INTER\ S\ r)\hat{}-1 = (INT\ x{:}S.\ (r\ x)\hat{}-1)$
by *fast*

lemma *converse-UNION*: $(UNION\ S\ r)\hat{}-1 = (UN\ x{:}S.\ (r\ x)\hat{}-1)$
by *blast*

lemma *converse-Id* [*simp*]: $Id\hat{}-1 = Id$
by *blast*

lemma *converse-Id-on* [*simp*]: $(Id\text{-}on\ A)\hat{}-1 = Id\text{-}on\ A$
by *blast*

lemma *refl-on-converse* [*simp*]: $refl\text{-}on\ A\ (converse\ r) = refl\text{-}on\ A\ r$
by (*unfold refl-on-def*) *auto*

lemma *sym-converse* [*simp*]: $sym\ (converse\ r) = sym\ r$
by (*unfold sym-def*) *blast*

lemma *antisym-converse* [*simp*]: $antisym\ (converse\ r) = antisym\ r$
by (*unfold antisym-def*) *blast*

lemma *trans-converse* [*simp*]: $trans\ (converse\ r) = trans\ r$
by (*unfold trans-def*) *blast*

lemma *sym-conv-converse-eq*: $sym\ r = (r\hat{}-1 = r)$

by (*unfold sym-def*) *fast*

lemma *sym-Un-converse*: *sym* (*r* ∪ *r*^−1)
by (*unfold sym-def*) *blast*

lemma *sym-Int-converse*: *sym* (*r* ∩ *r*^−1)
by (*unfold sym-def*) *blast*

lemma *total-on-converse*[*simp*]: *total-on A* (*r*^−1) = *total-on A r*
by (*auto simp*: *total-on-def*)

18.12 Domain

declare *Domain-def* [*noatp*]

lemma *Domain-iff*: (*a* : *Domain r*) = (*EX y*. (*a*, *y*) : *r*)
by (*unfold Domain-def*) *blast*

lemma *DomainI* [*intro*]: (*a*, *b*) : *r* ==> *a* : *Domain r*
by (*iprover intro*!: *iffD2* [*OF Domain-iff*])

lemma *DomainE* [*elim*!]:
a : *Domain r* ==> (!!*y*. (*a*, *y*) : *r* ==> *P*) ==> *P*
by (*iprover dest*!: *iffD1* [*OF Domain-iff*])

lemma *Domain-empty* [*simp*]: *Domain* {} = {}
by *blast*

lemma *Domain-insert*: *Domain* (*insert* (*a*, *b*) *r*) = *insert a* (*Domain r*)
by *blast*

lemma *Domain-Id* [*simp*]: *Domain Id* = *UNIV*
by *blast*

lemma *Domain-Id-on* [*simp*]: *Domain* (*Id-on A*) = *A*
by *blast*

lemma *Domain-Un-eq*: *Domain*(*A* ∪ *B*) = *Domain*(*A*) ∪ *Domain*(*B*)
by *blast*

lemma *Domain-Int-subset*: *Domain*(*A* ∩ *B*) ⊆ *Domain*(*A*) ∩ *Domain*(*B*)
by *blast*

lemma *Domain-Diff-subset*: *Domain*(*A*) − *Domain*(*B*) ⊆ *Domain*(*A* − *B*)
by *blast*

lemma *Domain-Union*: *Domain* (*Union S*) = (⋃ *A*∈*S*. *Domain A*)
by *blast*

lemma *Domain-converse*[*simp*]: *Domain*(*r*^−1) = *Range r*
by(*auto simp*:*Range-def*)

lemma *Domain-mono*: *r* ⊆ *s* ==> *Domain r* ⊆ *Domain s*
by *blast*

lemma *fst-eq-Domain*: *fst* ‘ *R* = *Domain R*
by (*auto intro*!:*image-eqI*)

lemma *Domain-dprod* [*simp*]: *Domain* (*dprod r s*) = *uprod* (*Domain r*) (*Domain s*)
by *auto*

lemma *Domain-dsum* [*simp*]: *Domain* (*dsum r s*) = *usum* (*Domain r*) (*Domain s*)
by *auto*

18.13 Range

lemma *Range-iff*: (*a* : *Range r*) = (*EX y*. (*y*, *a*) : *r*)
by (*simp add*: *Domain-def Range-def*)

lemma *RangeI* [*intro*]: (*a*, *b*) : *r* ==> *b* : *Range r*
by (*unfold Range-def*) (*iprover intro*!: *converseI DomainI*)

lemma *RangeE* [*elim*!]: *b* : *Range r* ==> (!!*x*. (*x*, *b*) : *r* ==> *P*) ==> *P*
by (*unfold Range-def*) (*iprover elim*!: *DomainE dest*!: *converseD*)

lemma *Range-empty* [*simp*]: *Range* {} = {}
by *blast*

lemma *Range-insert*: *Range* (*insert* (*a*, *b*) *r*) = *insert b* (*Range r*)
by *blast*

lemma *Range-Id* [*simp*]: *Range Id* = *UNIV*
by *blast*

lemma *Range-Id-on* [*simp*]: *Range* (*Id-on A*) = *A*
by *auto*

lemma *Range-Un-eq*: *Range*(*A* ∪ *B*) = *Range*(*A*) ∪ *Range*(*B*)
by *blast*

lemma *Range-Int-subset*: *Range*(*A* ∩ *B*) ⊆ *Range*(*A*) ∩ *Range*(*B*)
by *blast*

lemma *Range-Diff-subset*: *Range*(*A*) − *Range*(*B*) ⊆ *Range*(*A* − *B*)
by *blast*

lemma *Range-Union*: *Range* (*Union S*) = (⋃ *A*∈*S*. *Range A*)
by *blast*

lemma *Range-converse*[*simp*]: *Range*(*r*^−*1*) = *Domain r*
by *blast*

lemma *snd-eq-Range*: *snd* ' *R* = *Range R*
by (*auto intro*!:*image-eqI*)

18.14 Field

lemma *mono-Field*: *r* ⊆ *s* ⟹ *Field r* ⊆ *Field s*
by(*auto simp*:*Field-def Domain-def Range-def*)

lemma *Field-empty*[*simp*]: *Field* {} = {}
by(*auto simp*:*Field-def*)

lemma *Field-insert*[*simp*]: *Field* (*insert* (*a*,*b*) *r*) = {*a*,*b*} ∪ *Field r*
by(*auto simp*:*Field-def*)

lemma *Field-Un*[*simp*]: *Field* (*r* ∪ *s*) = *Field r* ∪ *Field s*
by(*auto simp*:*Field-def*)

lemma *Field-Union*[*simp*]: *Field* (⋃ *R*) = ⋃ (*Field* ' *R*)
by(*auto simp*:*Field-def*)

lemma *Field-converse*[*simp*]: *Field*(*r*^−*1*) = *Field r*
by(*auto simp*:*Field-def*)

18.15 Image of a set under a relation

declare *Image-def* [*noatp*]

lemma *Image-iff*: (*b* : *r*``*A*) = (*EX x*:*A*. (*x*, *b*) : *r*)
by (*simp add*: *Image-def*)

lemma *Image-singleton*: *r*``{*a*} = {*b*. (*a*, *b*) : *r*}
by (*simp add*: *Image-def*)

lemma *Image-singleton-iff* [*iff*]: (*b* : *r*``{*a*}) = ((*a*, *b*) : *r*)
by (*rule Image-iff* [*THEN trans*]) *simp*

lemma *ImageI* [*intro*,*noatp*]: (*a*, *b*) : *r* ==> *a* : *A* ==> *b* : *r*``*A*
by (*unfold Image-def*) *blast*

lemma *ImageE* [*elim*!]:
b : *r* `` *A* ==> (!!*x*. (*x*, *b*) : *r* ==> *x* : *A* ==> *P*) ==> *P*
by (*unfold Image-def*) (*iprover elim*!: *CollectE bexE*)

lemma *rev-ImageI*: *a* : *A* ==> (*a*, *b*) : *r* ==> *b* : *r* `` *A*

— This version's more effective when we already have the required a
by *blast*

lemma *Image-empty* [*simp*]: $R``\{\} = \{\}$
by *blast*

lemma *Image-Id* [*simp*]: $Id `` A = A$
by *blast*

lemma *Image-Id-on* [*simp*]: $Id\text{-}on\ A `` B = A \cap B$
by *blast*

lemma *Image-Int-subset*: $R `` (A \cap B) \subseteq R `` A \cap R `` B$
by *blast*

lemma *Image-Int-eq*:
$single\text{-}valued\ (converse\ R) ==> R `` (A \cap B) = R `` A \cap R `` B$
by (*simp add*: *single-valued-def*, *blast*)

lemma *Image-Un*: $R `` (A \cup B) = R `` A \cup R `` B$
by *blast*

lemma *Un-Image*: $(R \cup S) `` A = R `` A \cup S `` A$
by *blast*

lemma *Image-subset*: $r \subseteq A \times B ==> r``C \subseteq B$
by (*iprover intro*!: *subsetI elim*!: *ImageE dest*!: *subsetD SigmaD2*)

lemma *Image-eq-UN*: $r``B = (\bigcup y\in B.\ r``\{y\})$
— NOT suitable for rewriting
by *blast*

lemma *Image-mono*: $r' \subseteq r ==> A' \subseteq A ==> (r' `` A') \subseteq (r `` A)$
by *blast*

lemma *Image-UN*: $(r `` (UNION\ A\ B)) = (\bigcup x\in A.\ r `` (B\ x))$
by *blast*

lemma *Image-INT-subset*: $(r `` INTER\ A\ B) \subseteq (\bigcap x\in A.\ r `` (B\ x))$
by *blast*

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
$[|single\text{-}valued\ (r^{-1}); A\neq\{\}|] ==> r `` INTER\ A\ B = (\bigcap x\in A.\ r `` B\ x)$
apply (*rule equalityI*)
apply (*rule Image-INT-subset*)
apply (*simp add*: *single-valued-def*, *blast*)
done

lemma *Image-subset-eq*: $(r``A \subseteq B) = (A \subseteq - ((r\hat{}-1) \mathbin{``} (-B)))$
by *blast*

18.16 Single valued relations

lemma *single-valuedI*:
$ALL\ x\ y.\ (x,y):r \longrightarrow (ALL\ z.\ (x,z):r \longrightarrow y=z) \Longrightarrow single\text{-}valued\ r$
by (*unfold single-valued-def*)

lemma *single-valuedD*:
$single\text{-}valued\ r \Longrightarrow (x,\ y) : r \Longrightarrow (x,\ z) : r \Longrightarrow y = z$
by (*simp add*: *single-valued-def*)

lemma *single-valued-rel-comp*:
$single\text{-}valued\ r \Longrightarrow single\text{-}valued\ s \Longrightarrow single\text{-}valued\ (r\ O\ s)$
by (*unfold single-valued-def*) *blast*

lemma *single-valued-subset*:
$r \subseteq s \Longrightarrow single\text{-}valued\ s \Longrightarrow single\text{-}valued\ r$
by (*unfold single-valued-def*) *blast*

lemma *single-valued-Id* [*simp*]: *single-valued Id*
by (*unfold single-valued-def*) *blast*

lemma *single-valued-Id-on* [*simp*]: *single-valued* (*Id-on A*)
by (*unfold single-valued-def*) *blast*

18.17 Graphs given by *Collect*

lemma *Domain-Collect-split* [*simp*]: $Domain\{(x,y).\ P\ x\ y\} = \{x.\ EX\ y.\ P\ x\ y\}$
by *auto*

lemma *Range-Collect-split* [*simp*]: $Range\{(x,y).\ P\ x\ y\} = \{y.\ EX\ x.\ P\ x\ y\}$
by *auto*

lemma *Image-Collect-split* [*simp*]: $\{(x,y).\ P\ x\ y\} \mathbin{``} A = \{y.\ EX\ x{:}A.\ P\ x\ y\}$
by *auto*

18.18 Inverse image

lemma *sym-inv-image*: $sym\ r \Longrightarrow sym\ (inv\text{-}image\ r\ f)$
by (*unfold sym-def inv-image-def*) *blast*

lemma *trans-inv-image*: $trans\ r \Longrightarrow trans\ (inv\text{-}image\ r\ f)$
apply (*unfold trans-def inv-image-def*)
apply (*simp* (*no-asm*))
apply *blast*
done

18.19 Finiteness

lemma *finite-converse* [*iff*]: *finite* (*r*^−1) = *finite r*
apply (*subgoal-tac r*^−1 = (%(*x*,*y*). (*y*,*x*)) ‘*r*)
apply *simp*
apply (*rule iffI*)
apply (*erule finite-imageD* [*unfolded inj-on-def*])
apply (*simp split add*: *split-split*)
apply (*erule finite-imageI*)
apply (*simp add*: *converse-def image-def*, *auto*)
apply (*rule bexI*)
prefer *2* **apply** *assumption*
apply *simp*
done

Finiteness of transitive closure (Thanks to Sidi Ehmety)

lemma *finite-Field*: *finite r* ==> *finite* (*Field r*)
— A finite relation has a finite field (= *domain* ∪ *range*.
apply (*induct set*: *finite*)
apply (*auto simp add*: *Field-def Domain-insert Range-insert*)
done

18.20 Version of *lfp-induct* for binary relations

lemmas *lfp-induct2* =
lfp-induct-set [*of* (*a*, *b*), *split-format* (*complete*)]

end

19 Predicate: Predicates as relations and enumerations

theory *Predicate*
imports *Inductive Relation*
begin

notation
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅ - [*900*] *900*) **and**
Sup (⨆ - [*900*] *900*) **and**
top (⊤) **and**
bot (⊥)

19.1 Predicates as (complete) lattices

19.1.1 *op* ⊔ **on** *bool*

lemma *sup-boolI1*:
$P \Longrightarrow P \sqcup Q$
by (*simp add*: *sup-bool-eq*)

lemma *sup-boolI2*:
$Q \Longrightarrow P \sqcup Q$
by (*simp add*: *sup-bool-eq*)

lemma *sup-boolE*:
$P \sqcup Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
by (*auto simp add*: *sup-bool-eq*)

19.1.2 Equality and Subsets

lemma *pred-equals-eq*: $((\lambda x.\ x \in R) = (\lambda x.\ x \in S)) = (R = S)$
by (*simp add*: *mem-def*)

lemma *pred-equals-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) = (\lambda x\ y.\ (x,\ y) \in S)) = (R = S)$
by (*simp add*: *expand-fun-eq mem-def*)

lemma *pred-subset-eq*: $((\lambda x.\ x \in R) <= (\lambda x.\ x \in S)) = (R <= S)$
by (*simp add*: *mem-def*)

lemma *pred-subset-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) <= (\lambda x\ y.\ (x,\ y) \in S)) = (R <= S)$
by *fast*

19.1.3 Top and bottom elements

lemma *top1I* [*intro*!]: *top x*
by (*simp add*: *top-fun-eq top-bool-eq*)

lemma *top2I* [*intro*!]: *top x y*
by (*simp add*: *top-fun-eq top-bool-eq*)

lemma *bot1E* [*elim*!]: *bot x* $\Longrightarrow P$
by (*simp add*: *bot-fun-eq bot-bool-eq*)

lemma *bot2E* [*elim*!]: *bot x y* $\Longrightarrow P$
by (*simp add*: *bot-fun-eq bot-bool-eq*)

19.1.4 The empty set

lemma *bot-empty-eq*: $bot = (\lambda x.\ x \in \{\})$
by (*auto simp add*: *expand-fun-eq*)

lemma *bot-empty-eq2*: $bot = (\lambda x\ y.\ (x,\ y) \in \{\})$
by (*auto simp add*: *expand-fun-eq*)

19.1.5 Binary union

lemma *sup1-iff* [*simp*]: $sup\ A\ B\ x \longleftrightarrow A\ x \mid B\ x$
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup2-iff* [*simp*]: $sup\ A\ B\ x\ y \longleftrightarrow A\ x\ y \mid B\ x\ y$
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup-Un-eq* [*pred-set-conv*]: $sup\ (\lambda x.\ x \in R)\ (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cup S)$
by (*simp add*: *expand-fun-eq*)

lemma *sup-Un-eq2* [*pred-set-conv*]: $sup\ (\lambda x\ y.\ (x,\ y) \in R)\ (\lambda x\ y.\ (x,\ y) \in S) = (\lambda x\ y.\ (x,\ y) \in R \cup S)$
by (*simp add*: *expand-fun-eq*)

lemma *sup1I1* [*elim?*]: $A\ x \Longrightarrow sup\ A\ B\ x$
by *simp*

lemma *sup2I1* [*elim?*]: $A\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
by *simp*

lemma *sup1I2* [*elim?*]: $B\ x \Longrightarrow sup\ A\ B\ x$
by *simp*

lemma *sup2I2* [*elim?*]: $B\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
by *simp*

Classical introduction rule: no commitment to A vs B.

lemma *sup1CI* [*intro!*]: $(\sim B\ x \Longrightarrow A\ x) \Longrightarrow sup\ A\ B\ x$
by *auto*

lemma *sup2CI* [*intro!*]: $(\sim B\ x\ y \Longrightarrow A\ x\ y) \Longrightarrow sup\ A\ B\ x\ y$
by *auto*

lemma *sup1E* [*elim!*]: $sup\ A\ B\ x \Longrightarrow (A\ x \Longrightarrow P) \Longrightarrow (B\ x \Longrightarrow P) \Longrightarrow P$
by *simp iprover*

lemma *sup2E* [*elim!*]: $sup\ A\ B\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow P) \Longrightarrow (B\ x\ y \Longrightarrow P) \Longrightarrow P$
by *simp iprover*

19.1.6 Binary intersection

lemma *inf1-iff* [*simp*]: $inf\ A\ B\ x \longleftrightarrow A\ x \wedge B\ x$

by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf2-iff* [*simp*]: *inf A B x y* ⟷ *A x y* ∧ *B x y*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf-Int-eq* [*pred-set-conv*]: *inf* (λ*x*. *x* ∈ *R*) (λ*x*. *x* ∈ *S*) = (λ*x*. *x* ∈ *R* ∩ *S*)
by (*simp add*: *expand-fun-eq*)

lemma *inf-Int-eq2* [*pred-set-conv*]: *inf* (λ*x y*. (*x*, *y*) ∈ *R*) (λ*x y*. (*x*, *y*) ∈ *S*) = (λ*x y*. (*x*, *y*) ∈ *R* ∩ *S*)
by (*simp add*: *expand-fun-eq*)

lemma *inf1I* [*intro*!]: *A x* ==> *B x* ==> *inf A B x*
by *simp*

lemma *inf2I* [*intro*!]: *A x y* ==> *B x y* ==> *inf A B x y*
by *simp*

lemma *inf1D1*: *inf A B x* ==> *A x*
by *simp*

lemma *inf2D1*: *inf A B x y* ==> *A x y*
by *simp*

lemma *inf1D2*: *inf A B x* ==> *B x*
by *simp*

lemma *inf2D2*: *inf A B x y* ==> *B x y*
by *simp*

lemma *inf1E* [*elim*!]: *inf A B x* ==> (*A x* ==> *B x* ==> *P*) ==> *P*
by *simp*

lemma *inf2E* [*elim*!]: *inf A B x y* ==> (*A x y* ==> *B x y* ==> *P*) ==> *P*
by *simp*

19.1.7 Unions of families

lemma *SUP1-iff* [*simp*]: (*SUP x*:*A*. *B x*) *b* = (*EX x*:*A*. *B x b*)
by (*simp add*: *SUPR-def Sup-fun-def Sup-bool-def*) *blast*

lemma *SUP2-iff* [*simp*]: (*SUP x*:*A*. *B x*) *b c* = (*EX x*:*A*. *B x b c*)
by (*simp add*: *SUPR-def Sup-fun-def Sup-bool-def*) *blast*

lemma *SUP1-I* [*intro*]: *a* : *A* ==> *B a b* ==> (*SUP x*:*A*. *B x*) *b*
by *auto*

lemma *SUP2-I* [*intro*]: *a* : *A* ==> *B a b c* ==> (*SUP x*:*A*. *B x*) *b c*
by *auto*

lemma *SUP1-E* [*elim*!]: (*SUP* x:*A*. *B* x) b ==> (!!x. x : *A* ==> *B* x b ==> *R*) ==> *R*
by *auto*

lemma *SUP2-E* [*elim*!]: (*SUP* x:*A*. *B* x) b c ==> (!!x. x : *A* ==> *B* x b c ==> *R*) ==> *R*
by *auto*

lemma *SUP-UN-eq*: (*SUP* i. (λx. x ∈ r i)) = (λx. x ∈ (*UN* i. r i))
by (*simp add*: *expand-fun-eq*)

lemma *SUP-UN-eq2*: (*SUP* i. (λx y. (x, y) ∈ r i)) = (λx y. (x, y) ∈ (*UN* i. r i))
by (*simp add*: *expand-fun-eq*)

19.1.8 Intersections of families

lemma *INF1-iff* [*simp*]: (*INF* x:*A*. *B* x) b = (*ALL* x:*A*. *B* x b)
by (*simp add*: *INFI-def Inf-fun-def Inf-bool-def*) *blast*

lemma *INF2-iff* [*simp*]: (*INF* x:*A*. *B* x) b c = (*ALL* x:*A*. *B* x b c)
by (*simp add*: *INFI-def Inf-fun-def Inf-bool-def*) *blast*

lemma *INF1-I* [*intro*!]: (!!x. x : *A* ==> *B* x b) ==> (*INF* x:*A*. *B* x) b
by *auto*

lemma *INF2-I* [*intro*!]: (!!x. x : *A* ==> *B* x b c) ==> (*INF* x:*A*. *B* x) b c
by *auto*

lemma *INF1-D* [*elim*]: (*INF* x:*A*. *B* x) b ==> a : *A* ==> *B* a b
by *auto*

lemma *INF2-D* [*elim*]: (*INF* x:*A*. *B* x) b c ==> a : *A* ==> *B* a b c
by *auto*

lemma *INF1-E* [*elim*]: (*INF* x:*A*. *B* x) b ==> (*B* a b ==> *R*) ==> (a ~: *A* ==> *R*) ==> *R*
by *auto*

lemma *INF2-E* [*elim*]: (*INF* x:*A*. *B* x) b c ==> (*B* a b c ==> *R*) ==> (a ~: *A* ==> *R*) ==> *R*
by *auto*

lemma *INF-INT-eq*: (*INF* i. (λx. x ∈ r i)) = (λx. x ∈ (*INT* i. r i))
by (*simp add*: *expand-fun-eq*)

lemma *INF-INT-eq2*: (*INF* i. (λx y. (x, y) ∈ r i)) = (λx y. (x, y) ∈ (*INT* i. r i))
by (*simp add*: *expand-fun-eq*)

19.2 Predicates as relations

19.2.1 Composition

inductive
pred-comp :: [*'b => 'c => bool*, *'a => 'b => bool*] *=> 'a => 'c => bool*
(**infixr** *OO 75*)
for *r* :: *'b => 'c => bool* **and** *s* :: *'a => 'b => bool*
where
pred-compI [*intro*]: *s a b ==> r b c ==> (r OO s) a c*

inductive-cases *pred-compE* [*elim!*]: *(r OO s) a c*

lemma *pred-comp-rel-comp-eq* [*pred-set-conv*]:
((λx y. (x, y) ∈ r) OO (λx y. (x, y) ∈ s)) = (λx y. (x, y) ∈ r O s)
by (*auto simp add*: *expand-fun-eq elim*: *pred-compE*)

19.2.2 Converse

inductive
conversep :: *('a => 'b => bool) => 'b => 'a => bool*
(*(-^--1) [1000] 1000*)
for *r* :: *'a => 'b => bool*
where
conversepI: *r a b ==> r^--1 b a*

notation (*xsymbols*)
conversep (*(-$^{-1-1}$) [1000] 1000*)

lemma *conversepD*:
assumes *ab*: *r^--1 a b*
shows *r b a* **using** *ab*
by *cases simp*

lemma *conversep-iff* [*iff*]: *r^--1 a b = r b a*
by (*iprover intro*: *conversepI dest*: *conversepD*)

lemma *conversep-converse-eq* [*pred-set-conv*]:
(λx y. (x, y) ∈ r)^--1 = (λx y. (x, y) ∈ r^-1)
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-conversep* [*simp*]: *(r^--1)^--1 = r*
by (*iprover intro*: *order-antisym conversepI dest*: *conversepD*)

lemma *converse-pred-comp*: *(r OO s)^--1 = s^--1 OO r^--1*
by (*iprover intro*: *order-antisym conversepI pred-compI*
elim: *pred-compE dest*: *conversepD*)

lemma *converse-meet*: *(inf r s)^--1 = inf r^--1 s^--1*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *converse-join*: (*sup r s*) ^−−1 = *sup r* ^−−1 *s* ^−−1
by (*simp add*: *sup-fun-eq sup-bool-eq*)
(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *conversep-noteq* [*simp*]: (*op* ~=) ^−−1 = *op* ~=
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-eq* [*simp*]: (*op* =) ^−−1 = *op* =
by (*auto simp add*: *expand-fun-eq*)

19.2.3 Domain

inductive
DomainP :: ('*a* => '*b* => *bool*) => '*a* => *bool*
for *r* :: '*a* => '*b* => *bool*
where
DomainPI [*intro*]: *r a b* ==> *DomainP r a*

inductive-cases *DomainPE* [*elim*!]: *DomainP r a*

lemma *DomainP-Domain-eq* [*pred-set-conv*]: *DomainP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Domain r*)
by (*blast intro*!: *Orderings.order-antisym predicate1I*)

19.2.4 Range

inductive
RangeP :: ('*a* => '*b* => *bool*) => '*b* => *bool*
for *r* :: '*a* => '*b* => *bool*
where
RangePI [*intro*]: *r a b* ==> *RangeP r b*

inductive-cases *RangePE* [*elim*!]: *RangeP r b*

lemma *RangeP-Range-eq* [*pred-set-conv*]: *RangeP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Range r*)
by (*blast intro*!: *Orderings.order-antisym predicate1I*)

19.2.5 Inverse image

definition
inv-imagep :: ('*b* => '*b* => *bool*) => ('*a* => '*b*) => '*a* => '*a* => *bool* **where**
inv-imagep r f == %*x y*. *r* (*f x*) (*f y*)

lemma [*pred-set-conv*]: *inv-imagep* (λ*x y*. (*x*, *y*) ∈ *r*) *f* = (λ*x y*. (*x*, *y*) ∈ *inv-image r f*)
by (*simp add*: *inv-image-def inv-imagep-def*)

lemma *in-inv-imagep* [*simp*]: *inv-imagep r f x y = r (f x) (f y)*
by (*simp add*: *inv-imagep-def*)

19.2.6 Powerset

definition *Powp* :: (*'a ⇒ bool*) *⇒ 'a set ⇒ bool* **where**
Powp A == λB. ∀ x ∈ B. A x

lemma *Powp-Pow-eq* [*pred-set-conv*]: *Powp (λx. x ∈ A) = (λx. x ∈ Pow A)*
by (*auto simp add*: *Powp-def expand-fun-eq*)

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred pred-subset-eq*]

19.2.7 Properties of relations

abbreviation *antisymP* :: (*'a => 'a => bool*) *=> bool* **where**
antisymP r == antisym {(x, y). r x y}

abbreviation *transP* :: (*'a => 'a => bool*) *=> bool* **where**
transP r == trans {(x, y). r x y}

abbreviation *single-valuedP* :: (*'a => 'b => bool*) *=> bool* **where**
single-valuedP r == single-valued {(x, y). r x y}

19.3 Predicates as enumerations

19.3.1 The type of predicate enumerations (a monad)

datatype *'a pred = Pred 'a ⇒ bool*

primrec *eval* :: *'a pred ⇒ 'a ⇒ bool* **where**
eval-pred: *eval (Pred f) = f*

lemma *Pred-eval* [*simp*]:
Pred (eval x) = x
by (*cases x*) *simp*

lemma *eval-inject*: *eval x = eval y ⟷ x = y*
by (*cases x*) *auto*

definition *single* :: *'a ⇒ 'a pred* **where**
single x = Pred ((op =) x)

definition *bind* :: *'a pred ⇒ ('a ⇒ 'b pred) ⇒ 'b pred* (**infixl** *⨠= 70*) **where**
P ⨠= f = Pred (λx. (∃ y. eval P y ∧ eval (f y) x))

instantiation *pred* :: (*type*) *complete-lattice*
begin

definition

$P \leq Q \longleftrightarrow \mathit{eval}\ P \leq \mathit{eval}\ Q$

definition
$P < Q \longleftrightarrow \mathit{eval}\ P < \mathit{eval}\ Q$

definition
$\bot = \mathit{Pred}\ \bot$

definition
$\top = \mathit{Pred}\ \top$

definition
$P \sqcap Q = \mathit{Pred}\ (\mathit{eval}\ P \sqcap \mathit{eval}\ Q)$

definition
$P \sqcup Q = \mathit{Pred}\ (\mathit{eval}\ P \sqcup \mathit{eval}\ Q)$

definition
$\bigsqcap A = \mathit{Pred}\ (\mathit{INFI}\ A\ \mathit{eval})$

definition
$\bigsqcup A = \mathit{Pred}\ (\mathit{SUPR}\ A\ \mathit{eval})$

instance by *default*
(*auto simp add*: *less-eq-pred-def less-pred-def*
inf-pred-def sup-pred-def bot-pred-def top-pred-def
Inf-pred-def Sup-pred-def,
auto simp add: *le-fun-def less-fun-def le-bool-def less-bool-def*
eval-inject mem-def)

end

lemma *bind-bind*:
$(P \gg\!= Q) \gg\!= R = P \gg\!= (\lambda x.\ Q\ x \gg\!= R)$
by (*auto simp add*: *bind-def expand-fun-eq*)

lemma *bind-single*:
$P \gg\!= \mathit{single} = P$
by (*simp add*: *bind-def single-def*)

lemma *single-bind*:
$\mathit{single}\ x \gg\!= P = P\ x$
by (*simp add*: *bind-def single-def*)

lemma *bottom-bind*:
$\bot \gg\!= P = \bot$
by (*auto simp add*: *bot-pred-def bind-def expand-fun-eq*)

lemma *sup-bind*:

$(P \sqcup Q) \gg\!= R = P \gg\!= R \sqcup Q \gg\!= R$
by (*auto simp add*: *bind-def sup-pred-def expand-fun-eq*)

lemma *Sup-bind*: $(\bigsqcup A \gg\!= f) = \bigsqcup ((\lambda x.\ x \gg\!= f)\ `\ A)$
by (*auto simp add*: *bind-def Sup-pred-def expand-fun-eq*)

lemma *pred-iffI*:
assumes $\bigwedge x.$ *eval* $A\ x \Longrightarrow$ *eval* $B\ x$
and $\bigwedge x.$ *eval* $B\ x \Longrightarrow$ *eval* $A\ x$
shows $A = B$
proof −
from *assms* **have** $\bigwedge x.$ *eval* $A\ x \longleftrightarrow$ *eval* $B\ x$ **by** *blast*
then show *?thesis* **by** (*cases A*, *cases B*) (*simp add*: *expand-fun-eq*)
qed

lemma *singleI*: *eval* (*single x*) *x*
unfolding *single-def* **by** *simp*

lemma *singleI-unit*: *eval* (*single* ()) *x*
by *simp* (*rule singleI*)

lemma *singleE*: *eval* (*single x*) $y \Longrightarrow (y = x \Longrightarrow P) \Longrightarrow P$
unfolding *single-def* **by** *simp*

lemma *singleE'*: *eval* (*single x*) $y \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$
by (*erule singleE*) *simp*

lemma *bindI*: *eval P x* $\Longrightarrow$ *eval* $(Q\ x)\ y \Longrightarrow$ *eval* $(P \gg\!= Q)\ y$
unfolding *bind-def* **by** *auto*

lemma *bindE*: *eval* $(R \gg\!= Q)\ y \Longrightarrow (\bigwedge x.$ *eval R x* $\Longrightarrow$ *eval* $(Q\ x)\ y \Longrightarrow P) \Longrightarrow$
P
unfolding *bind-def* **by** *auto*

lemma *botE*: *eval* $\bot\ x \Longrightarrow P$
unfolding *bot-pred-def* **by** *auto*

lemma *supI1*: *eval A x* $\Longrightarrow$ *eval* $(A \sqcup B)\ x$
unfolding *sup-pred-def* **by** *simp*

lemma *supI2*: *eval B x* $\Longrightarrow$ *eval* $(A \sqcup B)\ x$
unfolding *sup-pred-def* **by** *simp*

lemma *supE*: *eval* $(A \sqcup B)\ x \Longrightarrow$ (*eval A x* $\Longrightarrow P) \Longrightarrow$ (*eval B x* $\Longrightarrow P) \Longrightarrow P$
unfolding *sup-pred-def* **by** *auto*

19.3.2 Derived operations

definition *if-pred* :: *bool* ⇒ *unit pred* **where**

if-pred-eq: *if-pred b* = (*if b then single* () *else* ⊥)

definition *not-pred* :: *unit pred* ⇒ *unit pred* **where**
not-pred-eq: *not-pred P* = (*if eval P* () *then* ⊥ *else single* ())

lemma *if-predI*: *P* ⟹ *eval* (*if-pred P*) ()
unfolding *if-pred-eq* **by** (*auto intro*: *singleI*)

lemma *if-predE*: *eval* (*if-pred b*) *x* ⟹ (*b* ⟹ *x* = () ⟹ *P*) ⟹ *P*
unfolding *if-pred-eq* **by** (*cases b*) (*auto elim*: *botE*)

lemma *not-predI*: ¬ *P* ⟹ *eval* (*not-pred* (*Pred* (λ*u*. *P*))) ()
unfolding *not-pred-eq eval-pred* **by** (*auto intro*: *singleI*)

lemma *not-predI'*: ¬ *eval P* () ⟹ *eval* (*not-pred P*) ()
unfolding *not-pred-eq* **by** (*auto intro*: *singleI*)

lemma *not-predE*: *eval* (*not-pred* (*Pred* (λ*u*. *P*))) *x* ⟹ (¬ *P* ⟹ *thesis*) ⟹ *thesis*
unfolding *not-pred-eq*
by (*auto split*: *split-if-asm elim*: *botE*)

lemma *not-predE'*: *eval* (*not-pred P*) *x* ⟹ (¬ *eval P x* ⟹ *thesis*) ⟹ *thesis*
unfolding *not-pred-eq*
by (*auto split*: *split-if-asm elim*: *botE*)

19.3.3 Implementation

datatype *'a seq* = *Empty* | *Insert 'a 'a pred* | *Join 'a pred 'a seq*

primrec *pred-of-seq* :: *'a seq* ⇒ *'a pred* **where**
pred-of-seq Empty = ⊥
| *pred-of-seq* (*Insert x P*) = *single x* ⊔ *P*
| *pred-of-seq* (*Join P xq*) = *P* ⊔ *pred-of-seq xq*

definition *Seq* :: (*unit* ⇒ *'a seq*) ⇒ *'a pred* **where**
Seq f = *pred-of-seq* (*f* ())

code-datatype *Seq*

primrec *member* :: *'a seq* ⇒ *'a* ⇒ *bool* **where**
member Empty x ⟷ *False*
| *member* (*Insert y P*) *x* ⟷ *x* = *y* ∨ *eval P x*
| *member* (*Join P xq*) *x* ⟷ *eval P x* ∨ *member xq x*

lemma *eval-member*:
member xq = *eval* (*pred-of-seq xq*)
proof (*induct xq*)
case *Empty* **show** *?case*

```
  by (auto simp add: expand-fun-eq elim: botE)
next
  case Insert show ?case
  by (auto simp add: expand-fun-eq elim: supE singleE intro: supI1 supI2 singleI)
next
  case Join then show ?case
  by (auto simp add: expand-fun-eq elim: supE intro: supI1 supI2)
qed

lemma eval-code [code]: eval (Seq f) = member (f ())
  unfolding Seq-def by (rule sym, rule eval-member)

lemma single-code [code]:
  single x = Seq (λu. Insert x ⊥)
  unfolding Seq-def by simp

primrec apply :: ('a ⇒ 'b Predicate.pred) ⇒ 'a seq ⇒ 'b seq where
    apply f Empty = Empty
  | apply f (Insert x P) = Join (f x) (Join (P ⤜ f) Empty)
  | apply f (Join P xq) = Join (P ⤜ f) (apply f xq)

lemma apply-bind:
  pred-of-seq (apply f xq) = pred-of-seq xq ⤜ f
proof (induct xq)
  case Empty show ?case
    by (simp add: bottom-bind)
next
  case Insert show ?case
    by (simp add: single-bind sup-bind)
next
  case Join then show ?case
    by (simp add: sup-bind)
qed

lemma bind-code [code]:
  Seq g ⤜ f = Seq (λu. apply f (g ()))
  unfolding Seq-def by (rule sym, rule apply-bind)

lemma bot-set-code [code]:
  ⊥ = Seq (λu. Empty)
  unfolding Seq-def by simp

primrec adjunct :: 'a pred ⇒ 'a seq ⇒ 'a seq where
    adjunct P Empty = Join P Empty
  | adjunct P (Insert x Q) = Insert x (Q ⊔ P)
  | adjunct P (Join Q xq) = Join Q (adjunct P xq)

lemma adjunct-sup:
  pred-of-seq (adjunct P xq) = P ⊔ pred-of-seq xq
```

by (*induct xq*) (*simp-all add*: *sup-assoc sup-commute sup-left-commute*)

lemma *sup-code* [*code*]:
Seq f ⊔ *Seq g* = *Seq* (λ*u*. *case f* ()
of Empty ⇒ *g* ()
| *Insert x P* ⇒ *Insert x* (*P* ⊔ *Seq g*)
| *Join P xq* ⇒ *adjunct* (*Seq g*) (*Join P xq*))
proof (*cases f* ())
case *Empty*
thus *?thesis*
unfolding *Seq-def* **by** (*simp add*: *sup-commute* [*of* ⊥] *sup-bot*)
next
case *Insert*
thus *?thesis*
unfolding *Seq-def* **by** (*simp add*: *sup-assoc*)
next
case *Join*
thus *?thesis*
unfolding *Seq-def*
by (*simp add*: *adjunct-sup sup-assoc sup-commute sup-left-commute*)
qed

primrec *contained* :: *'a seq* ⇒ *'a pred* ⇒ *bool* **where**
contained Empty Q ⟷ *True*
| *contained* (*Insert x P*) *Q* ⟷ *eval Q x* ∧ *P* ≤ *Q*
| *contained* (*Join P xq*) *Q* ⟷ *P* ≤ *Q* ∧ *contained xq Q*

lemma *single-less-eq-eval*:
single x ≤ *P* ⟷ *eval P x*
by (*auto simp add*: *single-def less-eq-pred-def mem-def*)

lemma *contained-less-eq*:
contained xq Q ⟷ *pred-of-seq xq* ≤ *Q*
by (*induct xq*) (*simp-all add*: *single-less-eq-eval*)

lemma *less-eq-pred-code* [*code*]:
Seq f ≤ *Q* = (*case f* ()
of Empty ⇒ *True*
| *Insert x P* ⇒ *eval Q x* ∧ *P* ≤ *Q*
| *Join P xq* ⇒ *P* ≤ *Q* ∧ *contained xq Q*)
by (*cases f* ())
(*simp-all add*: *Seq-def single-less-eq-eval contained-less-eq*)

lemma *eq-pred-code* [*code*]:
fixes *P Q* :: *'a*::*eq pred*
shows *eq-class.eq P Q* ⟷ *P* ≤ *Q* ∧ *Q* ≤ *P*
unfolding *eq* **by** *auto*

lemma [*code*]:

pred-case f P = f (eval P)
by (*cases P*) *simp*

lemma [*code*]:
pred-rec f P = f (eval P)
by (*cases P*) *simp*

no-notation
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*) **and**
top (⊤) **and**
bot (⊥) **and**
bind (**infixl** ⤜ *70*)

hide (**open**) *type pred seq*
hide (**open**) *const Pred eval single bind if-pred not-pred*
Empty Insert Join Seq member pred-of-seq apply adjunct

end

20 Transitive-Closure: Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
imports *Predicate*
uses ~~/*src/Provers/trancl.ML*
begin

rtrancl is reflexive/transitive closure, *trancl* is transitive closure, *reflcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

inductive-set
rtrancl :: ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ((-^*) [*1000*] *999*)
for *r* :: ('*a* × '*a*) *set*
where
rtrancl-refl [*intro*!, *Pure.intro*!, *simp*]: (*a*, *a*) : *r*^*
| *rtrancl-into-rtrancl* [*Pure.intro*]: (*a*, *b*) : *r*^* ==> (*b*, *c*) : *r* ==> (*a*, *c*) : *r*^*

inductive-set
trancl :: ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ((-^+) [*1000*] *999*)
for *r* :: ('*a* × '*a*) *set*
where
r-into-trancl [*intro*, *Pure.intro*]: (*a*, *b*) : *r* ==> (*a*, *b*) : *r*^+
| *trancl-into-trancl* [*Pure.intro*]: (*a*, *b*) : *r*^+ ==> (*b*, *c*) : *r* ==> (*a*, *c*) : *r*^+

notation
rtranclp ((-^**) [*1000*] *1000*) **and**
tranclp ((-^++) [*1000*] *1000*)

abbreviation
reflclp :: ('*a* => '*a* => *bool*) => '*a* => '*a* => *bool* ((-^==) [*1000*] *1000*)
where
r^== == *sup r op* =

abbreviation
reflcl :: ('*a* × '*a*) *set* => ('*a* × '*a*) *set* ((-^=) [*1000*] *999*) **where**
r^= == *r* ∪ *Id*

notation (*xsymbols*)
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-⁺⁺) [*1000*] *1000*) **and**
reflclp ((-==) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-⁺) [*1000*] *999*) **and**
reflcl ((-=) [*1000*] *999*)

notation (*HTML* **output**)
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-⁺⁺) [*1000*] *1000*) **and**
reflclp ((-==) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-⁺) [*1000*] *999*) **and**
reflcl ((-=) [*1000*] *999*)

20.1 Reflexive closure

lemma *refl-reflcl*[*simp*]: *refl*(*r*^=)
by(*simp add*:*refl-on-def*)

lemma *antisym-reflcl*[*simp*]: *antisym*(*r*^=) = *antisym r*
by(*simp add*:*antisym-def*)

lemma *trans-reflclI*[*simp*]: *trans r* ⟹ *trans*(*r*^=)
unfolding *trans-def* **by** *blast*

20.2 Reflexive-transitive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: (*sup* (λ*x y*. (*x*, *y*) ∈ *r*) *op* =) = (λ*x y*. (*x*, *y*) ∈ *r Un Id*)
by (*simp add*: *expand-fun-eq*)

lemma *r-into-rtrancl* [*intro*]: !!*p*. *p* ∈ *r* ==> *p* ∈ *r*^*
— *rtrancl* of *r* contains *r*
apply (*simp only*: *split-tupled-all*)

apply (*erule rtrancl-refl* [*THEN rtrancl-into-rtrancl*])
done

lemma *r-into-rtranclp* [*intro*]: *r x y ==> r^** x y*
— *rtrancl* of *r* contains *r*
by (*erule rtranclp.rtrancl-refl* [*THEN rtranclp.rtrancl-into-rtrancl*])

lemma *rtranclp-mono*: *r ≤ s ==> r^** ≤ s^***
— monotonicity of *rtrancl*
apply (*rule predicate2I*)
apply (*erule rtranclp.induct*)
apply (*rule-tac* [*2*] *rtranclp.rtrancl-into-rtrancl*, *blast+*)
done

lemmas *rtrancl-mono = rtranclp-mono* [*to-set*]

theorem *rtranclp-induct* [*consumes 1*, *case-names base step*, *induct set*: *rtranclp*]:
assumes *a*: *r^** a b*
and *cases*: *P a !!y z. [| r^** a y; r y z; P y |] ==> P z*
shows *P b*
proof −
from *a* **have** *a = a −−> P b*
by (*induct %x y. x = a −−> P y a b*) (*iprover intro*: *cases*)+
then show *?thesis* **by** *iprover*
qed

lemmas *rtrancl-induct* [*induct set*: *rtrancl*] = *rtranclp-induct* [*to-set*]

lemmas *rtranclp-induct2* =
rtranclp-induct[*of - (ax,ay) (bx,by)*, *split-rule*,
consumes 1, *case-names refl step*]

lemmas *rtrancl-induct2* =
rtrancl-induct[*of (ax,ay) (bx,by)*, *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *refl-rtrancl*: *refl* (*r^**)
by (*unfold refl-on-def*) *fast*

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* (*r^**)
proof (*rule transI*)
fix *x y z*
assume (*x*, *y*) ∈ *r**
assume (*y*, *z*) ∈ *r**
then show (*x*, *z*) ∈ *r**
proof *induct*
case *base*
show (*x*, *y*) ∈ *r** **by** *fact*

```
  next
    case (step u v)
    from ‹(x, u) ∈ r*› and ‹(u, v) ∈ r›
    show (x, v) ∈ r* ..
  qed
qed

lemmas rtrancl-trans = trans-rtrancl [THEN transD, standard]

lemma rtranclp-trans:
  assumes xy: r ^** x y
  and yz: r ^** y z
  shows r ^** x z using yz xy
  by induct iprover+

lemma rtranclE [cases set: rtrancl]:
  assumes major: (a::'a, b) : r ^*
  obtains
    (base) a = b
  | (step) y where (a, y) : r ^* and (y, b) : r
  — elimination of rtrancl – by induction on a special formula
  apply (subgoal-tac (a::'a) = b | (EX y. (a,y) : r ^* & (y,b) : r))
   apply (rule-tac [2] major [THEN rtrancl-induct])
    prefer 2 apply blast
   prefer 2 apply blast
  apply (erule asm-rl exE disjE conjE base step)+
  done

lemma rtrancl-Int-subset: [| Id ⊆ s; r O (r ^* ∩ s) ⊆ s|] ==> r ^* ⊆ s
  apply (rule subsetI)
  apply (rule-tac p=x in PairE, clarify)
  apply (erule rtrancl-induct, auto)
  done

lemma converse-rtranclp-into-rtranclp:
  r a b ⟹ r** b c ⟹ r** a c
  by (rule rtranclp-trans) iprover+

lemmas converse-rtrancl-into-rtrancl = converse-rtranclp-into-rtranclp [to-set]
```

More r^* equations and inclusions.

```
lemma rtranclp-idemp [simp]: (r ^**) ^** = r ^**
  apply (auto intro!: order-antisym)
  apply (erule rtranclp-induct)
   apply (rule rtranclp.rtrancl-refl)
  apply (blast intro: rtranclp-trans)
  done

lemmas rtrancl-idemp [simp] = rtranclp-idemp [to-set]
```

lemma *rtrancl-idemp-self-comp* [*simp*]: $R\hat{}* \; O \; R\hat{}* = R\hat{}*$
apply (*rule set-ext*)
apply (*simp only*: *split-tupled-all*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *rtrancl-subset-rtrancl*: $r \subseteq s\hat{}* \Longrightarrow r\hat{}* \subseteq s\hat{}*$
apply (*drule rtrancl-mono*)
apply *simp*
done

lemma *rtranclp-subset*: $R \leq S \Longrightarrow S \leq R\hat{}** \Longrightarrow S\hat{}** = R\hat{}**$
apply (*drule rtranclp-mono*)
apply (*drule rtranclp-mono*)
apply *simp*
done

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: $(sup \; (R\hat{}**) \; (S\hat{}**))\hat{}** = (sup \; R \; S)\hat{}**$
by (*blast intro*!: *rtranclp-subset intro*: *rtranclp-mono* [*THEN predicate2D*])

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

lemma *rtranclp-reflcl* [*simp*]: $(R\hat{}==)\hat{}** = R\hat{}**$
by (*blast intro*!: *rtranclp-subset*)

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflcl* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - Id)\hat{}* = r\hat{}*$
apply (*rule sym*)
apply (*rule rtrancl-subset*, *blast*, *clarify*)
apply (*rename-tac a b*)
apply (*case-tac a = b*)
apply *blast*
apply (*blast intro*!: *r-into-rtrancl*)
done

lemma *rtranclp-r-diff-Id*: $(inf \; r \; op \; \tilde{}=)\hat{}** = r\hat{}**$
apply (*rule sym*)
apply (*rule rtranclp-subset*)
apply *blast+*
done

theorem *rtranclp-converseD*:
assumes *r*: $(r\hat{}--1)\hat{}** \; x \; y$
shows $r\hat{}** \; y \; x$
proof −

```
  from r show ?thesis
    by induct (iprover intro: rtranclp-trans dest!: conversepD)+
qed

lemmas rtrancl-converseD = rtranclp-converseD [to-set]

theorem rtranclp-converseI:
  assumes r^** y x
  shows (r^--1)^** x y
  using assms
  by induct (iprover intro: rtranclp-trans conversepI)+

lemmas rtrancl-converseI = rtranclp-converseI [to-set]

lemma rtrancl-converse: (r^-1)^* = (r^*)^-1
  by (fast dest!: rtrancl-converseD intro!: rtrancl-converseI)

lemma sym-rtrancl: sym r ==> sym (r^*)
  by (simp only: sym-conv-converse-eq rtrancl-converse [symmetric])

theorem converse-rtranclp-induct[consumes 1]:
  assumes major: r^** a b
    and cases: P b !!y z. [| r y z; r^** z b; P z |] ==> P y
  shows P a
  using rtranclp-converseI [OF major]
  by induct (iprover intro: cases dest!: conversepD rtranclp-converseD)+

lemmas converse-rtrancl-induct = converse-rtranclp-induct [to-set]

lemmas converse-rtranclp-induct2 =
  converse-rtranclp-induct [of - (ax,ay) (bx,by), split-rule,
                consumes 1, case-names refl step]

lemmas converse-rtrancl-induct2 =
  converse-rtrancl-induct [of (ax,ay) (bx,by), split-format (complete),
                consumes 1, case-names refl step]

lemma converse-rtranclpE:
  assumes major: r^** x z
    and cases: x=z ==> P
      !!y. [| r x y; r^** y z |] ==> P
  shows P
  apply (subgoal-tac x = z | (EX y. r x y & r^** y z))
   apply (rule-tac [2] major [THEN converse-rtranclp-induct])
    prefer 2 apply iprover
   prefer 2 apply iprover
  apply (erule asm-rl exE disjE conjE cases)+
  done
```

lemmas *converse-rtranclE = converse-rtranclpE* [*to-set*]

lemmas *converse-rtranclpE2 = converse-rtranclpE* [*of* - (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemmas *converse-rtranclE2 = converse-rtranclE* [*of* (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemma *r-comp-rtrancl-eq*: *r O r^* = r^* O r*
by (*blast elim*: *rtranclE converse-rtranclE*
intro: *rtrancl-into-rtrancl converse-rtrancl-into-rtrancl*)

lemma *rtrancl-unfold*: *r^* = Id Un r O r^**
by (*auto intro*: *rtrancl-into-rtrancl elim*: *rtranclE*)

20.3 Transitive closure

lemma *trancl-mono*: !!*p*. *p* ∈ *r^+* ==> *r* ⊆ *s* ==> *p* ∈ *s^+*
apply (*simp add*: *split-tupled-all*)
apply (*erule trancl.induct*)
apply (*iprover dest*: *subsetD*)+
done

lemma *r-into-trancl'*: !!*p*. *p* : *r* ==> *p* : *r^+*
by (*simp only*: *split-tupled-all*) (*erule r-into-trancl*)

Conversions between *trancl* and *rtrancl*.

lemma *tranclp-into-rtranclp*: *r^++ a b* ==> *r^** a b*
by (*erule tranclp.induct*) *iprover*+

lemmas *trancl-into-rtrancl = tranclp-into-rtranclp* [*to-set*]

lemma *rtranclp-into-tranclp1*: **assumes** *r*: *r^** a b*
shows !!*c*. *r b c* ==> *r^++ a c* **using** *r*
by *induct iprover*+

lemmas *rtrancl-into-trancl1 = rtranclp-into-tranclp1* [*to-set*]

lemma *rtranclp-into-tranclp2*: [| *r a b*; *r^** b c* |] ==> *r^++ a c*
— intro rule from *r* and *rtrancl*
apply (*erule rtranclp.cases*)
apply *iprover*
apply (*rule rtranclp-trans* [*THEN rtranclp-into-tranclp1*])
apply (*simp* | *rule r-into-rtranclp*)+
done

lemmas *rtrancl-into-trancl2 = rtranclp-into-tranclp2* [*to-set*]

Nice induction rule for *trancl*

lemma *tranclp-induct* [*consumes 1*, *case-names base step*, *induct pred*: *tranclp*]:
assumes *r^++ a b*

and *cases*: !!*y*. *r a y* ==> *P y*
!!*y z*. *r*^++ *a y* ==> *r y z* ==> *P y* ==> *P z*
shows *P b*
proof −
from ‹*r*^++ *a b*› **have** *a* = *a* −−> *P b*
by (*induct* %*x y*. *x* = *a* −−> *P y a b*) (*iprover intro*: *cases*)+
then show *?thesis* **by** *iprover*
qed

lemmas *trancl-induct* [*induct set*: *trancl*] = *tranclp-induct* [*to-set*]

lemmas *tranclp-induct2* =
tranclp-induct [*of* - (*ax*,*ay*) (*bx*,*by*), *split-rule*,
consumes 1, *case-names base step*]

lemmas *trancl-induct2* =
trancl-induct [*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*),
consumes 1, *case-names base step*]

lemma *tranclp-trans-induct*:
assumes *major*: *r*^++ *x y*
and *cases*: !!*x y*. *r x y* ==> *P x y*
!!*x y z*. [| *r*^++ *x y*; *P x y*; *r*^++ *y z*; *P y z* |] ==> *P x z*
shows *P x y*
— Another induction rule for trancl, incorporating transitivity
by (*iprover intro*: *major* [*THEN tranclp-induct*] *cases*)

lemmas *trancl-trans-induct* = *tranclp-trans-induct* [*to-set*]

lemma *tranclE* [*cases set*: *trancl*]:
assumes (*a*, *b*) : *r*^+
obtains
(*base*) (*a*, *b*) : *r*
| (*step*) *c* **where** (*a*, *c*) : *r*^+ **and** (*c*, *b*) : *r*
using *assms* **by** *cases simp-all*

lemma *trancl-Int-subset*: [| *r* ⊆ *s*; *r O* (*r*^+ ∩ *s*) ⊆ *s*|] ==> *r*^+ ⊆ *s*
apply (*rule subsetI*)
apply (*rule-tac p* = *x* **in** *PairE*)
apply *clarify*
apply (*erule trancl-induct*)
apply *auto*
done

lemma *trancl-unfold*: *r*^+ = *r Un r O r*^+
by (*auto intro*: *trancl-into-trancl elim*: *tranclE*)

Transitivity of r^+

lemma *trans-trancl* [*simp*]: *trans* (*r*^+)

proof (*rule transI*)
fix *x y z*
assume $(x, y) \in r\hat{}+$
assume $(y, z) \in r\hat{}+$
then show $(x, z) \in r\hat{}+$
proof *induct*
case (*base u*)
from ‹$(x, y) \in r\hat{}+$› **and** ‹$(y, u) \in r$›
show $(x, u) \in r\hat{}+$ **..**
next
case (*step u v*)
from ‹$(x, u) \in r\hat{}+$› **and** ‹$(u, v) \in r$›
show $(x, v) \in r\hat{}+$ **..**
qed
qed

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*, *standard*]

lemma *tranclp-trans*:
assumes *xy*: $r\hat{}++\ x\ y$
and *yz*: $r\hat{}++\ y\ z$
shows $r\hat{}++\ x\ z$ **using** *yz xy*
by *induct iprover+*

lemma *trancl-id* [*simp*]: *trans* $r \Longrightarrow r\hat{}+ = r$
apply *auto*
apply (*erule trancl-induct*)
apply *assumption*
apply (*unfold trans-def*)
apply *blast*
done

lemma *rtranclp-tranclp-tranclp*:
assumes $r\hat{}**\ x\ y$
shows !!*z*. $r\hat{}++\ y\ z \Longrightarrow r\hat{}++\ x\ z$ **using** *assms*
by *induct* (*iprover intro*: *tranclp-trans*)+

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: $r\ a\ b \Longrightarrow r\hat{}++\ b\ c \Longrightarrow r\hat{}++\ a\ c$
by (*erule tranclp-trans* [*OF tranclp.r-into-trancl*])

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *trancl-insert*:
$(insert\ (y, x)\ r)\hat{}+ = r\hat{}+ \cup \{(a, b).\ (a, y) \in r\hat{}* \wedge (x, b) \in r\hat{}*\}$
— primitive recursion for *trancl* over finite relations
apply (*rule equalityI*)
apply (*rule subsetI*)

```
 apply (simp only: split-tupled-all)
 apply (erule trancl-induct, blast)
apply (blast intro: rtrancl-into-trancl1 trancl-into-rtrancl r-into-trancl trancl-trans)
apply (rule subsetI)
apply (blast intro: trancl-mono rtrancl-mono
  [THEN [2] rev-subsetD] rtrancl-trancl-trancl rtrancl-into-trancl2)
done

lemma tranclp-converseI: (r^++)^--1 x y ==> (r^--1)^++ x y
apply (drule conversepD)
apply (erule tranclp-induct)
apply (iprover intro: conversepI tranclp-trans)+
done

lemmas trancl-converseI = tranclp-converseI [to-set]

lemma tranclp-converseD: (r^--1)^++ x y ==> (r^++)^--1 x y
apply (rule conversepI)
apply (erule tranclp-induct)
apply (iprover dest: conversepD intro: tranclp-trans)+
done

lemmas trancl-converseD = tranclp-converseD [to-set]

lemma tranclp-converse: (r^--1)^++ = (r^++)^--1
by (fastsimp simp add: expand-fun-eq
  intro!: tranclp-converseI dest!: tranclp-converseD)

lemmas trancl-converse = tranclp-converse [to-set]

lemma sym-trancl: sym r ==> sym (r^+)
by (simp only: sym-conv-converse-eq trancl-converse [symmetric])

lemma converse-tranclp-induct:
assumes major: r^++ a b
  and cases: !!y. r y b ==> P(y)
    !!y z.[| r y z;  r^++ z b;  P(z) |] ==> P(y)
shows P a
apply (rule tranclp-induct [OF tranclp-converseI, OF conversepI, OF major])
 apply (rule cases)
 apply (erule conversepD)
apply (blast intro: prems dest!: tranclp-converseD conversepD)
done

lemmas converse-trancl-induct = converse-tranclp-induct [to-set]

lemma tranclpD: R^++ x y ==> EX z. R x z ∧ R^** z y
apply (erule converse-tranclp-induct)
 apply auto
```

apply (*blast intro*: *rtranclp-trans*)
done

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *tranclD2*:
$(x, y) \in R^+ \Longrightarrow \exists z.\ (x, z) \in R^* \wedge (z, y) \in R$
by (*blast elim*: *tranclE intro*: *trancl-into-rtrancl*)

lemma *irrefl-tranclI*: $r\hat{}-1 \cap r\hat{}* = \{\} ==> (x, x) \notin r\hat{}+$
by (*blast elim*: *tranclE dest*: *trancl-into-rtrancl*)

lemma *irrefl-trancl-rD*: $!!X.\ ALL\ x.\ (x, x) \notin r\hat{}+ ==> (x, y) \in r ==> x \neq y$
by (*blast dest*: *r-into-trancl*)

lemma *trancl-subset-Sigma-aux*:
$(a, b) \in r\hat{}* ==> r \subseteq A \times A ==> a = b \vee a \in A$
by (*induct rule*: *rtrancl-induct*) *auto*

lemma *trancl-subset-Sigma*: $r \subseteq A \times A ==> r\hat{}+ \subseteq A \times A$
apply (*rule subsetI*)
apply (*simp only*: *split-tupled-all*)
apply (*erule tranclE*)
apply (*blast dest*!: *trancl-into-rtrancl trancl-subset-Sigma-aux*)+
done

lemma *reflcl-tranclp* [*simp*]: $(r\hat{}++)\hat{}== = r\hat{}**$
apply (*safe intro*!: *order-antisym*)
apply (*erule tranclp-into-rtranclp*)
apply (*blast elim*: *rtranclp.cases dest*: *rtranclp-into-tranclp1*)
done

lemmas *reflcl-trancl* [*simp*] = *reflcl-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r\hat{}=)\hat{}+ = r\hat{}*$
apply *safe*
apply (*drule trancl-into-rtrancl*, *simp*)
apply (*erule rtranclE*, *safe*)
apply (*rule r-into-trancl*, *simp*)
apply (*rule rtrancl-into-trancl1*)
apply (*erule rtrancl-reflcl* [*THEN equalityD2*, *THEN subsetD*], *fast*)
done

lemma *trancl-empty* [*simp*]: $\{\}\hat{}+ = \{\}$
by (*auto elim*: *trancl-induct*)

lemma *rtrancl-empty* [*simp*]: $\{\}\hat{}* = Id$
by (*rule subst* [*OF reflcl-trancl*]) *simp*

lemma *rtranclpD*: $R\hat{}** a\ b ==> a = b \vee a \neq b \wedge R\hat{}++ a\ b$
by (*force simp add*: *reflcl-tranclp* [*symmetric*] *simp del*: *reflcl-tranclp*)

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*:
$(x,y) \in R^* = (x=y \vee x\neq y \wedge (x,y) \in R^+)$
by (*fast elim*: *trancl-into-rtrancl dest*: *rtranclD*)

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: *Domain* $(R\hat{}*) = UNIV$
by *blast*

lemma *Range-rtrancl* [*simp*]: *Range* $(R\hat{}*) = UNIV$
by *blast*

lemma *rtrancl-Un-subset*: $(R\hat{}* \cup S\hat{}*) \subseteq (R\ Un\ S)\hat{}*$
by (*rule rtrancl-Un-rtrancl* [*THEN subst*]) *fast*

lemma *in-rtrancl-UnI*: $x \in R\hat{}* \vee x \in S\hat{}* ==> x \in (R \cup S)\hat{}*$
by (*blast intro*: *subsetD* [*OF rtrancl-Un-subset*])

lemma *trancl-domain* [*simp*]: *Domain* $(r\hat{}+) = Domain\ r$
by (*unfold Domain-def*) (*blast dest*: *tranclD*)

lemma *trancl-range* [*simp*]: *Range* $(r\hat{}+) = Range\ r$
unfolding *Range-def* **by**(*simp add*: *trancl-converse* [*symmetric*])

lemma *Not-Domain-rtrancl*:
$x \sim$: *Domain* $R ==> ((x,\ y) : R\hat{}*) = (x = y)$
apply *auto*
apply (*erule rev-mp*)
apply (*erule rtrancl-induct*)
apply *auto*
done

lemma *trancl-subset-Field2*: $r\hat{}+ <= Field\ r \times Field\ r$
apply *clarify*
apply (*erule trancl-induct*)
apply (*auto simp add*: *Field-def*)
done

lemma *finite-trancl*: *finite* $(r\hat{}+) =$ *finite* r
apply *auto*
prefer *2*
apply (*rule trancl-subset-Field2* [*THEN finite-subset*])
apply (*rule finite-SigmaI*)
prefer *3*
apply (*blast intro*: *r-into-trancl'* *finite-subset*)

apply (*auto simp add*: *finite-Field*)
done

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
⟦ *single-valued r*; (*x*,*y*) ∈ *r*^*; (*x*,*z*) ∈ *r*^* ⟧
⟹ (*y*,*z*) ∈ *r*^* ∨ (*z*,*y*) ∈ *r*^*
apply (*erule rtrancl-induct*)
apply *simp*
apply (*erule disjE*)
apply (*blast elim*:*converse-rtranclE dest*:*single-valuedD*)
apply(*blast intro*:*rtrancl-trans*)
done

lemma *r-r-into-trancl*: (*a*, *b*) ∈ *R* ==> (*b*, *c*) ∈ *R* ==> (*a*, *c*) ∈ *R*^+
by (*fast intro*: *trancl-trans*)

lemma *trancl-into-trancl* [*rule-format*]:
(*a*, *b*) ∈ *r*^+ ==> (*b*, *c*) ∈ *r* --> (*a*,*c*) ∈ *r*^+
apply (*erule trancl-induct*)
apply (*fast intro*: *r-r-into-trancl*)
apply (*fast intro*: *r-r-into-trancl trancl-trans*)
done

lemma *tranclp-rtranclp-tranclp*:
r^++ *a b* ==> *r*^** *b c* ==> *r*^++ *a c*
apply (*drule tranclpD*)
apply (*elim exE conjE*)
apply (*drule rtranclp-trans*, *assumption*)
apply (*drule rtranclp-into-tranclp2*, *assumption*, *assumption*)
done

lemmas *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl.trancl-into-trancl trancl-into-trancl2
rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

20.4 Setup of transitivity reasoner

ML ⟪

```
structure Trancl-Tac = Trancl-Tac-Fun (
  struct
    val r-into-trancl = @{thm trancl.r-into-trancl};
    val trancl-trans  = @{thm trancl-trans};
    val rtrancl-refl = @{thm rtrancl.rtrancl-refl};
    val r-into-rtrancl = @{thm r-into-rtrancl};
    val trancl-into-rtrancl = @{thm trancl-into-rtrancl};
    val rtrancl-trancl-trancl = @{thm rtrancl-trancl-trancl};
    val trancl-rtrancl-trancl = @{thm trancl-rtrancl-trancl};
    val rtrancl-trans = @{thm rtrancl-trans};

  fun decomp (@{const Trueprop} $ t) =
    let fun dec (Const (op :, -) $ (Const (Pair, -) $ a $ b) $ rel ) =
        let fun decr (Const (Transitive-Closure.rtrancl, - ) $ r) = (r,r*)
              | decr (Const (Transitive-Closure.trancl, - ) $ r)  = (r,r+)
              | decr r = (r,r);
            val (rel,r) = decr (Envir.beta-eta-contract rel);
        in SOME (a,b,rel,r) end
      | dec - =  NONE
    in dec t end
    | decomp - = NONE;

  end);

structure Tranclp-Tac = Trancl-Tac-Fun (
  struct
    val r-into-trancl = @{thm tranclp.r-into-trancl};
    val trancl-trans  = @{thm tranclp-trans};
    val rtrancl-refl = @{thm rtranclp.rtrancl-refl};
    val r-into-rtrancl = @{thm r-into-rtranclp};
    val trancl-into-rtrancl = @{thm tranclp-into-rtranclp};
    val rtrancl-trancl-trancl = @{thm rtranclp-tranclp-tranclp};
    val trancl-rtrancl-trancl = @{thm tranclp-rtranclp-tranclp};
    val rtrancl-trans = @{thm rtranclp-trans};

  fun decomp (@{const Trueprop} $ t) =
    let fun dec (rel $ a $ b) =
        let fun decr (Const (Transitive-Closure.rtranclp, - ) $ r) = (r,r*)
              | decr (Const (Transitive-Closure.tranclp, - ) $ r)  = (r,r+)
              | decr r = (r,r);
            val (rel,r) = decr rel;
        in SOME (a, b, rel, r) end
      | dec - =  NONE
    in dec t end
    | decomp - = NONE;
```

```
  end);
⟩⟩

declaration ⟨⟨ fn - =>
  Simplifier.map-ss (fn ss => ss
    addSolver (mk-solver Trancl (fn - => Trancl-Tac.trancl-tac))
    addSolver (mk-solver Rtrancl (fn - => Trancl-Tac.rtrancl-tac))
    addSolver (mk-solver Tranclp (fn - => Tranclp-Tac.trancl-tac))
    addSolver (mk-solver Rtranclp (fn - => Tranclp-Tac.rtrancl-tac)))
⟩⟩



method-setup trancl =
  ⟨⟨ Scan.succeed (K (SIMPLE-METHOD' Trancl-Tac.trancl-tac)) ⟩⟩
  ⟨⟨ simple transitivity reasoner ⟩⟩
method-setup rtrancl =
  ⟨⟨ Scan.succeed (K (SIMPLE-METHOD' Trancl-Tac.rtrancl-tac)) ⟩⟩
  ⟨⟨ simple transitivity reasoner ⟩⟩
method-setup tranclp =
  ⟨⟨ Scan.succeed (K (SIMPLE-METHOD' Tranclp-Tac.trancl-tac)) ⟩⟩
  ⟨⟨ simple transitivity reasoner (predicate version) ⟩⟩
method-setup rtranclp =
  ⟨⟨ Scan.succeed (K (SIMPLE-METHOD' Tranclp-Tac.rtrancl-tac)) ⟩⟩
  ⟨⟨ simple transitivity reasoner (predicate version) ⟩⟩

end
```

21 Wellfounded: Well-founded Recursion

```
theory Wellfounded
imports Finite-Set Transitive-Closure Nat
uses (Tools/function-package/size.ML)
begin
```

21.1 Basic Definitions

```
inductive
  wfrec-rel :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b => bool
  for R :: ('a * 'a) set
  and F :: ('a => 'b) => 'a => 'b
where
  wfrecI: ALL z. (z, x) : R --> wfrec-rel R F z (g z) ==>
          wfrec-rel R F x (F g x)

constdefs
  wf         :: ('a * 'a)set => bool
  wf(r) == (!P. (!x. (!y. (y,x):r --> P(y)) --> P(x)) --> (!x. P(x)))
```

wfP :: (*'a* => *'a* => *bool*) => *bool*
wfP r == *wf* {(*x*, *y*). *r x y*}

acyclic :: (*'a***'a*)*set* => *bool*
acyclic r == !*x*. (*x*,*x*) ~: *r*^+

cut :: (*'a* => *'b*) => (*'a* * *'a*)*set* => *'a* => *'a* => *'b*
cut f r x == (%*y*. *if* (*y*,*x*):*r then f y else undefined*)

adm-wf :: (*'a* * *'a*) *set* => ((*'a* => *'b*) => *'a* => *'b*) => *bool*
adm-wf R F == *ALL f g x*.
(*ALL z*. (*z*, *x*) : *R* --> *f z* = *g z*) --> *F f x* = *F g x*

wfrec :: (*'a* * *'a*) *set* => ((*'a* => *'b*) => *'a* => *'b*) => *'a* => *'b*
[*code del*]: *wfrec R F* == %*x*. *THE y*. *wfrec-rel R* (%*f x*. *F* (*cut f R x*) *x*) *x y*

abbreviation *acyclicP* :: (*'a* => *'a* => *bool*) => *bool* **where**
acyclicP r == *acyclic* {(*x*, *y*). *r x y*}

lemma *wfP-wf-eq* [*pred-set-conv*]: *wfP* (λ*x y*. (*x*, *y*) ∈ *r*) = *wf r*
by (*simp add*: *wfP-def*)

lemma *wfUNIVI*:
(!!*P x*. (*ALL x*. (*ALL y*. (*y*,*x*) : *r* --> *P*(*y*)) --> *P*(*x*)) ==> *P*(*x*)) ==> *wf*(*r*)
unfolding *wf-def* **by** *blast*

lemmas *wfPUNIVI* = *wfUNIVI* [*to-pred*]

Restriction to domain *A* and range *B*. If *r* is well-founded over their intersection, then *wf r*

lemma *wfI*:
[| *r* ⊆ *A* <*> *B*;
!!*x P*. [|∀ *x*. (∀ *y*. (*y*,*x*) : *r* --> *P y*) --> *P x*; *x* : *A*; *x* : *B* |] ==> *P x* |]
==> *wf r*
unfolding *wf-def* **by** *blast*

lemma *wf-induct*:
[| *wf*(*r*);
!!*x*.[| *ALL y*. (*y*,*x*): *r* --> *P*(*y*) |] ==> *P*(*x*)
|] ==> *P*(*a*)
unfolding *wf-def* **by** *blast*

lemmas *wfP-induct* = *wf-induct* [*to-pred*]

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *consumes 1*, *case-names less*, *induct set*: *wf*]

lemmas *wfP-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set*: *wfP*]

lemma *wf-not-sym*: *wf r* ==> (*a*, *x*) : *r* ==> (*x*, *a*) ~: *r*
by (*induct a arbitrary*: *x set*: *wf*) *blast*

lemmas *wf-asym* = *wf-not-sym* [*elim-format*]

lemma *wf-not-refl* [*simp*]: *wf r* ==> (*a*, *a*) ~: *r*
by (*blast elim*: *wf-asym*)

lemmas *wf-irrefl* = *wf-not-refl* [*elim-format*]

lemma *wf-wellorderI*:
assumes *wf*: *wf* {(*x*::'*a*::*ord*, *y*). *x* < *y*}
assumes *lin*: *OFCLASS*('*a*::*ord*, *linorder-class*)
shows *OFCLASS*('*a*::*ord*, *wellorder-class*)
using *lin* **by** (*rule wellorder-class.intro*)
(*blast intro*: *wellorder-axioms.intro wf-induct-rule* [*OF wf*])

lemma (**in** *wellorder*) *wf*:
wf {(*x*, *y*). *x* < *y*}
unfolding *wf-def* **by** (*blast intro*: *less-induct*)

21.2 Basic Results

transitive closure of a well-founded relation is well-founded!

lemma *wf-trancl*:
assumes *wf r*
shows *wf* (*r*^+)
proof −
{
fix *P* **and** *x*
assume *induct-step*: !!*x*. (!!*y*. (*y*, *x*) : *r*^+ ==> *P y*) ==> *P x*
have *P x*
proof (*rule induct-step*)
fix *y* **assume** (*y*, *x*) : *r*^+
with ‹*wf r*› **show** *P y*
proof (*induct x arbitrary*: *y*)
case (*less x*)
note *hyp* = ‹⋀*x'* *y'*. (*x'*, *x*) : *r* ==> (*y'*, *x'*) : *r*^+ ==> *P y'*›
from ‹(*y*, *x*) : *r*^+› **show** *P y*
proof *cases*
case *base*
show *P y*
proof (*rule induct-step*)
fix *y'* **assume** (*y'*, *y*) : *r*^+
with ‹(*y*, *x*) : *r*› **show** *P y'* **by** (*rule hyp* [*of y y'*])

```
      qed
    next
      case step
      then obtain x' where (x', x) : r and (y, x') : r^+ by simp
      then show P y by (rule hyp [of x' y])
    qed
  qed
 qed
} then show ?thesis unfolding wf-def by blast
qed

lemmas wfP-trancl = wf-trancl [to-pred]

lemma wf-converse-trancl: wf (r^-1) ==> wf ((r^+)^-1)
 apply (subst trancl-converse [symmetric])
 apply (erule wf-trancl)
 done
```

Minimal-element characterization of well-foundedness

```
lemma wf-eq-minimal: wf r = (∀Q x. x∈Q --> (∃z∈Q. ∀y. (y,z)∈r -->
y∉Q))
proof (intro iffI strip)
 fix Q :: 'a set and x
 assume wf r and x ∈ Q
 then show ∃z∈Q. ∀y. (y, z) ∈ r ⟶ y ∉ Q
  unfolding wf-def
  by (blast dest: spec [of - %x. x∈Q ⟶ (∃z∈Q. ∀y. (y,z) ∈ r ⟶ y∉Q)])
next
 assume 1: ∀Q x. x ∈ Q ⟶ (∃z∈Q. ∀y. (y, z) ∈ r ⟶ y ∉ Q)
 show wf r
 proof (rule wfUNIVI)
  fix P :: 'a ⇒ bool and x
  assume 2: ∀x. (∀y. (y, x) ∈ r ⟶ P y) ⟶ P x
  let ?Q = {x. ¬ P x}
  have x ∈ ?Q ⟶ (∃z ∈ ?Q. ∀y. (y, z) ∈ r ⟶ y ∉ ?Q)
   by (rule 1 [THEN spec, THEN spec])
  then have ¬ P x ⟶ (∃z. ¬ P z ∧ (∀y. (y, z) ∈ r ⟶ P y)) by simp
  with 2 have ¬ P x ⟶ (∃z. ¬ P z ∧ P z) by fast
  then show P x by simp
 qed
qed

lemma wfE-min:
 assumes wf R x ∈ Q
 obtains z where z ∈ Q ⋀y. (y, z) ∈ R ⟹ y ∉ Q
 using assms unfolding wf-eq-minimal by blast

lemma wfI-min:
 (⋀x Q. x ∈ Q ⟹ ∃z∈Q. ∀y. (y, z) ∈ R ⟶ y ∉ Q)
```

$\Longrightarrow$ *wf R*
unfolding *wf-eq-minimal* **by** *blast*

lemmas *wfP-eq-minimal* = *wf-eq-minimal* [*to-pred*]

Well-foundedness of subsets

lemma *wf-subset*: [| *wf*(*r*); *p*<=*r* |] ==> *wf*(*p*)
apply (*simp* (*no-asm-use*) *add*: *wf-eq-minimal*)
apply *fast*
done

lemmas *wfP-subset* = *wf-subset* [*to-pred*]

Well-foundedness of the empty relation

lemma *wf-empty* [*iff*]: *wf*({})
by (*simp add*: *wf-def*)

lemmas *wfP-empty* [*iff*] =
wf-empty [*to-pred bot-empty-eq2*, *simplified bot-fun-eq bot-bool-eq*]

lemma *wf-Int1*: *wf r* ==> *wf* (*r Int r'*)
apply (*erule wf-subset*)
apply (*rule Int-lower1*)
done

lemma *wf-Int2*: *wf r* ==> *wf* (*r' Int r*)
apply (*erule wf-subset*)
apply (*rule Int-lower2*)
done

Well-foundedness of insert

lemma *wf-insert* [*iff*]: *wf*(*insert* (*y*,*x*) *r*) = (*wf*(*r*) & (*x*,*y*) ~: *r*^*)
apply (*rule iffI*)
apply (*blast elim*: *wf-trancl* [*THEN wf-irrefl*]
intro: *rtrancl-into-trancl1 wf-subset*
rtrancl-mono [*THEN* [*2*] *rev-subsetD*])
apply (*simp add*: *wf-eq-minimal*, *safe*)
apply (*rule allE*, *assumption*, *erule impE*, *blast*)
apply (*erule bexE*)
apply (*rename-tac a*, *case-tac a* = *x*)
prefer *2*
apply *blast*
apply (*case-tac y*:*Q*)
prefer *2* **apply** *blast*
apply (*rule-tac x* = {*z*. *z*:*Q* & (*z*,*y*) : *r*^*} **in** *allE*)
apply *assumption*
apply (*erule-tac V* = *ALL Q*. (*EX x*. *x* : *Q*) −−> *?P Q* **in** *thin-rl*)
— essential for speed

Blast with new substOccur fails

apply (*fast intro*: *converse-rtrancl-into-rtrancl*)
done

Well-foundedness of image

lemma *wf-prod-fun-image*: [| *wf r*; *inj f* |] ==> *wf*(*prod-fun f f* ‘ *r*)
apply (*simp only*: *wf-eq-minimal*, *clarify*)
apply (*case-tac EX p*. *f p* : *Q*)
apply (*erule-tac x* = {*p*. *f p* : *Q*} **in** *allE*)
apply (*fast dest*: *inj-onD*, *blast*)
done

21.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:
assumes *wf R wf S*
assumes $S\ O\ R \subseteq R$
shows *wf* $(R \cup S)$
proof (*rule wfI-min*)
fix $x :: 'a$ **and** Q
let $?Q' = \{x \in Q.\ \forall y.\ (y, x) \in R \longrightarrow y \notin Q\}$
assume $x \in Q$
obtain *a* **where** $a \in ?Q'$
by (*rule wfE-min* [*OF* ‹*wf R*› ‹$x \in Q$›]) *blast*
with ‹*wf S*›
obtain *z* **where** $z \in ?Q'$ **and** *zmin*: $\bigwedge y.\ (y, z) \in S \Longrightarrow y \notin ?Q'$ **by** (*erule wfE-min*)
{
fix *y* **assume** $(y, z) \in S$
then have $y \notin ?Q'$ **by** (*rule zmin*)

have $y \notin Q$
proof
assume $y \in Q$
with ‹$y \notin ?Q'$›
obtain *w* **where** $(w, y) \in R$ **and** $w \in Q$ **by** *auto*
from ‹$(w, y) \in R$› ‹$(y, z) \in S$› **have** $(w, z) \in S\ O\ R$ **by** (*rule rel-compI*)
with ‹$S\ O\ R \subseteq R$› **have** $(w, z) \in R$ **..**
with ‹$z \in ?Q'$› **have** $w \notin Q$ **by** *blast*
with ‹$w \in Q$› **show** *False* **by** *contradiction*
qed
}
with ‹$z \in ?Q'$› **show** $\exists z{\in}Q.\ \forall y.\ (y, z) \in R \cup S \longrightarrow y \notin Q$ **by** *blast*
qed

Well-foundedness of indexed union with disjoint domains and ranges

lemma *wf-UN*: [| *ALL i*:*I*. *wf*(*r i*);
ALL i:*I*. *ALL j*:*I*. *r i* ~= *r j* --> *Domain*(*r i*) *Int Range*(*r j*) = {}
|] ==> *wf*(*UN i*:*I*. *r i*)

apply (*simp only*: *wf-eq-minimal*, *clarify*)
apply (*rename-tac A a*, *case-tac EX i:I. EX a:A. EX b:A.* (*b*,*a*) : *r i*)
prefer *2*
apply *force*
apply *clarify*
apply (*drule bspec*, *assumption*)
apply (*erule-tac x*={*a. a*:*A* & (*EX b*:*A.* (*b*,*a*) : *r i*) } **in** *allE*)
apply (*blast elim*!: *allE*)
done

lemmas *wfP-SUP* = *wf-UN* [**where** *I*=*UNIV* **and** *r*=λ*i.* {(*x*, *y*). *r i x y*},
to-pred SUP-UN-eq2 bot-empty-eq pred-equals-eq, *simplified*, *standard*]

lemma *wf-Union*:
[| *ALL r*:*R. wf r*;
ALL r:*R. ALL s*:*R. r* ~= *s* −−> *Domain r Int Range s* = {}
|] ==> *wf*(*Union R*)
apply (*simp add*: *Union-def*)
apply (*blast intro*: *wf-UN*)
done

lemma *wf-Un*:
[| *wf r*; *wf s*; *Domain r Int Range s* = {} |] ==> *wf*(*r Un s*)
using *wf-union-compatible*[*of s r*]
by (*auto simp*: *Un-ac*)

lemma *wf-union-merge*:
wf (*R* ∪ *S*) = *wf* (*R O R* ∪ *R O S* ∪ *S*) (**is** *wf ?A* = *wf ?B*)
proof
assume *wf ?A*
with *wf-trancl* **have** *wfT*: *wf* (*?A* ^+) **.**
moreover have *?B* ⊆ *?A* ^+
by (*subst trancl-unfold*, *subst trancl-unfold*) *blast*
ultimately show *wf ?B* **by** (*rule wf-subset*)
next
assume *wf ?B*

show *wf ?A*
proof (*rule wfI-min*)
fix *Q* :: *'a set* **and** *x*
assume *x* ∈ *Q*

with ‹*wf ?B*›
obtain *z* **where** *z* ∈ *Q* **and** ⋀*y.* (*y*, *z*) ∈ *?B* ⟹ *y* ∉ *Q*
by (*erule wfE-min*)
then have *A1*: ⋀*y.* (*y*, *z*) ∈ *R O R* ⟹ *y* ∉ *Q*
and *A2*: ⋀*y.* (*y*, *z*) ∈ *R O S* ⟹ *y* ∉ *Q*
and *A3*: ⋀*y.* (*y*, *z*) ∈ *S* ⟹ *y* ∉ *Q*

```
    by auto

  show ∃z∈Q. ∀y. (y, z) ∈ ?A ⟶ y ∉ Q
  proof (cases ∀y. (y, z) ∈ R ⟶ y ∉ Q)
    case True
    with ‹z ∈ Q› A3 show ?thesis by blast
  next
    case False
    then obtain z' where z'∈Q (z', z) ∈ R by blast

    have ∀y. (y, z') ∈ ?A ⟶ y ∉ Q
    proof (intro allI impI)
      fix y assume (y, z') ∈ ?A
      then show y ∉ Q
      proof
        assume (y, z') ∈ R
        then have (y, z) ∈ R O R using ‹(z', z) ∈ R› ..
        with A1 show y ∉ Q .
      next
        assume (y, z') ∈ S
        then have (y, z) ∈ R O S using ‹(z', z) ∈ R› ..
        with A2 show y ∉ Q .
      qed
    qed
    with ‹z' ∈ Q› show ?thesis ..
  qed
 qed
qed

lemma wf-comp-self: wf R = wf (R O R)  — special case
 by (rule wf-union-merge [where S = {}, simplified])
```

21.3.1 acyclic

```
lemma acyclicI: ALL x. (x, x) ~: r^+ ==> acyclic r
 by (simp add: acyclic-def)

lemma wf-acyclic: wf r ==> acyclic r
apply (simp add: acyclic-def)
apply (blast elim: wf-trancl [THEN wf-irrefl])
done

lemmas wfP-acyclicP = wf-acyclic [to-pred]

lemma acyclic-insert [iff]:
    acyclic(insert (y,x) r) = (acyclic r & (x,y) ~: r^*)
apply (simp add: acyclic-def trancl-insert)
apply (blast intro: rtrancl-trans)
done
```

lemma *acyclic-converse* [*iff*]: *acyclic*(*r*^−1) = *acyclic r*
by (*simp add*: *acyclic-def trancl-converse*)

lemmas *acyclicP-converse* [*iff*] = *acyclic-converse* [*to-pred*]

lemma *acyclic-impl-antisym-rtrancl*: *acyclic r* ==> *antisym*(*r*^*)
apply (*simp add*: *acyclic-def antisym-def*)
apply (*blast elim*: *rtranclE intro*: *rtrancl-into-trancl1 rtrancl-trancl-trancl*)
done

lemma *acyclic-subset*: [| *acyclic s*; *r* <= *s* |] ==> *acyclic r*
apply (*simp add*: *acyclic-def*)
apply (*blast intro*: *trancl-mono*)
done

Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf* [*rule-format*]: *finite r* ==> *acyclic r* −−> *wf r*
apply (*erule finite-induct*, *blast*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply *simp*
done

lemma *finite-acyclic-wf-converse*: [|*finite r*; *acyclic r*|] ==> *wf* (*r*^−1)
apply (*erule finite-converse* [*THEN iffD2*, *THEN finite-acyclic-wf*])
apply (*erule acyclic-converse* [*THEN iffD2*])
done

lemma *wf-iff-acyclic-if-finite*: *finite r* ==> *wf r* = *acyclic r*
by (*blast intro*: *finite-acyclic-wf wf-acyclic*)

21.4 Well-Founded Recursion

cut

lemma *cuts-eq*: (*cut f r x* = *cut g r x*) = (*ALL y*. (*y*,*x*):*r* −−> *f*(*y*)=*g*(*y*))
by (*simp add*: *expand-fun-eq cut-def*)

lemma *cut-apply*: (*x*,*a*):*r* ==> (*cut f r a*)(*x*) = *f*(*x*)
by (*simp add*: *cut-def*)

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application"

lemma *wfrec-unique*: [| *adm-wf R F*; *wf R* |] ==> *EX*! *y*. *wfrec-rel R F x y*
apply (*simp add*: *adm-wf-def*)
apply (*erule-tac a*=*x* **in** *wf-induct*)
apply (*rule ex1I*)

apply (*rule-tac g = %x. THE y. wfrec-rel R F x y* **in** *wfrec-rel.wfrecI*)
apply (*fast dest*!: *theI'*)
apply (*erule wfrec-rel.cases*, *simp*)
apply (*erule allE*, *erule allE*, *erule allE*, *erule mp*)
apply (*fast intro*: *the-equality* [*symmetric*])
done

lemma *adm-lemma*: *adm-wf R* (*%f x. F* (*cut f R x*) *x*)
apply (*simp add*: *adm-wf-def*)
apply (*intro strip*)
apply (*rule cuts-eq* [*THEN iffD2*, *THEN subst*], *assumption*)
apply (*rule refl*)
done

lemma *wfrec*: *wf*(*r*) ==> *wfrec r H a = H* (*cut* (*wfrec r H*) *r a*) *a*
apply (*simp add*: *wfrec-def*)
apply (*rule adm-lemma* [*THEN wfrec-unique*, *THEN the1-equality*], *assumption*)
apply (*rule wfrec-rel.wfrecI*)
apply (*intro strip*)
apply (*erule adm-lemma* [*THEN wfrec-unique*, *THEN theI'*])
done

21.5 Code generator setup

consts-code
wfrec (⟨**module**⟩*wfrec?*)
attach ⟪
fun wfrec f x = f (*wfrec f*) *x*;
⟫

21.6 *nat* is well-founded

lemma *less-nat-rel*: *op <* = (*λm n. n = Suc m*) ^++
proof (*rule ext*, *rule ext*, *rule iffI*)
fix *n m* :: *nat*
assume *m < n*
then show (*λm n. n = Suc m*) ^++ *m n*
proof (*induct n*)
case *0* **then show** *?case* **by** *auto*
next
case (*Suc n*) **then show** *?case*
by (*auto simp add*: *less-Suc-eq-le le-less intro*: *tranclp.trancl-into-trancl*)
qed
next
fix *n m* :: *nat*
assume (*λm n. n = Suc m*) ^++ *m n*
then show *m < n*
by (*induct n*)
(*simp-all add*: *less-Suc-eq-le reflexive le-less*)
qed

definition
pred-nat :: $(nat * nat)$ *set* **where**
pred-nat = $\{(m, n).\ n = Suc\ m\}$

definition
less-than :: $(nat * nat)$ *set* **where**
less-than = *pred-nat*^+

lemma *less-eq*: $(m, n) \in$ *pred-nat*^+ $\longleftrightarrow$ $m < n$
unfolding *less-nat-rel pred-nat-def trancl-def* **by** *simp*

lemma *pred-nat-trancl-eq-le*:
$(m, n) \in$ *pred-nat*^* $\longleftrightarrow$ $m \leq n$
unfolding *less-eq rtrancl-eq-or-trancl* **by** *auto*

lemma *wf-pred-nat*: *wf pred-nat*
apply (*unfold wf-def pred-nat-def*, *clarify*)
apply (*induct-tac x*, *blast*+)
done

lemma *wf-less-than* [*iff*]: *wf less-than*
by (*simp add*: *less-than-def wf-pred-nat* [*THEN wf-trancl*])

lemma *trans-less-than* [*iff*]: *trans less-than*
by (*simp add*: *less-than-def trans-trancl*)

lemma *less-than-iff* [*iff*]: ((*x*,*y*): *less-than*) = (*x*<*y*)
by (*simp add*: *less-than-def less-eq*)

lemma *wf-less*: *wf* $\{(x, y{::}nat).\ x < y\}$
using *wf-less-than* **by** (*simp add*: *less-than-def less-eq* [*symmetric*])

21.7 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [**?**].

inductive-set
acc :: $('a * 'a)$ *set* => $'a$ *set*
for *r* :: $('a * 'a)$ *set*
where
accI: $(!!y.\ (y, x) : r$ ==> $y : acc\ r)$ ==> $x : acc\ r$

abbreviation
termip :: $('a$ => $'a$ => *bool*) => $'a$ => *bool* **where**
termip r == *accp* (r^{-1-1})

abbreviation
termi :: $('a * 'a)$ *set* => $'a$ *set* **where**
termi r == *acc* (r^{-1})

lemmas *accpI* = *accp.accI*

Induction rules

theorem *accp-induct*:
assumes *major*: *accp r a*
assumes *hyp*: !!*x*. *accp r x* ==> ∀ *y*. *r y x* −−> *P y* ==> *P x*
shows *P a*
apply (*rule major* [*THEN accp.induct*])
apply (*rule hyp*)
apply (*rule accp.accI*)
apply *fast*
apply *fast*
done

theorems *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: *accp r b* ==> *r a b* ==> *accp r a*
apply (*erule accp.cases*)
apply *fast*
done

lemma *not-accp-down*:
assumes *na*: ¬ *accp R x*
obtains *z* **where** *R z x* **and** ¬ *accp R z*
proof −
assume *a*: ⋀*z*. ⟦*R z x*; ¬ *accp R z*⟧ ⟹ *thesis*

show *thesis*
proof (*cases* ∀ *z*. *R z x* ⟶ *accp R z*)
case *True*
hence ⋀*z*. *R z x* ⟹ *accp R z* **by** *auto*
hence *accp R x*
by (*rule accp.accI*)
with *na* **show** *thesis* **..**
next
case *False* **then obtain** *z* **where** *R z x* **and** ¬ *accp R z*
by *auto*
with *a* **show** *thesis* **.**
qed
qed

lemma *accp-downwards-aux*: *r*** *b a* ==> *accp r a* −−> *accp r b*
apply (*erule rtranclp-induct*)
apply *blast*
apply (*blast dest*: *accp-downward*)
done

theorem *accp-downwards*: *accp r a* ==> *r*** *b a* ==> *accp r b*

apply (*blast dest*: *accp-downwards-aux*)
done

theorem *accp-wfPI*: $\forall x.\ accp\ r\ x \Longrightarrow wfP\ r$
apply (*rule wfPUNIVI*)
apply (*induct-tac P x rule*: *accp-induct*)
apply *blast*
apply *blast*
done

theorem *accp-wfPD*: $wfP\ r \Longrightarrow accp\ r\ x$
apply (*erule wfP-induct-rule*)
apply (*rule accp.accI*)
apply *blast*
done

theorem *wfP-accp-iff*: $wfP\ r = (\forall x.\ accp\ r\ x)$
apply (*blast intro*: *accp-wfPI dest*: *accp-wfPD*)
done

Smaller relations have bigger accessible parts:

lemma *accp-subset*:
assumes *sub*: $R1 \leq R2$
shows $accp\ R2 \leq accp\ R1$
proof (*rule predicate1I*)
fix x **assume** $accp\ R2\ x$
then show $accp\ R1\ x$
proof (*induct x*)
fix x
assume *ih*: $\bigwedge y.\ R2\ y\ x \Longrightarrow accp\ R1\ y$
with *sub* **show** $accp\ R1\ x$
by (*blast intro*: *accp.accI*)
qed
qed

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:
assumes *subset*: $D \leq accp\ R$
and *dcl*: $\bigwedge x\ z.\ [\![D\ x;\ R\ z\ x]\!] \Longrightarrow D\ z$
and $D\ x$
and *istep*: $\bigwedge x.\ [\![D\ x;\ (\bigwedge z.\ R\ z\ x \Longrightarrow P\ z)]\!] \Longrightarrow P\ x$
shows $P\ x$
proof −
from *subset* **and** ‹$D\ x$›
have $accp\ R\ x$ **..**
then show $P\ x$ **using** ‹$D\ x$›
proof (*induct x*)
fix x

assume $D\ x$
and $\bigwedge y.\ R\ y\ x \Longrightarrow D\ y \Longrightarrow P\ y$
with *dcl* **and** *istep* **show** $P\ x$ **by** *blast*
qed
qed

Set versions of the above theorems

lemmas *acc-induct* = *accp-induct* [*to-set*]

lemmas *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

lemmas *acc-downward* = *accp-downward* [*to-set*]

lemmas *not-acc-down* = *not-accp-down* [*to-set*]

lemmas *acc-downwards-aux* = *accp-downwards-aux* [*to-set*]

lemmas *acc-downwards* = *accp-downwards* [*to-set*]

lemmas *acc-wfI* = *accp-wfPI* [*to-set*]

lemmas *acc-wfD* = *accp-wfPD* [*to-set*]

lemmas *wf-acc-iff* = *wfP-accp-iff* [*to-set*]

lemmas *acc-subset* = *accp-subset* [*to-set pred-subset-eq*]

lemmas *acc-subset-induct* = *accp-subset-induct* [*to-set pred-subset-eq*]

21.8 Tools for building wellfounded relations

Inverse Image

lemma *wf-inv-image* [*simp*,*intro*!]: *wf*(*r*) ==> *wf*(*inv-image r* (*f*::'*a*=>'*b*))
apply (*simp* (*no-asm-use*) *add*: *inv-image-def wf-eq-minimal*)
apply *clarify*
apply (*subgoal-tac EX* (*w*::'*b*) . *w* : {*w*. *EX* (*x*::'*a*) . *x*: *Q* & (*f x* = *w*) })
prefer *2* **apply** (*blast del*: *allE*)
apply (*erule allE*)
apply (*erule* (*1*) *notE impE*)
apply *blast*
done

lemma *in-inv-image*[*simp*]: ((*x*,*y*) : *inv-image r f*) = ((*f x*, *f y*) : *r*)
by (*auto simp*:*inv-image-def*)

Measure functions into *nat*

definition *measure* :: ('*a* => *nat*) => ('*a* * '*a*)*set*
where *measure* == *inv-image less-than*

lemma *in-measure*[*simp*]: $((x,y) : measure\ f) = (f\ x < f\ y)$
by (*simp add*:*measure-def*)

lemma *wf-measure* [*iff*]: *wf* (*measure f*)
apply (*unfold measure-def*)
apply (*rule wf-less-than* [*THEN wf-inv-image*])
done

Lexicographic combinations

definition
lex-prod :: [('*a*×'*a*)*set*, ('*b*×'*b*)*set*] => (('*a*×'*b*)×('*a*×'*b*))*set*
(**infixr** <*lex*> 80)
where
ra <*lex*> *rb* == {((*a*,*b*),(*a*',*b*')). (*a*,*a*') : *ra* | *a*=*a*' & (*b*,*b*') : *rb*}

lemma *wf-lex-prod* [*intro*!]: [| *wf*(*ra*); *wf*(*rb*) |] ==> *wf*(*ra* <*lex*> *rb*)
apply (*unfold wf-def lex-prod-def*)
apply (*rule allI*, *rule impI*)
apply (*simp* (*no-asm-use*) *only*: *split-paired-All*)
apply (*drule spec*, *erule mp*)
apply (*rule allI*, *rule impI*)
apply (*drule spec*, *erule mp*, *blast*)
done

lemma *in-lex-prod*[*simp*]:
((*a*,*b*),(*a*',*b*')): *r* <*lex*> *s*) = ((*a*,*a*'): *r* ∨ (*a* = *a*' ∧ (*b*, *b*') : *s*))
by (*auto simp*:*lex-prod-def*)

op <*lex*> preserves transitivity

lemma *trans-lex-prod* [*intro*!]:
[| *trans R1*; *trans R2* |] ==> *trans* (*R1* <*lex*> *R2*)
by (*unfold trans-def lex-prod-def*, *blast*)

lexicographic combinations with measure functions

definition
mlex-prod :: ('*a* ⇒ *nat*) ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* (**infixr** <*mlex*> 80)
where
f <*mlex*> *R* = *inv-image* (*less-than* <*lex*> *R*) (%*x*. (*f x*, *x*))

lemma *wf-mlex*: *wf R* ⟹ *wf* (*f* <*mlex*> *R*)
unfolding *mlex-prod-def*
by *auto*

lemma *mlex-less*: *f x* < *f y* ⟹ (*x*, *y*) ∈ *f* <*mlex*> *R*
unfolding *mlex-prod-def* **by** *simp*

lemma *mlex-leq*: *f x* ≤ *f y* ⟹ (*x*, *y*) ∈ *R* ⟹ (*x*, *y*) ∈ *f* <*mlex*> *R*
unfolding *mlex-prod-def* **by** *auto*

proper subset relation on finite sets

definition *finite-psubset* :: (*'a set* ∗ *'a set*) *set*
where *finite-psubset* == {(*A*,*B*). *A* < *B* & *finite B*}

lemma *wf-finite-psubset*[*simp*]: *wf*(*finite-psubset*)
apply (*unfold finite-psubset-def*)
apply (*rule wf-measure* [*THEN wf-subset*])
apply (*simp add*: *measure-def inv-image-def less-than-def less-eq*)
apply (*fast elim*!: *psubset-card-mono*)
done

lemma *trans-finite-psubset*: *trans finite-psubset*
by (*simp add*: *finite-psubset-def less-le trans-def*, *blast*)

lemma *in-finite-psubset*[*simp*]: (*A*, *B*) ∈ *finite-psubset* = (*A* < *B* & *finite B*)
unfolding *finite-psubset-def* **by** *auto*

max- and min-extension of order to finite sets

inductive-set *max-ext* :: (*'a* × *'a*) *set* ⇒ (*'a set* × *'a set*) *set*
for *R* :: (*'a* × *'a*) *set*
where
max-extI[*intro*]: *finite X* ⟹ *finite Y* ⟹ *Y* ≠ {} ⟹ (⋀*x*. *x* ∈ *X* ⟹ ∃ *y*∈*Y*. (*x*, *y*) ∈ *R*) ⟹ (*X*, *Y*) ∈ *max-ext R*

lemma *max-ext-wf*:
assumes *wf*: *wf r*
shows *wf* (*max-ext r*)
proof (*rule acc-wfI*, *intro allI*)
fix *M* **show** *M* ∈ *acc* (*max-ext r*) (**is** - ∈ *?W*)
proof *cases*
assume *finite M*
thus *?thesis*
proof (*induct M*)
show {} ∈ *?W*
by (*rule accI*) (*auto elim*: *max-ext.cases*)
next
fix *M a* **assume** *M* ∈ *?W finite M*
with *wf* **show** *insert a M* ∈ *?W*
proof (*induct arbitrary*: *M*)
fix *M a*
assume *M* ∈ *?W* **and** [*intro*]: *finite M*
assume *hyp*: ⋀*b M*. (*b*, *a*) ∈ *r* ⟹ *M* ∈ *?W* ⟹ *finite M* ⟹ *insert b M* ∈ *?W*
{
fix *N M* :: *'a set*
assume *finite N finite M*
then
have ⟦*M* ∈ *?W* ; (⋀*y*. *y* ∈ *N* ⟹ (*y*, *a*) ∈ *r*)⟧ ⟹ *N* ∪ *M* ∈ *?W*
by (*induct N arbitrary*: *M*) (*auto simp*: *hyp*)

```
      }
      note add-less = this

      show insert a M ∈ ?W
      proof (rule accI)
        fix N assume Nless: (N, insert a M) ∈ max-ext r
        hence asm1: ⋀x. x ∈ N ⟹ (x, a) ∈ r ∨ (∃ y ∈ M. (x, y) ∈ r)
          by (auto elim!: max-ext.cases)

        let ?N1 = { n ∈ N. (n, a) ∈ r }
        let ?N2 = { n ∈ N. (n, a) ∉ r }
        have N: ?N1 ∪ ?N2 = N by (rule set-ext) auto
        from Nless have finite N by (auto elim: max-ext.cases)
        then have finites: finite ?N1 finite ?N2 by auto

        have ?N2 ∈ ?W
        proof cases
          assume [simp]: M = {}
          have Mw: {} ∈ ?W by (rule accI) (auto elim: max-ext.cases)

          from asm1 have ?N2 = {} by auto
          with Mw show ?N2 ∈ ?W by (simp only:)
        next
          assume M ≠ {}
          have N2: (?N2, M) ∈ max-ext r
            by (rule max-extI[OF - - ‹M ≠ {}›]) (insert asm1, auto intro: finites)

          with ‹M ∈ ?W› show ?N2 ∈ ?W by (rule acc-downward)
        qed
        with finites have ?N1 ∪ ?N2 ∈ ?W
          by (rule add-less) simp
        then show N ∈ ?W by (simp only: N)
      qed
    qed
  qed
next
  assume [simp]: ¬ finite M
  show ?thesis
    by (rule accI) (auto elim: max-ext.cases)
qed
qed

lemma max-ext-additive:
 (A, B) ∈ max-ext R ⟹ (C, D) ∈ max-ext R ⟹
  (A ∪ C, B ∪ D) ∈ max-ext R
by (force elim!: max-ext.cases)


definition
```

min-ext :: $('a \times 'a)$ *set* $\Rightarrow$ $('a$ *set* $\times$ $'a$ *set*) *set*
where
[*code del*]: *min-ext* $r = \{(X,\ Y) \mid X\ Y.\ X \neq \{\} \wedge (\forall y \in Y.\ (\exists x \in X.\ (x,\ y) \in r))\}$

lemma *min-ext-wf*:
assumes *wf* r
shows *wf* (*min-ext* r)
proof (*rule wfI-min*)
fix Q :: $'a$ *set set*
fix x
assume *nonempty*: $x \in Q$
show $\exists m \in Q.\ (\forall\ n.\ (n,\ m) \in$ *min-ext* $r \longrightarrow n \notin Q)$
proof *cases*
assume $Q = \{\{\}\}$ **thus** *?thesis* **by** (*simp add*: *min-ext-def*)
next
assume $Q \neq \{\{\}\}$
with *nonempty*
obtain $e\ x$ **where** $x \in Q\ e \in x$ **by** *force*
then have *eU*: $e \in \bigcup Q$ **by** *auto*
with ‹*wf* r›
obtain z **where** z: $z \in \bigcup Q\ \bigwedge y.\ (y,\ z) \in r \Longrightarrow y \notin \bigcup Q$
by (*erule wfE-min*)
from z **obtain** m **where** $m \in Q\ z \in m$ **by** *auto*
from ‹$m \in Q$›
show *?thesis*
proof (*rule*, *intro bexI allI impI*)
fix n
assume *smaller*: $(n,\ m) \in$ *min-ext* r
with ‹$z \in m$› **obtain** y **where** y: $y \in n\ (y,\ z) \in r$ **by** (*auto simp*: *min-ext-def*)
then show $n \notin Q$ **using** $z(2)$ **by** *auto*
qed
qed
qed

Wellfoundedness of *same-fst*

definition
same-fst :: $('a$ => *bool*$)$ => $('a$ => $('b * 'b)$*set*$)$ => $(('a*'b)*('a*'b))$*set*
where
same-fst P R == {((x',y'),(x,y)) . x'=x & P x & (y',y) : R x}
— For *rec-def* declarations where the first n parameters stay unchanged in the recursive call.

lemma *same-fstI* [*intro*!]:
[| P x; (y',y) : R x |] ==> ((x,y'),(x,y)) : *same-fst P R*
by (*simp add*: *same-fst-def*)

lemma *wf-same-fst*:
assumes *prem*: (!!x. P x ==> *wf*(R x))

shows *wf*(*same-fst P R*)
apply (*simp cong del*: *imp-cong add*: *wf-def same-fst-def*)
apply (*intro strip*)
apply (*rename-tac a b*)
apply (*case-tac wf* (*R a*))
apply (*erule-tac a* = *b* **in** *wf-induct*, *blast*)
apply (*blast intro*: *prem*)
done

21.9 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.

This material does not appear to be used any longer.

lemma *sequence-trans*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^* |] ==> (*f* (*i*+*k*), *f i*) : *r*^*
by (*induct k*) (*auto intro*: *rtrancl-trans*)

lemma *wf-weak-decr-stable*:
assumes *as*: *ALL i*. (*f* (*Suc i*), *f i*) : *r*^* *wf* (*r*^+)
shows *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
proof −
have *lem*: !!*x*. [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^*; *wf* (*r*^+) |]
==> *ALL m*. *f m* = *x* −−> (*EX i*. *ALL k*. *f* (*m*+*i*+*k*) = *f* (*m*+*i*))
apply (*erule wf-induct*, *clarify*)
apply (*case-tac EX j*. (*f* (*m*+*j*), *f m*) : *r*^+)
apply *clarify*
apply (*subgoal-tac EX i*. *ALL k*. *f* ((*m*+*j*) +*i*+*k*) = *f* ((*m*+*j*) +*i*))
apply *clarify*
apply (*rule-tac x* = *j*+*i* **in** *exI*)
apply (*simp add*: *add-ac*, *blast*)
apply (*rule-tac x* = *0* **in** *exI*, *clarsimp*)
apply (*drule-tac i* = *m* **and** *k* = *k* **in** *sequence-trans*)
apply (*blast elim*: *rtranclE dest*: *rtrancl-into-trancl1*)
done

from *lem*[*OF as*, *THEN spec*, *of 0*, *simplified*]
show *?thesis* **by** *auto*
qed

lemma *weak-decr-stable*:
ALL i. *f* (*Suc i*) <= ((*f i*)::*nat*) ==> *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
apply (*rule-tac r* = *pred-nat* **in** *wf-weak-decr-stable*)
apply (*simp add*: *pred-nat-trancl-eq-le*)
apply (*intro wf-trancl wf-pred-nat*)
done

21.10 size of a datatype value

use *Tools/function-package/size.ML*

setup *Size.setup*

lemma *size-bool* [*code*]:
size (*b*::*bool*) = *0* **by** (*cases b*) *auto*

lemma *nat-size* [*simp*, *code*]: *size* (*n*::*nat*) = *n*
by (*induct n*) *simp-all*

declare *prod.size* [*noatp*]

lemma [*code*]:
size (*P* :: *'a Predicate.pred*) = *0* **by** (*cases P*) *simp*

lemma [*code*]:
pred-size f P = *0* **by** (*cases P*) *simp*

end

22 FunDef: Function Definitions and Termination Proofs

theory *FunDef*
imports *Wellfounded*
uses
Tools/prop-logic.ML
Tools/sat-solver.ML
(*Tools/function-package/fundef-lib.ML*)
(*Tools/function-package/fundef-common.ML*)
(*Tools/function-package/inductive-wrap.ML*)
(*Tools/function-package/context-tree.ML*)
(*Tools/function-package/fundef-core.ML*)
(*Tools/function-package/sum-tree.ML*)
(*Tools/function-package/mutual.ML*)
(*Tools/function-package/pattern-split.ML*)
(*Tools/function-package/fundef-package.ML*)
(*Tools/function-package/auto-term.ML*)
(*Tools/function-package/measure-functions.ML*)
(*Tools/function-package/lexicographic-order.ML*)
(*Tools/function-package/fundef-datatype.ML*)
(*Tools/function-package/induction-scheme.ML*)
(*Tools/function-package/termination.ML*)
(*Tools/function-package/decompose.ML*)
(*Tools/function-package/descent.ML*)
(*Tools/function-package/scnp-solve.ML*)
(*Tools/function-package/scnp-reconstruct.ML*)
begin

22.1 Definitions with default value.

definition
THE-default :: $'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
THE-default d P = (*if* $(\exists!x.\ P\ x)$ *then* $(THE\ x.\ P\ x)$ *else d*)

lemma *THE-defaultI'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
by (*simp add*: *theI' THE-default-def*)

lemma *THE-default1-equality*:
$[\![\exists!x.\ P\ x;\ P\ a]\!] \Longrightarrow THE\text{-}default\ d\ P = a$
by (*simp add*: *the1-equality THE-default-def*)

lemma *THE-default-none*:
$\neg(\exists!x.\ P\ x) \Longrightarrow THE\text{-}default\ d\ P = d$
by (*simp add*:*THE-default-def*)

lemma *fundef-ex1-existence*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $G\ x\ (f\ x)$
apply (*simp only*: *f-def*)
apply (*rule THE-defaultI'*)
apply (*rule ex1*)
done

lemma *fundef-ex1-uniqueness*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
assumes *elm*: $G\ x\ (h\ x)$
shows $h\ x = f\ x$
apply (*simp only*: *f-def*)
apply (*rule THE-default1-equality* [*symmetric*])
apply (*rule ex1*)
apply (*rule elm*)
done

lemma *fundef-ex1-iff*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $(G\ x\ y) = (f\ x = y)$
apply (*auto simp*:*ex1 f-def THE-default1-equality*)
apply (*rule THE-defaultI'*)
apply (*rule ex1*)
done

lemma *fundef-default-value*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *graph*: $\bigwedge x\ y.\ G\ x\ y \Longrightarrow D\ x$

assumes $\neg$ *D x*
shows *f x* = *d x*
proof −
have $\neg(\exists\, y.\ G\ x\ y)$
proof
assume $\exists\, y.\ G\ x\ y$
hence *D x* **using** *graph* **..**
with ‹$\neg$ *D x*› **show** *False* **..**
qed
hence $\neg(\exists!y.\ G\ x\ y)$ **by** *blast*

thus *?thesis*
unfolding *f-def*
by (*rule THE-default-none*)
qed

definition *in-rel-def*[*simp*]:
in-rel R x y == $(x,\ y) \in R$

lemma *wf-in-rel*:
wf R $\Longrightarrow$ *wfP* (*in-rel R*)
by (*simp add*: *wfP-def*)

use *Tools/function-package/fundef-lib.ML*
use *Tools/function-package/fundef-common.ML*
use *Tools/function-package/inductive-wrap.ML*
use *Tools/function-package/context-tree.ML*
use *Tools/function-package/fundef-core.ML*
use *Tools/function-package/sum-tree.ML*
use *Tools/function-package/mutual.ML*
use *Tools/function-package/pattern-split.ML*
use *Tools/function-package/auto-term.ML*
use *Tools/function-package/fundef-package.ML*
use *Tools/function-package/fundef-datatype.ML*
use *Tools/function-package/induction-scheme.ML*

setup ⟪
FundefPackage.setup
#> *FundefDatatype.setup*
#> *InductionScheme.setup*
⟫

22.2 Measure Functions

inductive *is-measure* :: ('*a* ⇒ *nat*) ⇒ *bool*
where *is-measure-trivial*: *is-measure f*

use *Tools/function-package/measure-functions.ML*
setup *MeasureFunctions.setup*

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

lemma *measure-fst*[*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* ($\lambda p.$ *f* (*fst p*))
by (*rule is-measure-trivial*)
lemma *measure-snd*[*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* ($\lambda p.$ *f* (*snd p*))
by (*rule is-measure-trivial*)

use *Tools/function-package/lexicographic-order.ML*
setup *LexicographicOrder.setup*

22.3 Congruence Rules

lemma *let-cong* [*fundef-cong*]:
M = *N* $\Longrightarrow$ ($\bigwedge x.$ *x* = *N* $\Longrightarrow$ *f x* = *g x*) $\Longrightarrow$ *Let M f* = *Let N g*
unfolding *Let-def* **by** *blast*

lemmas [*fundef-cong*] =
if-cong image-cong INT-cong UN-cong
bex-cong ball-cong imp-cong

lemma *split-cong* [*fundef-cong*]:
($\bigwedge x\ y.$ (*x*, *y*) = *q* $\Longrightarrow$ *f x y* = *g x y*) $\Longrightarrow$ *p* = *q*
$\Longrightarrow$ *split f p* = *split g q*
by (*auto simp*: *split-def*)

lemma *comp-cong* [*fundef-cong*]:
f (*g x*) = *f'* (*g' x'*) $\Longrightarrow$ (*f o g*) *x* = (*f' o g'*) *x'*
unfolding *o-apply* **.**

22.4 Simp rules for termination proofs

lemma *termination-basic-simps*[*termination-simp*]:
x < (*y*::*nat*) $\Longrightarrow$ *x* < *y* + *z*
x < *z* $\Longrightarrow$ *x* < *y* + *z*
x ≤ *y* $\Longrightarrow$ *x* ≤ *y* + (*z*::*nat*)
x ≤ *z* $\Longrightarrow$ *x* ≤ *y* + (*z*::*nat*)
x < *y* $\Longrightarrow$ *x* ≤ (*y*::*nat*)
by *arith*+

declare *le-imp-less-Suc*[*termination-simp*]

lemma *prod-size-simp*[*termination-simp*]:
prod-size f g p = *f* (*fst p*) + *g* (*snd p*) + *Suc 0*
by (*induct p*) *auto*

22.5 Decomposition

lemma *less-by-empty*:
  $A = \{\} \Longrightarrow A \subseteq B$
and *union-comp-emptyL*:
  $[\![A\ O\ C = \{\};\ B\ O\ C = \{\}]\!] \Longrightarrow (A \cup B)\ O\ C = \{\}$
and *union-comp-emptyR*:
  $[\![A\ O\ B = \{\};\ A\ O\ C = \{\}]\!] \Longrightarrow A\ O\ (B \cup C) = \{\}$
and *wf-no-loop*:
  $R\ O\ R = \{\} \Longrightarrow wf\ R$
by (*auto simp add*: *wf-comp-self*[*of R*])

22.6 Reduction Pairs

definition
  reduction-pair $P = (wf\ (fst\ P) \wedge snd\ P\ O\ fst\ P \subseteq fst\ P)$

lemma *reduction-pairI*[*intro*]: $wf\ R \Longrightarrow S\ O\ R \subseteq R \Longrightarrow$ *reduction-pair* (R, S)
unfolding *reduction-pair-def* **by** *auto*

lemma *reduction-pair-lemma*:
  assumes *rp*: *reduction-pair P*
  assumes $R \subseteq fst\ P$
  assumes $S \subseteq snd\ P$
  assumes *wf S*
  shows $wf\ (R \cup S)$
proof −
  from *rp* ‹$S \subseteq snd\ P$› **have** $wf\ (fst\ P)\ S\ O\ fst\ P \subseteq fst\ P$
    unfolding *reduction-pair-def* **by** *auto*
  with ‹*wf S*› **have** $wf\ (fst\ P \cup S)$
    by (*auto intro*: *wf-union-compatible*)
  moreover from ‹$R \subseteq fst\ P$› **have** $R \cup S \subseteq fst\ P \cup S$ **by** *auto*
  ultimately show *?thesis* **by** (*rule wf-subset*)
qed

definition
  rp-inv-image $= (\lambda(R,S)\ f.\ (inv\text{-}image\ R\ f,\ inv\text{-}image\ S\ f))$

lemma *rp-inv-image-rp*:
  reduction-pair $P \Longrightarrow$ *reduction-pair* (*rp-inv-image P f*)
  unfolding *reduction-pair-def rp-inv-image-def split-def*
  by *force*

22.7 Concrete orders for SCNP termination proofs

definition *pair-less* $=$ *less-than* $<*lex*>$ *less-than*
definition [*code del*]: *pair-leq* $=$ *pair-less*$^=$
definition *max-strict* $=$ *max-ext pair-less*
definition [*code del*]: *max-weak* $=$ *max-ext pair-leq* $\cup$ $\{(\{\}, \{\})\}$
definition [*code del*]: *min-strict* $=$ *min-ext pair-less*

definition [*code del*]: *min-weak* = *min-ext pair-leq* ∪ {({}, {})}

lemma *wf-pair-less*[*simp*]: *wf pair-less*
by (*auto simp*: *pair-less-def*)

Introduction rules for *pair-less*/*pair-leq*

lemma *pair-leqI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in$ *pair-leq*
and *pair-leqI2*: $a \leq b \Longrightarrow s \leq t \Longrightarrow ((a, s), (b, t)) \in$ *pair-leq*
and *pair-lessI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in$ *pair-less*
and *pair-lessI2*: $a \leq b \Longrightarrow s < t \Longrightarrow ((a, s), (b, t)) \in$ *pair-less*
unfolding *pair-leq-def pair-less-def* **by** *auto*

Introduction rules for max

lemma *smax-emptyI*:
finite $Y \Longrightarrow Y \neq \{\} \Longrightarrow (\{\}, Y) \in$ *max-strict*
and *smax-insertI*:
$[\![y \in Y; (x, y) \in$ *pair-less*; $(X, Y) \in$ *max-strict*$]\!] \Longrightarrow$ (*insert x X*, Y) ∈ *max-strict*
and *wmax-emptyI*:
finite $X \Longrightarrow (\{\}, X) \in$ *max-weak*
and *wmax-insertI*:
$[\![y \in YS; (x, y) \in$ *pair-leq*; $(XS, YS) \in$ *max-weak*$]\!] \Longrightarrow$ (*insert x XS*, YS) ∈ *max-weak*
unfolding *max-strict-def max-weak-def* **by** (*auto elim*!: *max-ext.cases*)

Introduction rules for min

lemma *smin-emptyI*:
$X \neq \{\} \Longrightarrow (X, \{\}) \in$ *min-strict*
and *smin-insertI*:
$[\![x \in XS; (x, y) \in$ *pair-less*; $(XS, YS) \in$ *min-strict*$]\!] \Longrightarrow (XS,$ *insert y* $YS) \in$ *min-strict*
and *wmin-emptyI*:
$(X, \{\}) \in$ *min-weak*
and *wmin-insertI*:
$[\![x \in XS; (x, y) \in$ *pair-leq*; $(XS, YS) \in$ *min-weak*$]\!] \Longrightarrow (XS,$ *insert y* $YS) \in$ *min-weak*
by (*auto simp*: *min-strict-def min-weak-def min-ext-def*)

Reduction Pairs

lemma *max-ext-compat*:
assumes *S O R* ⊆ *R*
shows (*max-ext S* ∪ {({},{})}) *O max-ext R* ⊆ *max-ext R*
using *assms*
apply *auto*
apply (*elim max-ext.cases*)
apply *rule*
apply *auto*[*3*]
apply (*drule-tac x*=*xa* **in** *meta-spec*)
apply *simp*

apply (*erule bexE*)
apply (*drule-tac x=xb* **in** *meta-spec*)
by *auto*

lemma *max-rpair-set*: *reduction-pair* (*max-strict*, *max-weak*)
  unfolding *max-strict-def max-weak-def*
apply (*intro reduction-pairI max-ext-wf*)
apply *simp*
apply (*rule max-ext-compat*)
by (*auto simp*: *pair-less-def pair-leq-def*)

lemma *min-ext-compat*:
  assumes *S O R* ⊆ *R*
  shows (*min-ext S* ∪ {({},{})}) *O min-ext R* ⊆ *min-ext R*
using *assms*
apply (*auto simp*: *min-ext-def*)
apply (*drule-tac x=ya* **in** *bspec*, *assumption*)
apply (*erule bexE*)
apply (*drule-tac x=xc* **in** *bspec*)
apply *assumption*
by *auto*

lemma *min-rpair-set*: *reduction-pair* (*min-strict*, *min-weak*)
  unfolding *min-strict-def min-weak-def*
apply (*intro reduction-pairI min-ext-wf*)
apply *simp*
apply (*rule min-ext-compat*)
by (*auto simp*: *pair-less-def pair-leq-def*)

22.8 Tool setup

use *Tools/function-package/termination.ML*
use *Tools/function-package/decompose.ML*
use *Tools/function-package/descent.ML*
use *Tools/function-package/scnp-solve.ML*
use *Tools/function-package/scnp-reconstruct.ML*

setup ⟪ *ScnpReconstruct.setup* ⟫

ML-val — setup inactive
⟪
 Context.theory-map (*FundefCommon.set-termination-prover* (*ScnpReconstruct.decomp-scnp*

 [*ScnpSolve.MAX*, *ScnpSolve.MIN*, *ScnpSolve.MS*]))
⟫

end

23 Record: Extensible records with structural subtyping

theory *Record*
imports *Product-Type*
uses (*Tools/record-package.ML*)
begin

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
by *simp*

lemma *rec-UNIV-I*: $\bigwedge x.\ x{\in}UNIV \equiv True$
by *simp*

lemma *rec-True-simp*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
by *simp*

lemma *K-record-comp*: $(\lambda x.\ c) \circ f = (\lambda x.\ c)$
by (*simp add*: *comp-def*)

23.1 Concrete record syntax

nonterminals
ident field-type field-types field fields update updates
syntax
-constify :: *id* => *ident* (-)
-constify :: *longid* => *ident* (-)

-field-type :: [*ident*, *type*] => *field-type* ((*2*- ::/ -))
:: *field-type* => *field-types* (-)
-field-types :: [*field-type*, *field-types*] => *field-types* (-,/ -)
-record-type :: *field-types* => *type* ((*3'*(| - |')))
-record-type-scheme :: [*field-types*, *type*] => *type* ((*3'*(| -,/ (*2*... ::/ -) |')))

-field :: [*ident*, *'a*] => *field* ((*2*- =/ -))
:: *field* => *fields* (-)
-fields :: [*field*, *fields*] => *fields* (-,/ -)
-record :: *fields* => *'a* ((*3'*(| - |')))
-record-scheme :: [*fields*, *'a*] => *'a* ((*3'*(| -,/ (*2*... =/ -) |')))

-update-name :: *idt*
-update :: [*ident*, *'a*] => *update* ((*2*- :=/ -))
:: *update* => *updates* (-)
-updates :: [*update*, *updates*] => *updates* (-,/ -)
-record-update :: [*'a*, *updates*] => *'b* (-/(*3'*(| - |')) [*900*,*0*] *900*)

syntax (*xsymbols*)
-record-type :: *field-types* => *type* ((*3*(|-|)))
-record-type-scheme :: [*field-types*, *type*] => *type* ((*3*(|-,/ (*2*... ::/ -)|)))

```
  -record             :: fields => 'a                        ((3(|-|)))
  -record-scheme      :: [fields, 'a] => 'a                  ((3(|-,/ (2... =/ -)|)))
  -record-update      :: ['a, updates] => 'b                 (-/(3(|-|)) [900,0] 900)
```

use *Tools/record-package.ML*
setup *RecordPackage.setup*

end

24 Option: Datatype option

theory *Option*
imports *Datatype Finite-Set*
begin

datatype *'a option = None | Some 'a*

lemma *not-None-eq* [*iff*]: *(x ~= None) = (EX y. x = Some y)*
by *(induct x) auto*

lemma *not-Some-eq* [*iff*]: *(ALL y. x ~= Some y) = (x = None)*
by *(induct x) auto*

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

lemma *option-caseE*:
assumes *c*: *(case x of None => P | Some y => Q y)*
obtains
(None) x = None **and** *P*
| *(Some) y* **where** *x = Some y* **and** *Q y*
using *c* **by** *(cases x) simp-all*

lemma *insert-None-conv-UNIV*: *insert None (range Some) = UNIV*
by *(rule set-ext, case-tac x) auto*

instance *option* :: (*finite*) *finite* **proof**
qed *(simp add*: *insert-None-conv-UNIV* [*symmetric*])

lemma *inj-Some* [*simp*]: *inj-on Some A*
by *(rule inj-onI) simp*

24.0.1 Operations

primrec *the* :: *'a option => 'a* **where**
the (Some x) = x

primrec *set* :: *'a option => 'a set* **where**

set None = {} |
set (*Some x*) = {*x*}

lemma *ospec* [*dest*]: (*ALL x*:*set A*. *P x*) ==> *A* = *Some x* ==> *P x*
by *simp*

declaration ⟪ *fn* - =>
Classical.map-cs (*fn cs* => *cs addSD2* (*ospec*, *thm ospec*))
⟫

lemma *elem-set* [*iff*]: (*x* : *set xo*) = (*xo* = *Some x*)
by (*cases xo*) *auto*

lemma *set-empty-eq* [*simp*]: (*set xo* = {}) = (*xo* = *None*)
by (*cases xo*) *auto*

definition
map :: ('*a* ⇒ '*b*) ⇒ '*a option* ⇒ '*b option*
where
[*code del*]: *map* = (%*f y*. *case y of None* => *None* | *Some x* => *Some* (*f x*))

lemma *option-map-None* [*simp*, *code*]: *map f None* = *None*
by (*simp add*: *map-def*)

lemma *option-map-Some* [*simp*, *code*]: *map f* (*Some x*) = *Some* (*f x*)
by (*simp add*: *map-def*)

lemma *option-map-is-None* [*iff*]:
(*map f opt* = *None*) = (*opt* = *None*)
by (*simp add*: *map-def split add*: *option.split*)

lemma *option-map-eq-Some* [*iff*]:
(*map f xo* = *Some y*) = (*EX z*. *xo* = *Some z* & *f z* = *y*)
by (*simp add*: *map-def split add*: *option.split*)

lemma *option-map-comp*:
map f (*map g opt*) = *map* (*f o g*) *opt*
by (*simp add*: *map-def split add*: *option.split*)

lemma *option-map-o-sum-case* [*simp*]:
map f o sum-case g h = *sum-case* (*map f o g*) (*map f o h*)
by (*rule ext*) (*simp split*: *sum.split*)

hide (**open**) *const set map*

24.0.2 Code generator setup

definition

is-none :: *'a option* ⇒ *bool* **where**
is-none-none [*code post*, *symmetric*, *code inline*]: *is-none x* ⟷ *x* = *None*

lemma *is-none-code* [*code*]:
shows *is-none None* ⟷ *True*
and *is-none* (*Some x*) ⟷ *False*
unfolding *is-none-none* [*symmetric*] **by** *simp-all*

hide (**open**) *const is-none*

code-type *option*
(*SML* - *option*)
(*OCaml* - *option*)
(*Haskell Maybe* -)

code-const *None* **and** *Some*
(*SML NONE* **and** *SOME*)
(*OCaml None* **and** *Some* -)
(*Haskell Nothing* **and** *Just*)

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

code-const *eq-class.eq* :: *'a*::*eq option* ⇒ *'a option* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-reserved *SML*
option NONE SOME

code-reserved *OCaml*
option None Some

end

25 Extraction: Program extraction for HOL

theory *Extraction*
imports *Option*
uses *Tools/rewrite-hol-proof.ML*
begin

25.1 Setup

setup ⟪
let
fun realizes-set-proc (*Const* (*realizes*, *Type* (*fun*, [*Type* (*Null*, []), -])) $ *r* $
(*Const* (*op* :, -) $ *x* $ *S*)) = (*case strip-comb S of*
(*Var* (*ixn*, *U*), *ts*) => *SOME* (*list-comb* (*Var* (*ixn*, *U*), *ts* @ [*x*]))

```
    | (Free (s, U), ts) => SOME (list-comb (Free (s, U), ts @ [x]))
    | - => NONE)
  | realizes-set-proc (Const (realizes, Type (fun, [T, -])) $ r $
    (Const (op :, -) $ x $ S)) = (case strip-comb S of
      (Var (ixn, U), ts) => SOME (list-comb (Var (ixn, T --> U), r :: ts @
[x]))
    | (Free (s, U), ts) => SOME (list-comb (Free (s, T --> U), r :: ts @ [x]))
    | - => NONE)
  | realizes-set-proc - = NONE;

in
  Extraction.add-types
    [(bool, ([], NONE))] #>
  Extraction.set-preprocessor (fn thy =>
    Proofterm.rewrite-proof-notypes
      ([], RewriteHOLProof.elim-cong :: ProofRewriteRules.rprocs true) o
    Proofterm.rewrite-proof thy
      (RewriteHOLProof.rews, ProofRewriteRules.rprocs true) o
    ProofRewriteRules.elim-vars (curry Const @{const-name default}))
end
⟩⟩

lemmas [extraction-expand] =
  meta-spec atomize-eq atomize-all atomize-imp atomize-conj
  allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
  notE' impE' impE iffE imp-cong simp-thms eq-True eq-False
  induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def
  induct-atomize induct-rulify induct-rulify-fallback
  True-implies-equals TrueE

datatype sumbool = Left | Right
```

25.2 Type of extracted program

extract-type

```
  typeof (Trueprop P) ≡ typeof P

  typeof P ≡ Type (TYPE(Null)) ⟹ typeof Q ≡ Type (TYPE('Q)) ⟹
    typeof (P ⟶ Q) ≡ Type (TYPE('Q))

  typeof Q ≡ Type (TYPE(Null)) ⟹ typeof (P ⟶ Q) ≡ Type (TYPE(Null))

  typeof P ≡ Type (TYPE('P)) ⟹ typeof Q ≡ Type (TYPE('Q)) ⟹
    typeof (P ⟶ Q) ≡ Type (TYPE('P ⇒ 'Q))

  (λx. typeof (P x)) ≡ (λx. Type (TYPE(Null))) ⟹
    typeof (∀ x. P x) ≡ Type (TYPE(Null))
```

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$typeof\ (\forall\, x::'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$typeof\ (\exists\, x::'a.\ P\ x) \equiv Type\ (TYPE('a))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$typeof\ (\exists\, x::'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

25.3 Realizability

realizability
$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall\, x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall\, x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg P) \equiv (\forall x{::}'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P{::}bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

25.4 Computational content of basic inference rules

theorem *disjE-realizer*:
assumes *r*: $case\ x\ of\ Inl\ p \Rightarrow P\ p \mid Inr\ q \Rightarrow Q\ q$

and *r1*: $\bigwedge p.\ P\ p \Longrightarrow R\ (f\ p)$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows R (*case x of Inl p ⇒ f p | Inr q ⇒ g q*)
proof (*cases x*)
case *Inl*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Inr*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *disjE-realizer2*:
assumes *r*: *case x of None ⇒ P | Some q ⇒ Q q*
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows R (*case x of None ⇒ f | Some q ⇒ g q*)
proof (*cases x*)
case *None*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Some*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *disjE-realizer3*:
assumes *r*: *case x of Left ⇒ P | Right ⇒ Q*
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $Q \Longrightarrow R\ g$
shows R (*case x of Left ⇒ f | Right ⇒ g*)
proof (*cases x*)
case *Left*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Right*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *conjI-realizer*:
$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (fst\ (p,\ q)) \wedge Q\ (snd\ (p,\ q))$
by *simp*

theorem *exI-realizer*:
$P\ y\ x \Longrightarrow P\ (snd\ (x,\ y))\ (fst\ (x,\ y))$ **by** *simp*

theorem *exE-realizer*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q\ (f\ x\ y)) \Longrightarrow Q\ (let\ (x,\ y) = p\ in\ f\ x\ y)$
by (*cases p*) (*simp add*: *Let-def*)

theorem *exE-realizer′*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q$ **by** (*cases p*) *simp*

setup ⟪

Sign.add-const-constraint (@{*const-name default*}, *SOME* @{*typ 'a*::*type*})
⟫

realizers
impI (*P*, *Q*): λ*pq*. *pq*
Λ *P Q pq* (*h*: -). *allI* · - • (Λ *x*. *impI* · - - - • (*h* · *x*))

impI (*P*): *Null*
Λ *P Q* (*h*: -). *allI* · - • (Λ *x*. *impI* · - - - • (*h* · *x*))

impI (*Q*): λ*q*. *q* Λ *P Q q*. *impI* · - - -

impI: *Null impI*

mp (*P*, *Q*): λ*pq*. *pq*
Λ *P Q pq* (*h*: -) *p*. *mp* · - - - • (*spec* · - · *p* • *h*)

mp (*P*): *Null*
Λ *P Q* (*h*: -) *p*. *mp* · - - - • (*spec* · - · *p* • *h*)

mp (*Q*): λ*q*. *q* Λ *P Q q*. *mp* · - - -

mp: *Null mp*

allI (*P*): λ*p*. *p* Λ *P p*. *allI* · -

allI: *Null allI*

spec (*P*): λ*x p*. *p x* Λ *P x p*. *spec* · - · *x*

spec: *Null spec*

exI (*P*): λ*x p*. (*x*, *p*) Λ *P x p*. *exI-realizer* · *P* · *p* · *x*

exI: λ*x*. *x* Λ *P x* (*h*: -). *h*

exE (*P*, *Q*): λ*p pq*. *let* (*x*, *y*) = *p in pq x y*
Λ *P Q p* (*h*: -) *pq*. *exE-realizer* · *P* · *p* · *Q* · *pq* • *h*

exE (*P*): *Null*
Λ *P Q p*. *exE-realizer'* · - - - - -

exE (*Q*): λ*x pq*. *pq x*
Λ *P Q x* (*h1*: -) *pq* (*h2*: -). *h2* · *x* • *h1*

exE: *Null*
Λ *P Q x* (*h1*: -) (*h2*: -). *h2* · *x* • *h1*

conjI (*P*, *Q*): *Pair*

$\Lambda\ P\ Q\ p\ (h\colon \text{-})\ q.\ \mathit{conjI\text{-}realizer} \cdot P \cdot p \cdot Q \cdot q \bullet h$

conjI (P): $\lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjI} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjI (Q): $\lambda q.\ q$
$\Lambda\ P\ Q\ (h\colon \text{-})\ q.\ \mathit{conjI} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet h$

conjI: *Null conjI*

conjunct1 $(P,\ Q)$: *fst*
$\Lambda\ P\ Q\ pq.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct1 (P): $\lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct1 (Q): *Null*
$\Lambda\ P\ Q\ q.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct1: *Null conjunct1*

conjunct2 $(P,\ Q)$: *snd*
$\Lambda\ P\ Q\ pq.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct2 (P): *Null*
$\Lambda\ P\ Q\ p.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct2 (Q): $\lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct2: *Null conjunct2*

disjI1 $(P,\ Q)$: *Inl*
$\Lambda\ P\ Q\ p.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{sum.cases\text{-}1} \cdot P \cdot \text{-} \cdot \text{-} \cdot p)$

disjI1 (P): *Some*
$\Lambda\ P\ Q\ p.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{option.cases\text{-}2} \cdot \text{-} \cdot \text{-} \cdot P \cdot p)$

disjI1 (Q): *None*
$\Lambda\ P\ Q.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{option.cases\text{-}1} \cdot \text{-} \cdot \text{-} \cdot \text{-})$

disjI1: *Left*
$\Lambda\ P\ Q.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{sumbool.cases\text{-}1} \cdot \text{-} \cdot \text{-} \cdot \text{-})$

disjI2 $(P,\ Q)$: *Inr*
$\Lambda\ Q\ P\ q.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{sum.cases\text{-}2} \cdot \text{-} \cdot \text{-} \cdot Q \cdot q)$

disjI2 (P): *None*
$\Lambda\ Q\ P.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{option.cases\text{-}1} \cdot \text{-} \cdot \text{-} \cdot \text{-})$

disjI2 (*Q*): *Some*
  Λ *Q P q*. *iffD2* · - · - · (*option.cases-2* · - · *Q* · *q*)

disjI2: *Right*
  Λ *Q P*. *iffD2* · - · - · (*sumbool.cases-2* · - · -)

disjE (*P*, *Q*, *R*): λ*pq pr qr*.
  (*case pq of Inl p* ⇒ *pr p* | *Inr q* ⇒ *qr q*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*Q*, *R*): λ*pq pr qr*.
  (*case pq of None* ⇒ *pr* | *Some q* ⇒ *qr q*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer2 · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*P*, *R*): λ*pq pr qr*.
  (*case pq of None* ⇒ *qr* | *Some p* ⇒ *pr p*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr* (*h3*: -).
    disjE-realizer2 · - · - · *pq* · *R* · *qr* · *pr* · *h1* · *h3* · *h2*

disjE (*R*): λ*pq pr qr*.
  (*case pq of Left* ⇒ *pr* | *Right* ⇒ *qr*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer3 · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*P*, *Q*): *Null*
  Λ *P Q R pq*. *disjE-realizer* · - · - · *pq* · (λ*x*. *R*) · - · -

disjE (*Q*): *Null*
  Λ *P Q R pq*. *disjE-realizer2* · - · - · *pq* · (λ*x*. *R*) · - · -

disjE (*P*): *Null*
  Λ *P Q R pq* (*h1*: -) (*h2*: -) (*h3*: -).
    disjE-realizer2 · - · - · *pq* · (λ*x*. *R*) · - · - · *h1* · *h3* · *h2*

disjE: *Null*
  Λ *P Q R pq*. *disjE-realizer3* · - · - · *pq* · (λ*x*. *R*) · - · -

FalseE (*P*): *default*
  Λ *P*. *FalseE* · -

FalseE: *Null FalseE*

notI (*P*): *Null*
  Λ *P* (*h*: -). *allI* · - · (Λ *x*. *notI* · - · (*h* · *x*))

notI: *Null notI*

notE (*P*, *R*): $\lambda p.$ *default*
  Λ *P R* (*h*: -) *p*. *notE* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *p* $\cdot$ *h*)

notE (*P*): *Null*
  Λ *P R* (*h*: -) *p*. *notE* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *p* $\cdot$ *h*)

notE (*R*): *default*
  Λ *P R*. *notE* $\cdot$ - $\cdot$ - $\cdot$ -

notE: *Null notE*

subst (*P*): $\lambda s\ t\ ps.$ *ps*
  Λ *s t P* (*h*: -) *ps*. *subst* $\cdot$ *s* $\cdot$ *t* $\cdot$ *P ps* $\cdot$ *h*

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
  Λ *Q P pq* (*h*: -) *p*.
    mp $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *p* $\cdot$ (*conjunct1* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*))

iffD1 (*P*): $\lambda p.$ *p*
  Λ *Q P p* (*h*: -). *mp* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*conjunct1* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*)

iffD1 (*Q*): *Null*
  Λ *Q P q1* (*h*: -) *q2*.
    mp $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *q2* $\cdot$ (*conjunct1* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*))

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
  Λ *P Q pq* (*h*: -) *q*.
    mp $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *q* $\cdot$ (*conjunct2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*))

iffD2 (*P*): $\lambda p.$ *p*
  Λ *P Q p* (*h*: -). *mp* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*conjunct2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*)

iffD2 (*Q*): *Null*
  Λ *P Q q1* (*h*: -) *q2*.
    mp $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *q2* $\cdot$ (*conjunct2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*))

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
  Λ *P Q pq* (*h1* : -) *qp* (*h2* : -). *conjI-realizer* $\cdot$
    ($\lambda pq.\ \forall x.\ P\ x \longrightarrow Q\ (pq\ x)$) $\cdot$ *pq* $\cdot$
    ($\lambda qp.\ \forall x.\ Q\ x \longrightarrow P\ (qp\ x)$) $\cdot$ *qp* $\cdot$
    (*allI* $\cdot$ - $\cdot$ (Λ *x*. *impI* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*h1* $\cdot$ *x*))) $\cdot$
    (*allI* $\cdot$ - $\cdot$ (Λ *x*. *impI* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*h2* $\cdot$ *x*)))

iffI (*P*): $\lambda p.\ p$
$\Lambda\ P\ Q\ (h1 : \text{-})\ p\ (h2 : \text{-}).\ conjI \cdot \text{-} \cdot \text{-} \cdot$
$(allI \cdot \text{-} \cdot (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot (h1 \cdot x))) \cdot$
$(impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot h2)$

iffI (*Q*): $\lambda q.\ q$
$\Lambda\ P\ Q\ q\ (h1 : \text{-})\ (h2 : \text{-}).\ conjI \cdot \text{-} \cdot \text{-} \cdot$
$(impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot h1) \cdot$
$(allI \cdot \text{-} \cdot (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot (h2 \cdot x)))$

iffI: *Null iffI*

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

end

26 Divides: The division operators div and mod

theory *Divides*
imports *Nat Power Product-Type*
uses ~~/*src/Provers/Arith/cancel-div-mod.ML*
begin

26.1 Syntactic division operations

class *div* = *dvd* +
fixes *div* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *div* 70)
and *mod* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *mod* 70)

26.2 Abstract division in commutative semirings.

class *semiring-div* = *comm-semiring-1-cancel* + *div* +
assumes *mod-div-equality*: $a\ div\ b * b + a\ mod\ b = a$
and *div-by-0* [*simp*]: $a\ div\ 0 = 0$
and *div-0* [*simp*]: $0\ div\ a = 0$
and *div-mult-self1* [*simp*]: $b \neq 0 \Longrightarrow (a + c * b)\ div\ b = c + a\ div\ b$
begin

op div and *op mod*

lemma *mod-div-equality2*: $b * (a\ div\ b) + a\ mod\ b = a$
unfolding *mult-commute* [*of b*]
by (*rule mod-div-equality*)

lemma *mod-div-equality'*: $a\ mod\ b + a\ div\ b * b = a$
using *mod-div-equality* [*of a b*]
by (*simp only*: *add-ac*)

lemma *div-mod-equality*: $((a\ div\ b) * b + a\ mod\ b) + c = a + c$
by (*simp add*: *mod-div-equality*)

lemma *div-mod-equality2*: $(b * (a\ div\ b) + a\ mod\ b) + c = a + c$
by (*simp add*: *mod-div-equality2*)

lemma *mod-by-0* [*simp*]: $a\ mod\ 0 = a$
using *mod-div-equality* [*of a zero*] **by** *simp*

lemma *mod-0* [*simp*]: $0\ mod\ a = 0$
using *mod-div-equality* [*of zero a*] *div-0* **by** *simp*

lemma *div-mult-self2* [*simp*]:
assumes $b \neq 0$
shows $(a + b * c)\ div\ b = c + a\ div\ b$
using *assms div-mult-self1* [*of b a c*] **by** (*simp add*: *mult-commute*)

lemma *mod-mult-self1* [*simp*]: $(a + c * b)\ mod\ b = a\ mod\ b$
proof (*cases* $b = 0$)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
have $a + c * b = (a + c * b)\ div\ b * b + (a + c * b)\ mod\ b$
by (*simp add*: *mod-div-equality*)
also from *False div-mult-self1* [*of b a c*] **have**
$\ldots = (c + a\ div\ b) * b + (a + c * b)\ mod\ b$
by (*simp add*: *algebra-simps*)
finally have $a = a\ div\ b * b + (a + c * b)\ mod\ b$
by (*simp add*: *add-commute* [*of a*] *add-assoc left-distrib*)
then have $a\ div\ b * b + (a + c * b)\ mod\ b = a\ div\ b * b + a\ mod\ b$
by (*simp add*: *mod-div-equality*)
then show *?thesis* **by** *simp*
qed

lemma *mod-mult-self2* [*simp*]: $(a + b * c)\ mod\ b = a\ mod\ b$
by (*simp add*: *mult-commute* [*of b*])

lemma *div-mult-self1-is-id* [*simp*]: $b \neq 0 \Longrightarrow b * a\ div\ b = a$
using *div-mult-self2* [*of b 0 a*] **by** *simp*

lemma *div-mult-self2-is-id* [*simp*]: $b \neq 0 \Longrightarrow a * b\ div\ b = a$
using *div-mult-self1* [*of b 0 a*] **by** *simp*

lemma *mod-mult-self1-is-0* [*simp*]: $b * a\ mod\ b = 0$
using *mod-mult-self2* [*of 0 b a*] **by** *simp*

lemma *mod-mult-self2-is-0* [*simp*]: $a * b \bmod b = 0$
using *mod-mult-self1* [*of 0 a b*] **by** *simp*

lemma *div-by-1* [*simp*]: $a \operatorname{div} 1 = a$
using *div-mult-self2-is-id* [*of 1 a*] *zero-neq-one* **by** *simp*

lemma *mod-by-1* [*simp*]: $a \bmod 1 = 0$
proof −
from *mod-div-equality* [*of a one*] *div-by-1* **have** $a + a \bmod 1 = a$ **by** *simp*
then have $a + a \bmod 1 = a + 0$ **by** *simp*
then show *?thesis* **by** (*rule add-left-imp-eq*)
qed

lemma *mod-self* [*simp*]: $a \bmod a = 0$
using *mod-mult-self2-is-0* [*of 1*] **by** *simp*

lemma *div-self* [*simp*]: $a \neq 0 \Longrightarrow a \operatorname{div} a = 1$
using *div-mult-self2-is-id* [*of - 1*] **by** *simp*

lemma *div-add-self1* [*simp*]:
assumes $b \neq 0$
shows $(b + a) \operatorname{div} b = a \operatorname{div} b + 1$
using *assms div-mult-self1* [*of b a 1*] **by** (*simp add*: *add-commute*)

lemma *div-add-self2* [*simp*]:
assumes $b \neq 0$
shows $(a + b) \operatorname{div} b = a \operatorname{div} b + 1$
using *assms div-add-self1* [*of b a*] **by** (*simp add*: *add-commute*)

lemma *mod-add-self1* [*simp*]:
$(b + a) \bmod b = a \bmod b$
using *mod-mult-self1* [*of a 1 b*] **by** (*simp add*: *add-commute*)

lemma *mod-add-self2* [*simp*]:
$(a + b) \bmod b = a \bmod b$
using *mod-mult-self1* [*of a 1 b*] **by** *simp*

lemma *mod-div-decomp*:
fixes *a b*
obtains *q r* **where** $q = a \operatorname{div} b$ **and** $r = a \bmod b$
and $a = q * b + r$
proof −
from *mod-div-equality* **have** $a = a \operatorname{div} b * b + a \bmod b$ **by** *simp*
moreover have $a \operatorname{div} b = a \operatorname{div} b$ **..**
moreover have $a \bmod b = a \bmod b$ **..**
note *that* **ultimately show** *thesis* **by** *blast*
qed

lemma *dvd-eq-mod-eq-0* [*code unfold*]: *a dvd b* ⟷ *b mod a = 0*
proof
  assume *b mod a = 0*
  with *mod-div-equality* [*of b a*] **have** *b div a * a = b* **by** *simp*
  then have *b = a * (b div a)* **unfolding** *mult-commute* **..**
  then have ∃ *c*. *b = a * c* **..**
  then show *a dvd b* **unfolding** *dvd-def* **.**
next
  assume *a dvd b*
  then have ∃ *c*. *b = a * c* **unfolding** *dvd-def* **.**
  then obtain *c* **where** *b = a * c* **..**
  then have *b mod a = a * c mod a* **by** *simp*
  then have *b mod a = c * a mod a* **by** (*simp add*: *mult-commute*)
  then show *b mod a = 0* **by** *simp*
qed

lemma *mod-div-trivial* [*simp*]: *a mod b div b = 0*
proof (*cases b = 0*)
  assume *b = 0*
  thus *?thesis* **by** *simp*
next
  assume *b ≠ 0*
  hence *a div b + a mod b div b = (a mod b + a div b * b) div b*
    by (*rule div-mult-self1* [*symmetric*])
  also have *... = a div b*
    by (*simp only*: *mod-div-equality'*)
  also have *... = a div b + 0*
    by *simp*
  finally show *?thesis*
    by (*rule add-left-imp-eq*)
qed

lemma *mod-mod-trivial* [*simp*]: *a mod b mod b = a mod b*
proof −
  have *a mod b mod b = (a mod b + a div b * b) mod b*
    by (*simp only*: *mod-mult-self1*)
  also have *... = a mod b*
    by (*simp only*: *mod-div-equality'*)
  finally show *?thesis* **.**
qed

lemma *dvd-imp-mod-0*: *a dvd b* ⟹ *b mod a = 0*
by (*rule dvd-eq-mod-eq-0*[*THEN iffD1*])

lemma *dvd-div-mult-self*: *a dvd b* ⟹ *(b div a) * a = b*
by (*subst* (*2*) *mod-div-equality* [*of b a*, *symmetric*]) (*simp add*:*dvd-imp-mod-0*)

lemma *dvd-div-mult*: *a dvd b* ⟹ *(b div a) * c = b * c div a*
apply (*cases a = 0*)

apply *simp*
apply (*auto simp*: *dvd-def mult-assoc*)
done

lemma *div-dvd-div*[*simp*]:
  a dvd b $\Longrightarrow$ *a dvd c* $\Longrightarrow$ (*b div a dvd c div a*) = (*b dvd c*)
apply (*cases a* = *0*)
apply *simp*
apply (*unfold dvd-def*)
apply *auto*
apply(*blast intro*:*mult-assoc*[*symmetric*])
apply(*fastsimp simp add*: *mult-assoc*)
done

lemma *dvd-mod-imp-dvd*: [| *k dvd m mod n*; *k dvd n* |] ==> *k dvd m*
  apply (*subgoal-tac k dvd* (*m div n*) ∗*n* + *m mod n*)
  apply (*simp add*: *mod-div-equality*)
  apply (*simp only*: *dvd-add dvd-mult*)
  done

Addition respects modular equivalence.

lemma *mod-add-left-eq*: (*a* + *b*) *mod c* = (*a mod c* + *b*) *mod c*
proof −
  have (*a* + *b*) *mod c* = (*a div c* ∗ *c* + *a mod c* + *b*) *mod c*
    by (*simp only*: *mod-div-equality*)
  also have ... = (*a mod c* + *b* + *a div c* ∗ *c*) *mod c*
    by (*simp only*: *add-ac*)
  also have ... = (*a mod c* + *b*) *mod c*
    by (*rule mod-mult-self1*)
  finally show *?thesis* **.**
qed

lemma *mod-add-right-eq*: (*a* + *b*) *mod c* = (*a* + *b mod c*) *mod c*
proof −
  have (*a* + *b*) *mod c* = (*a* + (*b div c* ∗ *c* + *b mod c*)) *mod c*
    by (*simp only*: *mod-div-equality*)
  also have ... = (*a* + *b mod c* + *b div c* ∗ *c*) *mod c*
    by (*simp only*: *add-ac*)
  also have ... = (*a* + *b mod c*) *mod c*
    by (*rule mod-mult-self1*)
  finally show *?thesis* **.**
qed

lemma *mod-add-eq*: (*a* + *b*) *mod c* = (*a mod c* + *b mod c*) *mod c*
by (*rule trans* [*OF mod-add-left-eq mod-add-right-eq*])

lemma *mod-add-cong*:
  assumes *a mod c* = *a′ mod c*
  assumes *b mod c* = *b′ mod c*

shows $(a + b)$ *mod* $c = (a' + b')$ *mod* c
proof −
have $(a$ *mod* $c + b$ *mod* $c)$ *mod* $c = (a'$ *mod* $c + b'$ *mod* $c)$ *mod* c
unfolding *assms* **..**
thus *?thesis*
by (*simp only*: *mod-add-eq* [*symmetric*])
qed

lemma *div-add*[*simp*]: *z dvd x* ⟹ *z dvd y*
⟹ $(x + y)$ *div* $z = x$ *div* $z + y$ *div* z
by(*cases z=0*, *simp*, *unfold dvd-def*, *auto simp add*: *algebra-simps*)

Multiplication respects modular equivalence.

lemma *mod-mult-left-eq*: $(a * b)$ *mod* $c = ((a$ *mod* $c) * b)$ *mod* c
proof −
have $(a * b)$ *mod* $c = ((a$ *div* $c * c + a$ *mod* $c) * b)$ *mod* c
by (*simp only*: *mod-div-equality*)
also have ... $= (a$ *mod* $c * b + a$ *div* $c * b * c)$ *mod* c
by (*simp only*: *algebra-simps*)
also have ... $= (a$ *mod* $c * b)$ *mod* c
by (*rule mod-mult-self1*)
finally show *?thesis* **.**
qed

lemma *mod-mult-right-eq*: $(a * b)$ *mod* $c = (a * (b$ *mod* $c))$ *mod* c
proof −
have $(a * b)$ *mod* $c = (a * (b$ *div* $c * c + b$ *mod* $c))$ *mod* c
by (*simp only*: *mod-div-equality*)
also have ... $= (a * (b$ *mod* $c) + a * (b$ *div* $c) * c)$ *mod* c
by (*simp only*: *algebra-simps*)
also have ... $= (a * (b$ *mod* $c))$ *mod* c
by (*rule mod-mult-self1*)
finally show *?thesis* **.**
qed

lemma *mod-mult-eq*: $(a * b)$ *mod* $c = ((a$ *mod* $c) * (b$ *mod* $c))$ *mod* c
by (*rule trans* [*OF mod-mult-left-eq mod-mult-right-eq*])

lemma *mod-mult-cong*:
assumes a *mod* $c = a'$ *mod* c
assumes b *mod* $c = b'$ *mod* c
shows $(a * b)$ *mod* $c = (a' * b')$ *mod* c
proof −
have $(a$ *mod* $c * (b$ *mod* $c))$ *mod* $c = (a'$ *mod* $c * (b'$ *mod* $c))$ *mod* c
unfolding *assms* **..**
thus *?thesis*
by (*simp only*: *mod-mult-eq* [*symmetric*])
qed

lemma *mod-mod-cancel*:
assumes *c dvd b*
shows *a mod b mod c = a mod c*
proof −
from ‹*c dvd b*› **obtain** *k* **where** *b = c * k*
by (*rule dvdE*)
have *a mod b mod c = a mod (c * k) mod c*
by (*simp only*: ‹*b = c * k*›)
also have ... = (*a mod (c * k) + a div (c * k) * k * c*) *mod c*
by (*simp only*: *mod-mult-self1*)
also have ... = (*a div (c * k) * (c * k) + a mod (c * k*)) *mod c*
by (*simp only*: *add-ac mult-ac*)
also have ... = *a mod c*
by (*simp only*: *mod-div-equality*)
finally show *?thesis* **.**
qed

end

lemma *div-mult-div-if-dvd*: (*y*::*'a*::{*semiring-div*,*no-zero-divisors*}) *dvd x* ⟹
z dvd w ⟹ (*x div y*) * (*w div z*) = (*x * w*) *div* (*y * z*)
unfolding *dvd-def*
apply *clarify*
apply (*case-tac y = 0*)
apply *simp*
apply (*case-tac z = 0*)
apply *simp*
apply (*simp add*: *algebra-simps*)
apply (*subst mult-assoc* [*symmetric*])
apply (*simp add*: *no-zero-divisors*)
done

lemma *div-power*: (*y*::*'a*::{*semiring-div*,*no-zero-divisors*,*recpower*}) *dvd x* ⟹
(*x div y*) ^*n* = *x*^*n div y*^*n*
apply (*induct n*)
apply *simp*
apply(*simp add*: *div-mult-div-if-dvd dvd-power-same*)
done

class *ring-div* = *semiring-div* + *comm-ring-1*
begin

Negation respects modular equivalence.

lemma *mod-minus-eq*: (− *a*) *mod b* = (− (*a mod b*)) *mod b*
proof −
have (− *a*) *mod b* = (− (*a div b * b + a mod b*)) *mod b*
by (*simp only*: *mod-div-equality*)
also have ... = (− (*a mod b*) + − (*a div b*) * *b*) *mod b*

by (*simp only*: *minus-add-distrib minus-mult-left add-ac*)
also have $\ldots = (-\ (a\ mod\ b))\ mod\ b$
by (*rule mod-mult-self1*)
finally show *?thesis* **.**
qed

lemma *mod-minus-cong*:
assumes $a\ mod\ b = a'\ mod\ b$
shows $(-\ a)\ mod\ b = (-\ a')\ mod\ b$
proof −
have $(-\ (a\ mod\ b))\ mod\ b = (-\ (a'\ mod\ b))\ mod\ b$
unfolding *assms* **..**
thus *?thesis*
by (*simp only*: *mod-minus-eq* [*symmetric*])
qed

Subtraction respects modular equivalence.

lemma *mod-diff-left-eq*: $(a - b)\ mod\ c = (a\ mod\ c - b)\ mod\ c$
unfolding *diff-minus*
by (*intro mod-add-cong mod-minus-cong*) *simp-all*

lemma *mod-diff-right-eq*: $(a - b)\ mod\ c = (a - b\ mod\ c)\ mod\ c$
unfolding *diff-minus*
by (*intro mod-add-cong mod-minus-cong*) *simp-all*

lemma *mod-diff-eq*: $(a - b)\ mod\ c = (a\ mod\ c - b\ mod\ c)\ mod\ c$
unfolding *diff-minus*
by (*intro mod-add-cong mod-minus-cong*) *simp-all*

lemma *mod-diff-cong*:
assumes $a\ mod\ c = a'\ mod\ c$
assumes $b\ mod\ c = b'\ mod\ c$
shows $(a - b)\ mod\ c = (a' - b')\ mod\ c$
unfolding *diff-minus* **using** *assms*
by (*intro mod-add-cong mod-minus-cong*)

lemma *dvd-neg-div*: $y\ dvd\ x \Longrightarrow -x\ div\ y = -\ (x\ div\ y)$
apply (*case-tac* $y = 0$) **apply** *simp*
apply (*auto simp add*: *dvd-def*)
apply (*subgoal-tac* $-(y * k) = y * -\ k$)
apply (*erule ssubst*)
apply (*erule div-mult-self1-is-id*)
apply *simp*
done

lemma *dvd-div-neg*: $y\ dvd\ x \Longrightarrow x\ div\ -y = -\ (x\ div\ y)$
apply (*case-tac* $y = 0$) **apply** *simp*
apply (*auto simp add*: *dvd-def*)
apply (*subgoal-tac* $y * k = -y * -k$)

apply (*erule ssubst*)
apply (*rule div-mult-self1-is-id*)
apply *simp*
apply *simp*
done

end

26.3 Division on *nat*

We define *op div* and *op mod* on *nat* by means of a characteristic relation with two input arguments m, n and two output arguments q(uotient) and r(emainder).

definition *divmod-rel* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
divmod-rel m n q r ⟷ *m* = *q* ∗ *n* + *r* ∧ (*if n* > *0 then 0* ≤ *r* ∧ *r* < *n else q* = *0*)

divmod-rel is total:

lemma *divmod-rel-ex*:
obtains *q r* **where** *divmod-rel m n q r*
proof (*cases n* = *0*)
case *True* **with** *that* **show** *thesis*
by (*auto simp add*: *divmod-rel-def*)
next
case *False*
have ∃ *q r*. *m* = *q* ∗ *n* + *r* ∧ *r* < *n*
proof (*induct m*)
case *0* **with** ‹*n* ≠ *0*›
have (*0*::*nat*) = *0* ∗ *n* + *0* ∧ *0* < *n* **by** *simp*
then show *?case* **by** *blast*
next
case (*Suc m*) **then obtain** *q*′ *r*′
where *m*: *m* = *q*′ ∗ *n* + *r*′ **and** *n*: *r*′ < *n* **by** *auto*
then show *?case* **proof** (*cases Suc r*′ < *n*)
case *True*
from *m n* **have** *Suc m* = *q*′ ∗ *n* + *Suc r*′ **by** *simp*
with *True* **show** *?thesis* **by** *blast*
next
case *False* **then have** *n* ≤ *Suc r*′ **by** *auto*
moreover from *n* **have** *Suc r*′ ≤ *n* **by** *auto*
ultimately have *n* = *Suc r*′ **by** *auto*
with *m* **have** *Suc m* = *Suc q*′ ∗ *n* + *0* **by** *simp*
with ‹*n* ≠ *0*› **show** *?thesis* **by** *blast*
qed
qed
with *that* **show** *thesis*
using ‹*n* ≠ *0*› **by** (*auto simp add*: *divmod-rel-def*)
qed

divmod-rel is injective:

lemma *divmod-rel-unique-div*:
  assumes *divmod-rel m n q r*
    and *divmod-rel m n q′ r′*
  shows $q = q'$
proof (*cases* $n = 0$)
  case *True* **with** *assms* **show** *?thesis*
    by (*simp add*: *divmod-rel-def*)
next
  case *False*
  have *aux*: $\bigwedge q\ r\ q'\ r'.\ q' * n + r' = q * n + r \Longrightarrow r < n \Longrightarrow q' \leq (q::nat)$
  apply (*rule leI*)
  apply (*subst less-iff-Suc-add*)
  apply (*auto simp add*: *add-mult-distrib*)
  done
  from ‹$n \neq 0$› *assms* **show** *?thesis*
    by (*auto simp add*: *divmod-rel-def*
      intro: *order-antisym dest*: *aux sym*)
qed

lemma *divmod-rel-unique-mod*:
  assumes *divmod-rel m n q r*
    and *divmod-rel m n q′ r′*
  shows $r = r'$
proof −
  from *assms* **have** $q = q'$ **by** (*rule divmod-rel-unique-div*)
  with *assms* **show** *?thesis* **by** (*simp add*: *divmod-rel-def*)
qed

We instantiate divisibility on the natural numbers by means of *divmod-rel*:

instantiation *nat* :: *semiring-div*
begin

definition *divmod* :: $nat \Rightarrow nat \Rightarrow nat \times nat$ **where**
  [*code del*]: *divmod m n* = (*THE* (*q*, *r*). *divmod-rel m n q r*)

definition *div-nat* **where**
  m div n = *fst* (*divmod m n*)

definition *mod-nat* **where**
  m mod n = *snd* (*divmod m n*)

lemma *divmod-div-mod*:
  divmod m n = (*m div n*, *m mod n*)
  unfolding *div-nat-def mod-nat-def* **by** *simp*

lemma *divmod-eq*:
  assumes *divmod-rel m n q r*
  shows *divmod m n* = (*q*, *r*)

using *assms* **by** (*auto simp add*: *divmod-def*
dest: *divmod-rel-unique-div divmod-rel-unique-mod*)

lemma *div-eq*:
assumes *divmod-rel m n q r*
shows *m div n = q*
using *assms* **by** (*auto dest*: *divmod-eq simp add*: *div-nat-def*)

lemma *mod-eq*:
assumes *divmod-rel m n q r*
shows *m mod n = r*
using *assms* **by** (*auto dest*: *divmod-eq simp add*: *mod-nat-def*)

lemma *divmod-rel*: *divmod-rel m n* (*m div n*) (*m mod n*)
proof −
from *divmod-rel-ex*
obtain *q r* **where** *rel*: *divmod-rel m n q r* **.**
moreover with *div-eq mod-eq* **have** *m div n = q* **and** *m mod n = r*
by *simp-all*
ultimately show *?thesis* **by** *simp*
qed

lemma *divmod-zero*:
divmod m 0 = (*0*, *m*)
proof −
from *divmod-rel* [*of m 0*] **show** *?thesis*
unfolding *divmod-div-mod divmod-rel-def* **by** *simp*
qed

lemma *divmod-base*:
assumes *m < n*
shows *divmod m n* = (*0*, *m*)
proof −
from *divmod-rel* [*of m n*] **show** *?thesis*
unfolding *divmod-div-mod divmod-rel-def*
using *assms* **by** (*cases m div n = 0*)
(*auto simp add*: *gr0-conv-Suc* [*of m div n*])
qed

lemma *divmod-step*:
assumes *0 < n* **and** *n ≤ m*
shows *divmod m n* = (*Suc* ((*m − n*) *div n*), (*m − n*) *mod n*)
proof −
from *divmod-rel* **have** *divmod-m-n*: *divmod-rel m n* (*m div n*) (*m mod n*) **.**
with *assms* **have** *m-div-n*: *m div n ≥ 1*
by (*cases m div n*) (*auto simp add*: *divmod-rel-def*)
from *assms divmod-m-n* **have** *divmod-rel* (*m − n*) *n* (*m div n − Suc 0*) (*m mod n*)
by (*cases m div n*) (*auto simp add*: *divmod-rel-def*)

with *divmod-eq* **have** *divmod* $(m - n)$ $n = (m$ *div* $n -$ *Suc* 0, m *mod* $n)$ **by** *simp*
moreover from *divmod-div-mod* **have** *divmod* $(m - n)$ $n = ((m - n)$ *div* n, $(m - n)$ *mod* $n)$ **.**
ultimately have m *div* $n =$ *Suc* $((m - n)$ *div* $n)$
and m *mod* $n = (m - n)$ *mod* n **using** *m-div-n* **by** *simp-all*
then show *?thesis* **using** *divmod-div-mod* **by** *simp*
qed

The "recursion" equations for *op div* and *op mod*

lemma *div-less* [*simp*]:
fixes m n :: *nat*
assumes $m < n$
shows m *div* $n = 0$
using *assms divmod-base divmod-div-mod* **by** *simp*

lemma *le-div-geq*:
fixes m n :: *nat*
assumes $0 < n$ **and** $n \leq m$
shows m *div* $n =$ *Suc* $((m - n)$ *div* $n)$
using *assms divmod-step divmod-div-mod* **by** *simp*

lemma *mod-less* [*simp*]:
fixes m n :: *nat*
assumes $m < n$
shows m *mod* $n = m$
using *assms divmod-base divmod-div-mod* **by** *simp*

lemma *le-mod-geq*:
fixes m n :: *nat*
assumes $n \leq m$
shows m *mod* $n = (m - n)$ *mod* n
using *assms divmod-step divmod-div-mod* **by** (*cases* $n = 0$) *simp-all*

instance proof
fix m n :: *nat* **show** m *div* $n * n + m$ *mod* $n = m$
using *divmod-rel* [*of* m n] **by** (*simp add*: *divmod-rel-def*)
next
fix n :: *nat* **show** n *div* $0 = 0$
using *divmod-zero divmod-div-mod* [*of* n 0] **by** *simp*
next
fix n :: *nat* **show** 0 *div* $n = 0$
using *divmod-rel* [*of* 0 n] **by** (*cases* n) (*simp-all add*: *divmod-rel-def*)
next
fix m n q :: *nat* **assume** $n \neq 0$ **then show** $(q + m * n)$ *div* $n = m + q$ *div* n
by (*induct* m) (*simp-all add*: *le-div-geq*)
qed

end

Simproc for cancelling *op div* and *op mod*

ML ⟪
structure CancelDivModData =
struct

val div-name = @{const-name div};
val mod-name = @{const-name mod};
val mk-binop = HOLogic.mk-binop;
val mk-sum = Nat-Arith.mk-sum;
val dest-sum = Nat-Arith.dest-sum;

*(*logic*)*

val div-mod-eqs = map mk-meta-eq [@{thm div-mod-equality}, @{thm div-mod-equality2}]

val trans = trans

val prove-eq-sums =
let val simps = @{thm add-0} :: @{thm add-0-right} :: @{thms add-ac}
in Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac simps) end;

end;

structure CancelDivMod = CancelDivModFun(CancelDivModData);

val cancel-div-mod-proc = Simplifier.simproc (the-context ())
cancel-div-mod [(m::nat) + n] (K CancelDivMod.proc);

Addsimprocs[cancel-div-mod-proc];
⟫

code generator setup

lemma *divmod-if* [*code*]: *divmod m n = (if n = 0 ∨ m < n then (0, m) else*
let (q, r) = divmod (m − n) n in (Suc q, r))
by (*simp add*: *divmod-zero divmod-base divmod-step*)
(*simp add*: *divmod-div-mod*)

code-modulename *SML*
Divides Nat

code-modulename *OCaml*
Divides Nat

code-modulename *Haskell*
Divides Nat

26.3.1 Quotient

lemma *div-geq*: *0 < n ⟹ ¬ m < n ⟹ m div n = Suc ((m − n) div n)*

by (*simp add*: *le-div-geq linorder-not-less*)

lemma *div-if*: $0 < n \Longrightarrow m\ div\ n = (if\ m < n\ then\ 0\ else\ Suc\ ((m - n)\ div\ n))$
by (*simp add*: *div-geq*)

lemma *div-mult-self-is-m* [*simp*]: $0<n ==> (m*n)\ div\ n = (m::nat)$
by *simp*

lemma *div-mult-self1-is-m* [*simp*]: $0<n ==> (n*m)\ div\ n = (m::nat)$
by *simp*

26.3.2 Remainder

lemma *mod-less-divisor* [*simp*]:
fixes $m\ n :: nat$
assumes $n > 0$
shows $m\ mod\ n < (n::nat)$
using *assms divmod-rel* **unfolding** *divmod-rel-def* **by** *auto*

lemma *mod-less-eq-dividend* [*simp*]:
fixes $m\ n :: nat$
shows $m\ mod\ n \leq m$
proof (*rule add-leD2*)
from *mod-div-equality* **have** $m\ div\ n * n + m\ mod\ n = m$ **.**
then show $m\ div\ n * n + m\ mod\ n \leq m$ **by** *auto*
qed

lemma *mod-geq*: $\neg\ m < (n::nat) \Longrightarrow m\ mod\ n = (m - n)\ mod\ n$
by (*simp add*: *le-mod-geq linorder-not-less*)

lemma *mod-if*: $m\ mod\ (n::nat) = (if\ m < n\ then\ m\ else\ (m - n)\ mod\ n)$
by (*simp add*: *le-mod-geq*)

lemma *mod-1* [*simp*]: $m\ mod\ Suc\ 0 = 0$
by (*induct m*) (*simp-all add*: *mod-geq*)

lemma *mod-mult-distrib*: $(m\ mod\ n) * (k::nat) = (m * k)\ mod\ (n * k)$
apply (*cases* $n = 0$, *simp*)
apply (*cases* $k = 0$, *simp*)
apply (*induct m rule*: *nat-less-induct*)
apply (*subst mod-if*, *simp*)
apply (*simp add*: *mod-geq diff-mult-distrib*)
done

lemma *mod-mult-distrib2*: $(k::nat) * (m\ mod\ n) = (k*m)\ mod\ (k*n)$
by (*simp add*: *mult-commute* [*of k*] *mod-mult-distrib*)

lemma *mult-div-cancel*: $(n::nat) * (m\ div\ n) = m - (m\ mod\ n)$

by (*cut-tac* $a = m$ **and** $b = n$ **in** *mod-div-equality2*, *arith*)

lemma *mod-le-divisor*[*simp*]: $0 < n \Longrightarrow m \bmod n \leq (n::nat)$
apply (*drule mod-less-divisor* [**where** $m = m$])
apply *simp*
done

26.3.3 Quotient and Remainder

lemma *divmod-rel-mult1-eq*:
[| *divmod-rel b c q r*; $c > 0$ |]
==> *divmod-rel* $(a*b)$ c $(a*q + a*r \text{ div } c)$ $(a*r \bmod c)$
by (*auto simp add*: *split-ifs divmod-rel-def algebra-simps*)

lemma *div-mult1-eq*: $(a*b) \text{ div } c = a*(b \text{ div } c) + a*(b \bmod c) \text{ div } (c::nat)$
apply (*cases* $c = 0$, *simp*)
apply (*blast intro*: *divmod-rel* [*THEN divmod-rel-mult1-eq*, *THEN div-eq*])
done

lemma *divmod-rel-add1-eq*:
[| *divmod-rel a c aq ar*; *divmod-rel b c bq br*; $c > 0$ |]
==> *divmod-rel* $(a + b)$ c $(aq + bq + (ar+br)) \text{ div } c)$ $((ar + br) \bmod c)$
by (*auto simp add*: *split-ifs divmod-rel-def algebra-simps*)

lemma *div-add1-eq*:
$(a+b) \text{ div } (c::nat) = a \text{ div } c + b \text{ div } c + ((a \bmod c + b \bmod c) \text{ div } c)$
apply (*cases* $c = 0$, *simp*)
apply (*blast intro*: *divmod-rel-add1-eq* [*THEN div-eq*] *divmod-rel*)
done

lemma *mod-lemma*: [| $(0::nat) < c$; $r < b$ |] ==> $b * (q \bmod c) + r < b * c$
apply (*cut-tac* $m = q$ **and** $n = c$ **in** *mod-less-divisor*)
apply (*drule-tac* [2] $m = q \bmod c$ **in** *less-imp-Suc-add*, *auto*)
apply (*erule-tac* $P = \%x.\ ?lhs < ?rhs\ x$ **in** *ssubst*)
apply (*simp add*: *add-mult-distrib2*)
done

lemma *divmod-rel-mult2-eq*: [| *divmod-rel a b q r*; $0 < b$; $0 < c$ |]
==> *divmod-rel* a $(b*c)$ $(q \text{ div } c)$ $(b*(q \bmod c) + r)$
by (*auto simp add*: *mult-ac divmod-rel-def add-mult-distrib2* [*symmetric*] *mod-lemma*)

lemma *div-mult2-eq*: $a \text{ div } (b*c) = (a \text{ div } b) \text{ div } (c::nat)$
apply (*cases* $b = 0$, *simp*)
apply (*cases* $c = 0$, *simp*)
apply (*force simp add*: *divmod-rel* [*THEN divmod-rel-mult2-eq*, *THEN div-eq*])
done

lemma *mod-mult2-eq*: $a \bmod (b*c) = b*(a \text{ div } b \bmod c) + a \bmod (b::nat)$

apply (*cases b = 0*, *simp*)
apply (*cases c = 0*, *simp*)
apply (*auto simp add*: *mult-commute divmod-rel* [*THEN divmod-rel-mult2-eq*, *THEN mod-eq*])
done

26.3.4 Cancellation of Common Factors in Division

lemma *div-mult-mult-lemma*:
[| (*0*::*nat*) < *b*; *0* < *c* |] ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
by (*auto simp add*: *div-mult2-eq*)

lemma *div-mult-mult1* [*simp*]: (*0*::*nat*) < *c* ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
apply (*cases b = 0*)
apply (*auto simp add*: *linorder-neq-iff* [*of b*] *div-mult-mult-lemma*)
done

lemma *div-mult-mult2* [*simp*]: (*0*::*nat*) < *c* ==> (*a*∗*c*) *div* (*b*∗*c*) = *a div b*
apply (*drule div-mult-mult1*)
apply (*auto simp add*: *mult-commute*)
done

26.3.5 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: *m div Suc 0* = *m*
by (*induct m*) (*simp-all add*: *div-geq*)

lemma *div-le-mono* [*rule-format*]:
∀ *m*::*nat*. *m* ≤ *n* −−> (*m div k*) ≤ (*n div k*)
apply (*case-tac k=0*, *simp*)
apply (*induct n rule*: *nat-less-induct*, *clarify*)
apply (*case-tac n<k*)

apply *simp*

apply (*case-tac m<k*)

apply *simp*

apply (*simp add*: *div-geq diff-le-mono*)
done

lemma *div-le-mono2*: !!*m*::*nat*. [| *0*<*m*; *m*≤*n* |] ==> (*k div n*) ≤ (*k div m*)
apply (*subgoal-tac 0*<*n*)
prefer *2* **apply** *simp*
apply (*induct-tac k rule*: *nat-less-induct*)
apply (*rename-tac k*)

apply (*case-tac k<n*, *simp*)
apply (*subgoal-tac* ~ (*k<m*))
prefer *2* **apply** *simp*
apply (*simp add*: *div-geq*)
apply (*subgoal-tac* (*k−n*) *div n* ≤ (*k−m*) *div n*)
prefer *2*
apply (*blast intro*: *div-le-mono diff-le-mono2*)
apply (*rule le-trans*, *simp*)
apply (*simp*)
done

lemma *div-le-dividend* [*simp*]: *m div n* ≤ (*m*::*nat*)
apply (*case-tac n=0*, *simp*)
apply (*subgoal-tac m div n* ≤ *m div 1*, *simp*)
apply (*rule div-le-mono2*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *div-less-dividend* [*rule-format*]:
!!*n*::*nat*. *1<n* ==> *0* < *m* −−> *m div n* < *m*
apply (*induct-tac m rule*: *nat-less-induct*)
apply (*rename-tac m*)
apply (*case-tac m<n*, *simp*)
apply (*subgoal-tac 0<n*)
prefer *2* **apply** *simp*
apply (*simp add*: *div-geq*)
apply (*case-tac n<m*)
apply (*subgoal-tac* (*m−n*) *div n* < (*m−n*))
apply (*rule impI less-trans-Suc*)+
apply *assumption*
apply (*simp-all*)
done

lemma *nat-div-eq-0* [*simp*]: (*n*::*nat*) > *0* ==> ((*m div n*) = *0*) = (*m* < *n*)
by(*auto*, *subst mod-div-equality* [*of m n*, *symmetric*], *auto*)

lemma *nat-div-gt-0* [*simp*]: (*n*::*nat*) > *0* ==> ((*m div n*) > *0*) = (*m* >= *n*)
by (*subst neq0-conv* [*symmetric*], *auto*)

declare *div-less-dividend* [*simp*]

A fact for the mutilated chess board

lemma *mod-Suc*: *Suc*(*m*) *mod n* = (*if Suc*(*m mod n*) = *n then 0 else Suc*(*m mod n*))
apply (*case-tac n=0*, *simp*)
apply (*induct m rule*: *nat-less-induct*)
apply (*case-tac Suc* (*na*) <*n*)

apply (*frule lessI* [*THEN less-trans*], *simp add*: *less-not-refl3*)

apply (*simp add*: *linorder-not-less le-Suc-eq mod-geq*)
apply (*auto simp add*: *Suc-diff-le le-mod-geq*)
done

26.3.6 The Divides Relation

lemma *dvd-1-left* [*iff*]: *Suc 0 dvd k*
unfolding *dvd-def* **by** *simp*

lemma *dvd-1-iff-1* [*simp*]: (*m dvd Suc 0*) = (*m = Suc 0*)
by (*simp add*: *dvd-def*)

lemma *nat-dvd-1-iff-1* [*simp*]: *m dvd* (*1*::*nat*) ⟷ *m = 1*
by (*simp add*: *dvd-def*)

lemma *dvd-anti-sym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n*::*nat*)
unfolding *dvd-def*
by (*force dest*: *mult-eq-self-implies-10 simp add*: *mult-assoc mult-eq-1-iff*)

op dvd is a partial order

interpretation *dvd*: *order op dvd* λ*n m* :: *nat*. *n dvd m* ∧ ¬ *m dvd n*
proof qed (*auto intro*: *dvd-refl dvd-trans dvd-anti-sym*)

lemma *nat-dvd-diff* [*simp*]: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m*−*n* :: *nat*)
unfolding *dvd-def*
by (*blast intro*: *diff-mult-distrib2* [*symmetric*])

lemma *dvd-diffD*: [| *k dvd m*−*n*; *k dvd n*; *n*≤*m* |] ==> *k dvd* (*m*::*nat*)
apply (*erule linorder-not-less* [*THEN iffD2*, *THEN add-diff-inverse*, *THEN subst*])
apply (*blast intro*: *dvd-add*)
done

lemma *dvd-diffD1*: [| *k dvd m*−*n*; *k dvd m*; *n*≤*m* |] ==> *k dvd* (*n*::*nat*)
by (*drule-tac m = m* **in** *nat-dvd-diff*, *auto*)

lemma *dvd-reduce*: (*k dvd n* + *k*) = (*k dvd* (*n*::*nat*))
apply (*rule iffI*)
apply (*erule-tac* [*2*] *dvd-add*)
apply (*rule-tac* [*2*] *dvd-refl*)
apply (*subgoal-tac n* = (*n*+*k*) −*k*)
prefer *2* **apply** *simp*
apply (*erule ssubst*)
apply (*erule nat-dvd-diff*)
apply (*rule dvd-refl*)
done

lemma *dvd-mod*: !!*n*::*nat*. [| *f dvd m*; *f dvd n* |] ==> *f dvd m mod n*
unfolding *dvd-def*
apply (*case-tac n = 0*, *auto*)
apply (*blast intro*: *mod-mult-distrib2* [*symmetric*])
done

lemma *dvd-mod-iff*: *k dvd n* ==> ((*k*::*nat*) *dvd m mod n*) = (*k dvd m*)
by (*blast intro*: *dvd-mod-imp-dvd dvd-mod*)

lemma *dvd-mult-cancel*: !!*k*::*nat*. [| *k*∗*m dvd k*∗*n*; *0*<*k* |] ==> *m dvd n*
unfolding *dvd-def*
apply (*erule exE*)
apply (*simp add*: *mult-ac*)
done

lemma *dvd-mult-cancel1*: *0*<*m* ==> (*m*∗*n dvd m*) = (*n* = (*1*::*nat*))
apply *auto*
apply (*subgoal-tac m*∗*n dvd m*∗*1*)
apply (*drule dvd-mult-cancel*, *auto*)
done

lemma *dvd-mult-cancel2*: *0*<*m* ==> (*n*∗*m dvd m*) = (*n* = (*1*::*nat*))
apply (*subst mult-commute*)
apply (*erule dvd-mult-cancel1*)
done

lemma *dvd-imp-le*: [| *k dvd n*; *0* < *n* |] ==> *k* ≤ (*n*::*nat*)
by (*auto elim*!: *dvdE*) (*auto simp add*: *gr0-conv-Suc*)

lemma *nat-dvd-not-less*: (*0*::*nat*) < *m* ⟹ *m* < *n* ⟹ ¬ *n dvd m*
by (*auto elim*!: *dvdE*) (*auto simp add*: *gr0-conv-Suc*)

lemma *dvd-mult-div-cancel*: *n dvd m* ==> *n* ∗ (*m div n*) = (*m*::*nat*)
apply (*subgoal-tac m mod n = 0*)
apply (*simp add*: *mult-div-cancel*)
apply (*simp only*: *dvd-eq-mod-eq-0*)
done

lemma *nat-zero-less-power-iff* [*simp*]: (*x*^*n* > *0*) = (*x* > (*0*::*nat*) | *n*=*0*)
by (*induct n*) *auto*

lemma *power-dvd-imp-le*: [|*i*^*m dvd i*^*n*; (*1*::*nat*) < *i*|] ==> *m* ≤ *n*
apply (*rule power-le-imp-le-exp*, *assumption*)
apply (*erule dvd-imp-le*, *simp*)
done

lemma *mod-eq-0-iff*: (*m mod d* = *0*) = (∃ *q*::*nat*. *m* = *d*∗*q*)
by (*auto simp add*: *dvd-eq-mod-eq-0* [*symmetric*] *dvd-def*)

lemmas *mod-eq-0D* [*dest*!] = *mod-eq-0-iff* [*THEN iffD1*]

lemma *mod-eqD*: (*m mod d* = *r*) ==> ∃ *q*::*nat*. *m* = *r* + *q*∗*d*
apply (*cut-tac a* = *m* **in** *mod-div-equality*)
apply (*simp only*: *add-ac*)
apply (*blast intro*: *sym*)
done

lemma *split-div*:
P(*n div k* :: *nat*) =
((*k* = *0* ⟶ *P 0*) ∧ (*k* ≠ *0* ⟶ (!*i*. !*j*<*k*. *n* = *k*∗*i* + *j* ⟶ *P i*)))
(**is** *?P* = *?Q* **is** - = (- ∧ (- ⟶ *?R*)))
proof
assume *P*: *?P*
show *?Q*
proof (*cases*)
assume *k* = *0*
with *P* **show** *?Q* **by** *simp*
next
assume *not0*: *k* ≠ *0*
thus *?Q*
proof (*simp*, *intro allI impI*)
fix *i j*
assume *n*: *n* = *k*∗*i* + *j* **and** *j*: *j* < *k*
show *P i*
proof (*cases*)
assume *i* = *0*
with *n j P* **show** *P i* **by** *simp*
next
assume *i* ≠ *0*
with *not0 n j P* **show** *P i* **by**(*simp add*:*add-ac*)
qed
qed
qed
next
assume *Q*: *?Q*
show *?P*
proof (*cases*)
assume *k* = *0*
with *Q* **show** *?P* **by** *simp*
next
assume *not0*: *k* ≠ *0*
with *Q* **have** *R*: *?R* **by** *simp*
from *not0 R*[*THEN spec*,*of n div k*,*THEN spec*, *of n mod k*]
show *?P* **by** *simp*
qed
qed

lemma *split-div-lemma*:
assumes $0 < n$
shows $n * q \leq m \wedge m < n * Suc\ q \longleftrightarrow q = ((m::nat)\ div\ n)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?rhs*
with *mult-div-cancel* **have** *nq*: $n * q = m - (m\ mod\ n)$ **by** *simp*
then have *A*: $n * q \leq m$ **by** *simp*
have $n - (m\ mod\ n) > 0$ **using** *mod-less-divisor assms* **by** *auto*
then have $m < m + (n - (m\ mod\ n))$ **by** *simp*
then have $m < n + (m - (m\ mod\ n))$ **by** *simp*
with *nq* **have** $m < n + n * q$ **by** *simp*
then have *B*: $m < n * Suc\ q$ **by** *simp*
from *A B* **show** *?lhs* **..**
next
assume *P*: *?lhs*
then have *divmod-rel m n q* $(m - n * q)$
unfolding *divmod-rel-def* **by** (*auto simp add*: *mult-ac*)
then show *?rhs* **using** *divmod-rel* **by** (*rule divmod-rel-unique-div*)
qed

theorem *split-div'*:
$P\ ((m::nat)\ div\ n) = ((n = 0 \wedge P\ 0) \vee$
$(\exists q.\ (n * q \leq m \wedge m < n * (Suc\ q)) \wedge P\ q))$
apply (*case-tac* $0 < n$)
apply (*simp only*: *add*: *split-div-lemma*)
apply *simp-all*
done

lemma *split-mod*:
$P(n\ mod\ k :: nat) =$
$((k = 0 \longrightarrow P\ n) \wedge (k \neq 0 \longrightarrow (!i.\ !j{<}k.\ n = k{*}i + j \longrightarrow P\ j)))$
(**is** *?P* = *?Q* **is** - = (- $\wedge$ (- $\longrightarrow$ *?R*)))
proof
assume *P*: *?P*
show *?Q*
proof (*cases*)
assume $k = 0$
with *P* **show** *?Q* **by** *simp*
next
assume *not0*: $k \neq 0$
thus *?Q*
proof (*simp*, *intro allI impI*)
fix *i j*
assume $n = k{*}i + j\ j < k$
thus *P j* **using** *not0 P* **by**(*simp add*:*add-ac mult-ac*)
qed
qed
next
assume *Q*: *?Q*

```
  show ?P
  proof (cases)
    assume k = 0
    with Q show ?P by simp
  next
    assume not0: k ≠ 0
    with Q have R: ?R by simp
    from not0 R[THEN spec,of n div k,THEN spec, of n mod k]
    show ?P by simp
  qed
qed

theorem mod-div-equality': (m::nat) mod n = m - (m div n) * n
  apply (rule-tac P=%x. m mod n = x - (m div n) * n in
    subst [OF mod-div-equality [of - n]])
  apply arith
  done

lemma div-mod-equality':
  fixes m n :: nat
  shows m div n * n = m - m mod n
proof -
  have m mod n ≤ m mod n ..
  from div-mod-equality have
    m div n * n + m mod n - m mod n = m - m mod n by simp
  with diff-add-assoc [OF ‹m mod n ≤ m mod n›, of m div n * n] have
    m div n * n + (m mod n - m mod n) = m - m mod n
    by simp
  then show ?thesis by simp
qed
```

26.3.7 An "induction" law for modulus arithmetic.

```
lemma mod-induct-0:
  assumes step: ∀ i<p. P i ⟶ P ((Suc i) mod p)
  and base: P i and i: i<p
  shows P 0
proof (rule ccontr)
  assume contra: ¬(P 0)
  from i have p: 0<p by simp
  have ∀ k. 0<k ⟶ ¬ P (p-k) (is ∀ k. ?A k)
  proof
    fix k
    show ?A k
    proof (induct k)
      show ?A 0 by simp  — by contradiction
    next
      fix n
      assume ih: ?A n
```

```
      show ?A (Suc n)
      proof (clarsimp)
        assume y: P (p − Suc n)
        have n: Suc n < p
        proof (rule ccontr)
          assume ¬(Suc n < p)
          hence p − Suc n = 0
            by simp
          with y contra show False
            by simp
        qed
        hence n2: Suc (p − Suc n) = p−n by arith
        from p have p − Suc n < p by arith
        with y step have z: P ((Suc (p − Suc n)) mod p)
          by blast
        show False
        proof (cases n=0)
          case True
          with z n2 contra show ?thesis by simp
        next
          case False
          with p have p−n < p by arith
          with z n2 False ih show ?thesis by simp
        qed
      qed
    qed
  qed
  moreover
  from i obtain k where 0<k ∧ i+k=p
    by (blast dest: less-imp-add-positive)
  hence 0<k ∧ i=p−k by auto
  moreover
  note base
  ultimately
  show False by blast
qed

lemma mod-induct:
  assumes step: ∀ i<p. P i ⟶ P ((Suc i) mod p)
  and base: P i and i: i<p and j: j<p
  shows P j
proof −
  have ∀ j<p. P j
  proof
    fix j
    show j<p ⟶ P j (is ?A j)
    proof (induct j)
      from step base i show ?A 0
        by (auto elim: mod-induct-0)
```

```
    next
      fix k
      assume ih: ?A k
      show ?A (Suc k)
      proof
        assume suc: Suc k < p
        hence k: k<p by simp
        with ih have P k ..
        with step k have P (Suc k mod p)
          by blast
        moreover
        from suc have Suc k mod p = Suc k
          by simp
        ultimately
        show P (Suc k) by simp
      qed
    qed
  qed
  with j show ?thesis by blast
qed

end
```

27 Plain: Plain HOL

theory *Plain*
imports *Datatype FunDef Record Extraction Divides*
begin

Plain bootstrap of fundamental HOL tools and packages; does not include *Hilbert-Choice*.

ML ⟪ *path-add* ~~/*src/HOL/Library* ⟫

end

28 Relation-Power: Powers of Relations and Functions

theory *Relation-Power*
imports *Power Transitive-Closure Plain*
begin

instance
fun :: (*type*, *type*) *power* **..**
— only type $'a \Rightarrow 'a$ should be in class *power*!

overloading
relpow ≡ *power* :: $('a \times 'a)$ *set* ⇒ *nat* ⇒ $('a \times 'a)$ *set* (**unchecked**)
begin

$R \hat{} n = R\ O\ ...\ O\ R$, the n-fold composition of R

primrec *relpow* **where**
$(R :: ('a \times 'a)\ set) \hat{} 0 = Id$
| $(R :: ('a \times 'a)\ set) \hat{} Suc\ n = R\ O\ (R \hat{} n)$

end

overloading
funpow ≡ *power* :: $('a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a$ (**unchecked**)
begin

$f \hat{} n = f\ o\ ...\ o\ f$, the n-fold composition of f

primrec *funpow* **where**
$(f :: 'a \Rightarrow 'a) \hat{} 0 = id$
| $(f :: 'a \Rightarrow 'a) \hat{} Suc\ n = f\ o\ (f \hat{} n)$

end

WARNING: due to the limits of Isabelle's type classes, exponentiation on functions and relations has too general a domain, namely $('a \times 'b)$ *set* and $'a \Rightarrow 'b$. Explicit type constraints may therefore be necessary. For example, *range* $(f \hat{} n) = A$ and *Range* $(R \hat{} n) = B$ need constraints.

Circumvent this problem for code generation:

primrec
fun-pow :: *nat* ⇒ $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where
fun-pow 0 f = *id*
| *fun-pow* (*Suc n*) *f* = *f o fun-pow n f*

lemma *funpow-fun-pow* [*code unfold*]: $f \hat{} n$ = *fun-pow n f*
unfolding *funpow-def fun-pow-def* **..**

lemma *funpow-add*: $f \hat{} (m+n) = f \hat{} m\ o\ f \hat{} n$
by (*induct m*) *simp-all*

lemma *funpow-swap1*: $f((f \hat{} n)\ x) = (f \hat{} n)(f\ x)$
proof −
have $f((f \hat{} n)\ x) = (f \hat{} (n+1))\ x$ **unfolding** *One-nat-def* **by** *simp*
also have ... $= (f \hat{} n\ o\ f \hat{} 1)\ x$ **by** (*simp only*: *funpow-add*)
also have ... $= (f \hat{} n)(f\ x)$ **unfolding** *One-nat-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *rel-pow-1* [*simp*]:
fixes $R :: ('a*'a)set$
shows $R\hat{}1 = R$
unfolding *One-nat-def* **by** *simp*

lemma *rel-pow-0-I*: $(x,x) : R\hat{}0$
by *simp*

lemma *rel-pow-Suc-I*: $[|\ (x,y) : R\hat{}n;\ (y,z):R\ |] ==> (x,z):R\hat{}(Suc\ n)$
by *auto*

lemma *rel-pow-Suc-I2*:
$(x,\ y) : R \Longrightarrow (y,\ z) : R\hat{}n \Longrightarrow (x,z) : R\hat{}(Suc\ n)$
apply (*induct n arbitrary*: *z*)
apply *simp*
apply *fastsimp*
done

lemma *rel-pow-0-E*: $[|\ (x,y) : R\hat{}0;\ x=y ==> P\ |] ==> P$
by *simp*

lemma *rel-pow-Suc-E*:
$[|\ (x,z) : R\hat{}(Suc\ n);\ \ !!y.\ [|\ (x,y) : R\hat{}n;\ (y,z) : R\ |] ==> P\ |] ==> P$
by *auto*

lemma *rel-pow-E*:
$[|\ (x,z) : R\hat{}n;\ \ [|\ n=0;\ x = z\ |] ==> P;$
$\quad !!y\ m.\ [|\ n = Suc\ m;\ (x,y) : R\hat{}m;\ (y,z) : R\ |] ==> P$
$\ |] ==> P$
by (*cases n*) *auto*

lemma *rel-pow-Suc-D2*:
$(x,\ z) : R\hat{}(Suc\ n) \Longrightarrow (\exists\, y.\ (x,y) : R\ \&\ (y,z) : R\hat{}n)$
apply (*induct n arbitrary*: *x z*)
apply (*blast intro*: *rel-pow-0-I elim*: *rel-pow-0-E rel-pow-Suc-E*)
apply (*blast intro*: *rel-pow-Suc-I elim*: *rel-pow-0-E rel-pow-Suc-E*)
done

lemma *rel-pow-Suc-D2'*:
$\forall x\ y\ z.\ (x,y) : R\hat{}n\ \&\ (y,z) : R --> (\exists\, w.\ (x,w) : R\ \&\ (w,z) : R\hat{}n)$
by (*induct n*) (*simp-all*, *blast*)

lemma *rel-pow-E2*:
$[|\ (x,z) : R\hat{}n;\ \ [|\ n=0;\ x = z\ |] ==> P;$
$\quad !!y\ m.\ [|\ n = Suc\ m;\ (x,y) : R;\ (y,z) : R\hat{}m\ |] ==> P$
$\ |] ==> P$
apply (*case-tac n*, *simp*)
apply (*cut-tac n=nat* **and** *R=R* **in** *rel-pow-Suc-D2'*, *simp*, *blast*)
done

lemma *rtrancl-imp-UN-rel-pow*: !!*p*. *p*:*R* ^* ==> *p* : (*UN n*. *R* ^*n*)
apply (*simp only*: *split-tupled-all*)
apply (*erule rtrancl-induct*)
apply (*blast intro*: *rel-pow-0-I rel-pow-Suc-I*)+
done

lemma *rel-pow-imp-rtrancl*: !!*p*. *p*:*R* ^*n* ==> *p*:*R* ^*
apply (*simp only*: *split-tupled-all*)
apply (*induct n*)
apply (*blast intro*: *rtrancl-refl elim*: *rel-pow-0-E*)
apply (*blast elim*: *rel-pow-Suc-E intro*: *rtrancl-into-rtrancl*)
done

lemma *rtrancl-is-UN-rel-pow*: *R* ^* = (*UN n*. *R* ^*n*)
by (*blast intro*: *rtrancl-imp-UN-rel-pow rel-pow-imp-rtrancl*)

lemma *trancl-power*:
x ∈ *r* ^+ = (∃ *n* > *0*. *x* ∈ *r* ^*n*)
apply (*cases x*)
apply *simp*
apply (*rule iffI*)
apply (*drule tranclD2*)
apply (*clarsimp simp*: *rtrancl-is-UN-rel-pow*)
apply (*rule-tac x*=*Suc x* **in** *exI*)
apply (*clarsimp simp*: *rel-comp-def*)
apply *fastsimp*
apply *clarsimp*
apply (*case-tac n*, *simp*)
apply *clarsimp*
apply (*drule rel-pow-imp-rtrancl*)
apply *fastsimp*
done

lemma *single-valued-rel-pow*:
!!*r*::('*a* * '*a*)*set*. *single-valued r* ==> *single-valued* (*r* ^*n*)
apply (*rule single-valuedI*)
apply (*induct n*)
apply *simp*
apply (*fast dest*: *single-valuedD elim*: *rel-pow-Suc-E*)
done

ML
⟪
val funpow-add = *thm funpow-add*;
val rel-pow-1 = *thm rel-pow-1*;
val rel-pow-0-I = *thm rel-pow-0-I*;
val rel-pow-Suc-I = *thm rel-pow-Suc-I*;
val rel-pow-Suc-I2 = *thm rel-pow-Suc-I2*;

val rel-pow-0-E = thm rel-pow-0-E;
val rel-pow-Suc-E = thm rel-pow-Suc-E;
val rel-pow-E = thm rel-pow-E;
val rel-pow-Suc-D2 = thm rel-pow-Suc-D2;
val rel-pow-Suc-D2 = thm rel-pow-Suc-D2;
val rel-pow-E2 = thm rel-pow-E2;
val rtrancl-imp-UN-rel-pow = thm rtrancl-imp-UN-rel-pow;
val rel-pow-imp-rtrancl = thm rel-pow-imp-rtrancl;
val rtrancl-is-UN-rel-pow = thm rtrancl-is-UN-rel-pow;
val single-valued-rel-pow = thm single-valued-rel-pow;
⟫

end

29 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *Finite-Set Relation Plain*
begin

29.1 Equivalence relations

locale *equiv* =
  fixes *A* **and** *r*
  assumes *refl-on*: *refl-on A r*
    and *sym*: *sym r*
    and *trans*: *trans r*

Suppes, Theorem 70: r is an equiv relation iff $r^{-1}\ O\ r = r$.
First half: *equiv A r ==> r⁻¹ O r = r*.

lemma *sym-trans-comp-subset*:
    sym r ==> trans r ==> r⁻¹ O r ⊆ r
  by (*unfold trans-def sym-def converse-def*) *blast*

lemma *refl-on-comp-subset*: *refl-on A r ==> r ⊆ r⁻¹ O r*
  by (*unfold refl-on-def*) *blast*

lemma *equiv-comp-eq*: *equiv A r ==> r⁻¹ O r = r*
  apply (*unfold equiv-def*)
  apply *clarify*
  apply (*rule equalityI*)
   apply (*iprover intro*: *sym-trans-comp-subset refl-on-comp-subset*)+
  done

Second half.

lemma *comp-equivI*:

r^{-1} *O* $r = r$ ==> *Domain* $r = A$ ==> *equiv* A r
apply (*unfold equiv-def refl-on-def sym-def trans-def*)
apply (*erule equalityE*)
apply (*subgoal-tac* $\forall x\ y.\ (x, y) \in r$ −−> $(y, x) \in r$)
apply *fast*
apply *fast*
done

29.2 Equivalence classes

lemma *equiv-class-subset*:
equiv A r ==> $(a, b) \in r$ ==> $r``\{a\} \subseteq r``\{b\}$
— lemma for the next result
by (*unfold equiv-def trans-def sym-def*) *blast*

theorem *equiv-class-eq*: *equiv* A r ==> $(a, b) \in r$ ==> $r``\{a\} = r``\{b\}$
apply (*assumption* | *rule equalityI equiv-class-subset*)+
apply (*unfold equiv-def sym-def*)
apply *blast*
done

lemma *equiv-class-self*: *equiv* A r ==> $a \in A$ ==> $a \in r``\{a\}$
by (*unfold equiv-def refl-on-def*) *blast*

lemma *subset-equiv-class*:
equiv A r ==> $r``\{b\} \subseteq r``\{a\}$ ==> $b \in A$ ==> $(a,b) \in r$
— lemma for the next result
by (*unfold equiv-def refl-on-def*) *blast*

lemma *eq-equiv-class*:
$r``\{a\} = r``\{b\}$ ==> *equiv* A r ==> $b \in A$ ==> $(a, b) \in r$
by (*iprover intro*: *equalityD2 subset-equiv-class*)

lemma *equiv-class-nondisjoint*:
equiv A r ==> $x \in (r``\{a\} \cap r``\{b\})$ ==> $(a, b) \in r$
by (*unfold equiv-def trans-def sym-def*) *blast*

lemma *equiv-type*: *equiv* A r ==> $r \subseteq A \times A$
by (*unfold equiv-def refl-on-def*) *blast*

theorem *equiv-class-eq-iff*:
equiv A r ==> $((x, y) \in r) = (r``\{x\} = r``\{y\}\ \&\ x \in A\ \&\ y \in A)$
by (*blast intro*!: *equiv-class-eq dest*: *eq-equiv-class equiv-type*)

theorem *eq-equiv-class-iff*:
equiv A r ==> $x \in A$ ==> $y \in A$ ==> $(r``\{x\} = r``\{y\}) = ((x, y) \in r)$
by (*blast intro*!: *equiv-class-eq dest*: *eq-equiv-class equiv-type*)

29.3 Quotients

definition *quotient* :: *'a set* ⇒ (*'a* × *'a*) *set* ⇒ *'a set set* (**infixl** *'//'* *90*) **where**
[*code del*]: $A//r = (\bigcup x \in A.\ \{r``\{x\}\})$ — set of equiv classes

lemma *quotientI*: $x \in A$ ==> $r``\{x\} \in A//r$
by (*unfold quotient-def*) *blast*

lemma *quotientE*:
$X \in A//r$ ==> (!!x. $X = r``\{x\}$ ==> $x \in A$ ==> P) ==> P
by (*unfold quotient-def*) *blast*

lemma *Union-quotient*: *equiv A r* ==> *Union* $(A//r) = A$
by (*unfold equiv-def refl-on-def quotient-def*) *blast*

lemma *quotient-disj*:
equiv A r ==> $X \in A//r$ ==> $Y \in A//r$ ==> $X = Y \mid (X \cap Y = \{\})$
apply (*unfold quotient-def*)
apply *clarify*
apply (*rule equiv-class-eq*)
apply *assumption*
apply (*unfold equiv-def trans-def sym-def*)
apply *blast*
done

lemma *quotient-eqI*:
[|*equiv A r*; $X \in A//r$; $Y \in A//r$; $x \in X$; $y \in Y$; $(x,y) \in r$|] ==> $X = Y$
apply (*clarify elim*!: *quotientE*)
apply (*rule equiv-class-eq*, *assumption*)
apply (*unfold equiv-def sym-def trans-def*, *blast*)
done

lemma *quotient-eq-iff*:
[|*equiv A r*; $X \in A//r$; $Y \in A//r$; $x \in X$; $y \in Y$|] ==> $(X = Y) = ((x,y) \in r)$
apply (*rule iffI*)
prefer *2* **apply** (*blast del*: *equalityI intro*: *quotient-eqI*)
apply (*clarify elim*!: *quotientE*)
apply (*unfold equiv-def sym-def trans-def*, *blast*)
done

lemma *eq-equiv-class-iff2*:
⟦ *equiv A r*; $x \in A$; $y \in A$ ⟧ ⟹ $(\{x\}//r = \{y\}//r) = ((x,y) : r)$
by(*simp add*:*quotient-def eq-equiv-class-iff*)

lemma *quotient-empty* [*simp*]: $\{\}//r = \{\}$
by(*simp add*: *quotient-def*)

lemma *quotient-is-empty* [*iff*]: $(A//r = \{\}) = (A = \{\})$

by(*simp add*: *quotient-def*)

lemma *quotient-is-empty2* [*iff*]: $(\{\} = A//r) = (A = \{\})$
by(*simp add*: *quotient-def*)

lemma *singleton-quotient*: $\{x\}//r = \{r\ ``\ \{x\}\}$
by(*simp add*:*quotient-def*)

lemma *quotient-diff1*:
$[\![\ inj\text{-}on\ (\%a.\ \{a\}//r)\ A;\ a \in A\ ]\!] \Longrightarrow (A - \{a\})//r = A//r - \{a\}//r$
apply(*simp add*:*quotient-def inj-on-def*)
apply *blast*
done

29.4 Defining unary operations upon equivalence classes

A congruence-preserving function

locale *congruent* =
fixes r **and** f
assumes *congruent*: $(y,z) \in r$ ==> $f\ y = f\ z$

abbreviation
$RESPECTS :: ('a => 'b) => ('a * 'a)\ set => bool$
(**infixr** *respects 80*) **where**
$f\ respects\ r == congruent\ r\ f$

lemma *UN-constant-eq*: $a \in A$ ==> $\forall y \in A.\ f\ y = c$ ==> $(\bigcup y \in A.\ f(y))=c$
— lemma required to prove *UN-equiv-class*
by *auto*

lemma *UN-equiv-class*:
$equiv\ A\ r$ ==> $f\ respects\ r$ ==> $a \in A$
==> $(\bigcup x \in r``\{a\}.\ f\ x) = f\ a$
— Conversion rule
apply (*rule equiv-class-self* [*THEN UN-constant-eq*], *assumption*+)
apply (*unfold equiv-def congruent-def sym-def*)
apply (*blast del*: *equalityI*)
done

lemma *UN-equiv-class-type*:
$equiv\ A\ r$ ==> $f\ respects\ r$ ==> $X \in A//r$ ==>
$(!!x.\ x \in A$ ==> $f\ x \in B)$ ==> $(\bigcup x \in X.\ f\ x) \in B$
apply (*unfold quotient-def*)
apply *clarify*
apply (*subst UN-equiv-class*)
apply *auto*
done

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be $!!y.\ y \in A \Longrightarrow f\ y \in B$.

lemma *UN-equiv-class-inject*:
equiv A r ==> f respects r ==>
(⋃x ∈ X. f x) = (⋃y ∈ Y. f y) ==> X ∈ A//r ==> Y ∈ A//r
==> (!!x y. x ∈ A ==> y ∈ A ==> f x = f y ==> (x, y) ∈ r)
==> X = Y
apply (*unfold quotient-def*)
apply *clarify*
apply (*rule equiv-class-eq*)
apply *assumption*
apply (*subgoal-tac f x = f xa*)
apply *blast*
apply (*erule box-equals*)
apply (*assumption | rule UN-equiv-class*)+
done

29.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

locale *congruent2* =
fixes *r1* **and** *r2* **and** *f*
assumes *congruent2*:
(y1,z1) ∈ r1 ==> (y2,z2) ∈ r2 ==> f y1 y2 = f z1 z2

Abbreviation for the common case where the relations are identical

abbreviation
RESPECTS2:: ['*a* => '*a* => '*b*, ('*a* * '*a*) *set*] => *bool*
(**infixr** *respects2 80*) **where**
f respects2 r == congruent2 r r f

lemma *congruent2-implies-congruent*:
equiv A r1 ==> congruent2 r1 r2 f ==> a ∈ A ==> congruent r2 (f a)
by (*unfold congruent-def congruent2-def equiv-def refl-on-def*) *blast*

lemma *congruent2-implies-congruent-UN*:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a ∈ A2 ==>
congruent r1 (λx1. ⋃x2 ∈ r2``{a}. f x1 x2)
apply (*unfold congruent-def*)
apply *clarify*
apply (*rule equiv-type* [*THEN subsetD*, *THEN SigmaE2*], *assumption*+)
apply (*simp add*: *UN-equiv-class congruent2-implies-congruent*)
apply (*unfold congruent2-def equiv-def refl-on-def*)
apply (*blast del*: *equalityI*)
done

lemma *UN-equiv-class2*:

equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f ==> a1 ∈ A1 ==> a2 ∈ A2
==> (⋃x1 ∈ r1``{a1}. ⋃x2 ∈ r2``{a2}. f x1 x2) = f a1 a2
by (*simp add*: *UN-equiv-class congruent2-implies-congruent congruent2-implies-congruent-UN*)

lemma *UN-equiv-class-type2*:
equiv A1 r1 ==> equiv A2 r2 ==> congruent2 r1 r2 f
==> X1 ∈ A1//r1 ==> X2 ∈ A2//r2
==> (!!x1 x2. x1 ∈ A1 ==> x2 ∈ A2 ==> f x1 x2 ∈ B)
==> (⋃x1 ∈ X1. ⋃x2 ∈ X2. f x1 x2) ∈ B
apply (*unfold quotient-def*)
apply *clarify*
apply (*blast intro*: *UN-equiv-class-type congruent2-implies-congruent-UN congruent2-implies-congruent quotientI*)
done

lemma *UN-UN-split-split-eq*:
(⋃(*x1*, *x2*) ∈ *X*. ⋃(*y1*, *y2*) ∈ *Y*. *A x1 x2 y1 y2*) =
(⋃*x* ∈ *X*. ⋃*y* ∈ *Y*. (λ(*x1*, *x2*). (λ(*y1*, *y2*). *A x1 x2 y1 y2*) *y*) *x*)
— Allows a natural expression of binary operators,
— without explicit calls to *split*
by *auto*

lemma *congruent2I*:
equiv A1 r1 ==> equiv A2 r2
==> (!!y z w. w ∈ A2 ==> (y,z) ∈ r1 ==> f y w = f z w)
==> (!!y z w. w ∈ A1 ==> (y,z) ∈ r2 ==> f w y = f w z)
==> congruent2 r1 r2 f
— Suggested by John Harrison – the two subproofs may be
— *much* simpler than the direct proof.
apply (*unfold congruent2-def equiv-def refl-on-def*)
apply *clarify*
apply (*blast intro*: *trans*)
done

lemma *congruent2-commuteI*:
assumes *equivA*: *equiv A r*
and *commute*: !!*y z*. *y ∈ A ==> z ∈ A ==> f y z = f z y*
and *congt*: !!*y z w*. *w ∈ A ==> (y,z) ∈ r ==> f w y = f w z*
shows *f respects2 r*
apply (*rule congruent2I* [*OF equivA equivA*])
apply (*rule commute* [*THEN trans*])
apply (*rule-tac* [*3*] *commute* [*THEN trans*, *symmetric*])
apply (*rule-tac* [*5*] *sym*)
apply (*rule congt* | *assumption* |
erule equivA [*THEN equiv-type*, *THEN subsetD*, *THEN SigmaE2*])+
done

29.6 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*: *finite A ==> r ⊆ A × A ==> finite (A//r)*
— recall *equiv ?A ?r ⟹ ?r ⊆ ?A × ?A*
apply (*rule finite-subset*)
apply (*erule-tac* [*2*] *finite-Pow-iff* [*THEN iffD2*])
apply (*unfold quotient-def*)
apply *blast*
done

lemma *finite-equiv-class*:
finite A ==> r ⊆ A × A ==> X ∈ A//r ==> finite X
apply (*unfold quotient-def*)
apply (*rule finite-subset*)
prefer *2* **apply** *assumption*
apply *blast*
done

lemma *equiv-imp-dvd-card*:
finite A ==> equiv A r ==> ∀ X ∈ A//r. k dvd card X
==> k dvd card A
apply (*rule Union-quotient* [*THEN subst* [**where** *P=λA. k dvd card A*]])
apply *assumption*
apply (*rule dvd-partition*)
prefer *3* **apply** (*blast dest*: *quotient-disj*)
apply (*simp-all add*: *Union-quotient equiv-type*)
done

lemma *card-quotient-disjoint*:
⟦ *finite A*; *inj-on* (*λx*. {*x*} // *r*) *A* ⟧ ⟹ *card*(*A*//*r*) = *card A*
apply(*simp add*:*quotient-def*)
apply(*subst card-UN-disjoint*)
apply *assumption*
apply *simp*
apply(*fastsimp simp add*:*inj-on-def*)
apply (*simp add*:*setsum-constant*)
done

end

30 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *Int*
imports *Equiv-Relations Nat Wellfounded*
uses

(*Tools/numeral.ML*)
(*Tools/numeral-syntax.ML*)
~~/*src/Provers/Arith/assoc-fold.ML*
~~/*src/Provers/Arith/cancel-numerals.ML*
~~/*src/Provers/Arith/combine-numerals.ML*
(*Tools/int-arith.ML*)
begin

30.1 The equivalence relation underlying the integers

definition *intrel* :: $((nat \times nat) \times (nat \times nat))$ *set* **where**
[*code del*]: $intrel = \{((x, y), (u, v)) \mid x\ y\ u\ v.\ x + v = u + y\ \}$

typedef (*Integ*)
$int = UNIV//intrel$
by (*auto simp add*: *quotient-def*)

instantiation *int* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *ord*, *abs*, *sgn*}
begin

definition
Zero-int-def [*code del*]: $0 = Abs\text{-}Integ\ (intrel\ ``\ \{(0, 0)\})$

definition
One-int-def [*code del*]: $1 = Abs\text{-}Integ\ (intrel\ ``\ \{(1, 0)\})$

definition
add-int-def [*code del*]: $z + w = Abs\text{-}Integ$
$(\bigcup(x, y) \in Rep\text{-}Integ\ z.\ \bigcup(u, v) \in Rep\text{-}Integ\ w.$
$intrel\ ``\ \{(x + u, y + v)\})$

definition
minus-int-def [*code del*]:
$- z = Abs\text{-}Integ\ (\bigcup(x, y) \in Rep\text{-}Integ\ z.\ intrel\ ``\ \{(y, x)\})$

definition
diff-int-def [*code del*]: $z - w = z + (-w :: int)$

definition
mult-int-def [*code del*]: $z * w = Abs\text{-}Integ$
$(\bigcup(x, y) \in Rep\text{-}Integ\ z.\ \bigcup(u,v\ ) \in Rep\text{-}Integ\ w.$
$intrel\ ``\ \{(x*u + y*v, x*v + y*u)\})$

definition
le-int-def [*code del*]:
$z \leq w \longleftrightarrow (\exists x\ y\ u\ v.\ x+v \leq u+y \wedge (x, y) \in Rep\text{-}Integ\ z \wedge (u, v) \in Rep\text{-}Integ\ w)$

definition

less-int-def [*code del*]: $(z::int) < w \longleftrightarrow z \leq w \wedge z \neq w$

definition
zabs-def: $|i::int| = (if\ i < 0\ then\ -\ i\ else\ i)$

definition
zsgn-def: $sgn\ (i::int) = (if\ i=0\ then\ 0\ else\ if\ 0<i\ then\ 1\ else\ -\ 1)$

instance ..

end

30.2 Construction of the Integers

lemma *intrel-iff* [*simp*]: $(((x,y),(u,v)) \in intrel) = (x+v = u+y)$
by (*simp add*: *intrel-def*)

lemma *equiv-intrel*: *equiv UNIV intrel*
by (*simp add*: *intrel-def equiv-def refl-on-def sym-def trans-def*)

Reduces equality of equivalence classes to the *intrel* relation: $(intrel\ ``\ \{x\} = intrel\ ``\ \{y\}) = ((x,\ y) \in intrel)$

lemmas *equiv-intrel-iff* [*simp*] = *eq-equiv-class-iff* [*OF equiv-intrel UNIV-I UNIV-I*]

All equivalence classes belong to set of representatives

lemma [*simp*]: $intrel``\{(x,y)\} \in Integ$
by (*auto simp add*: *Integ-def intrel-def quotient-def*)

Reduces equality on abstractions to equality on representatives: $[\![x \in Integ; y \in Integ]\!] \Longrightarrow (Abs\text{-}Integ\ x = Abs\text{-}Integ\ y) = (x = y)$

declare *Abs-Integ-inject* [*simp*,*noatp*] *Abs-Integ-inverse* [*simp*,*noatp*]

Case analysis on the representation of an integer as an equivalence class of pairs of naturals.

lemma *eq-Abs-Integ* [*case-names Abs-Integ*, *cases type*: *int*]:
(!!$x\ y.\ z = Abs\text{-}Integ(intrel``\{(x,y)\}) ==> P) ==> P$
apply (*rule Abs-Integ-cases* [*of z*])
apply (*auto simp add*: *Integ-def quotient-def*)
done

30.3 Arithmetic Operations

lemma *minus*: $-\ Abs\text{-}Integ(intrel``\{(x,y)\}) = Abs\text{-}Integ(intrel\ ``\ \{(y,x)\})$
proof −
have $(\lambda(x,y).\ intrel``\{(y,x)\})$ *respects intrel*
by (*simp add*: *congruent-def*)
thus *?thesis*
by (*simp add*: *minus-int-def UN-equiv-class* [*OF equiv-intrel*])

qed

lemma *add*:
Abs-Integ (*intrel*``{(*x*,*y*)}) + *Abs-Integ* (*intrel*``{(*u*,*v*)}) =
Abs-Integ (*intrel*``{(*x*+*u*, *y*+*v*)})
proof −
have (λ*z w*. (λ(*x*,*y*). (λ(*u*,*v*). *intrel* `` {(*x*+*u*, *y*+*v*)}) *w*) *z*)
respects2 intrel
by (*simp add*: *congruent2-def*)
thus *?thesis*
by (*simp add*: *add-int-def UN-UN-split-split-eq*
UN-equiv-class2 [*OF equiv-intrel equiv-intrel*])
qed

Congruence property for multiplication

lemma *mult-congruent2*:
(%*p1 p2*. (%(*x*,*y*). (%(*u*,*v*). *intrel*``{(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)}) *p2*) *p1*)
respects2 intrel
apply (*rule equiv-intrel* [*THEN congruent2-commuteI*])
apply (*force simp add*: *mult-ac*, *clarify*)
apply (*simp add*: *congruent-def mult-ac*)
apply (*rename-tac u v w x y z*)
apply (*subgoal-tac u*∗*y* + *x*∗*y* = *w*∗*y* + *v*∗*y* & *u*∗*z* + *x*∗*z* = *w*∗*z* + *v*∗*z*)
apply (*simp add*: *mult-ac*)
apply (*simp add*: *add-mult-distrib* [*symmetric*])
done

lemma *mult*:
Abs-Integ((*intrel*``{(*x*,*y*)})) ∗ *Abs-Integ*((*intrel*``{(*u*,*v*)})) =
Abs-Integ(*intrel* `` {(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)})
by (*simp add*: *mult-int-def UN-UN-split-split-eq mult-congruent2*
UN-equiv-class2 [*OF equiv-intrel equiv-intrel*])

The integers form a *comm-ring-1*

instance *int* :: *comm-ring-1*
proof
fix *i j k* :: *int*
show (*i* + *j*) + *k* = *i* + (*j* + *k*)
by (*cases i*, *cases j*, *cases k*) (*simp add*: *add add-assoc*)
show *i* + *j* = *j* + *i*
by (*cases i*, *cases j*) (*simp add*: *add-ac add*)
show *0* + *i* = *i*
by (*cases i*) (*simp add*: *Zero-int-def add*)
show − *i* + *i* = *0*
by (*cases i*) (*simp add*: *Zero-int-def minus add*)
show *i* − *j* = *i* + − *j*
by (*simp add*: *diff-int-def*)
show (*i* ∗ *j*) ∗ *k* = *i* ∗ (*j* ∗ *k*)
by (*cases i*, *cases j*, *cases k*) (*simp add*: *mult algebra-simps*)

show $i * j = j * i$
by (*cases i*, *cases j*) (*simp add*: *mult algebra-simps*)
show $1 * i = i$
by (*cases i*) (*simp add*: *One-int-def mult*)
show $(i + j) * k = i * k + j * k$
by (*cases i*, *cases j*, *cases k*) (*simp add*: *add mult algebra-simps*)
show $0 \neq (1::int)$
by (*simp add*: *Zero-int-def One-int-def*)
qed

lemma *int-def*: *of-nat m* = *Abs-Integ* (*intrel* `` {(*m*, *0*)})
by (*induct m*, *simp-all add*: *Zero-int-def One-int-def add*)

30.4 The $\leq$ Ordering

lemma *le*:
(*Abs-Integ*(*intrel*``{(*x*,*y*)}) $\leq$ *Abs-Integ*(*intrel*``{(*u*,*v*)})) = ($x+v \leq u+y$)
by (*force simp add*: *le-int-def*)

lemma *less*:
(*Abs-Integ*(*intrel*``{(*x*,*y*)}) < *Abs-Integ*(*intrel*``{(*u*,*v*)})) = ($x+v < u+y$)
by (*simp add*: *less-int-def le order-less-le*)

instance *int* :: *linorder*
proof
fix *i j k* :: *int*
show *antisym*: $i \leq j \Longrightarrow j \leq i \Longrightarrow i = j$
by (*cases i*, *cases j*) (*simp add*: *le*)
show $(i < j) = (i \leq j \wedge \neg\, j \leq i)$
by (*auto simp add*: *less-int-def dest*: *antisym*)
show $i \leq i$
by (*cases i*) (*simp add*: *le*)
show $i \leq j \Longrightarrow j \leq k \Longrightarrow i \leq k$
by (*cases i*, *cases j*, *cases k*) (*simp add*: *le*)
show $i \leq j \vee j \leq i$
by (*cases i*, *cases j*) (*simp add*: *le linorder-linear*)
qed

instantiation *int* :: *distrib-lattice*
begin

definition
(*inf* :: *int* $\Rightarrow$ *int* $\Rightarrow$ *int*) = *min*

definition
(*sup* :: *int* $\Rightarrow$ *int* $\Rightarrow$ *int*) = *max*

instance
by *intro-classes*

(*auto simp add*: *inf-int-def sup-int-def min-max.sup-inf-distrib1*)

end

instance *int* :: *pordered-cancel-ab-semigroup-add*
proof
  fix *i j k* :: *int*
  show $i \leq j \Longrightarrow k + i \leq k + j$
    by (*cases i*, *cases j*, *cases k*) (*simp add*: *le add*)
qed

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on k¿0

lemma *zmult-zless-mono2-lemma*:
  (*i*::*int*)<*j* ==> 0<*k* ==> *of-nat k* * *i* < *of-nat k* * *j*
apply (*induct k*, *simp*)
apply (*simp add*: *left-distrib*)
apply (*case-tac k=0*)
apply (*simp-all add*: *add-strict-mono*)
done

lemma *zero-le-imp-eq-int*: (0::*int*) ≤ *k* ==> ∃ *n*. *k* = *of-nat n*
apply (*cases k*)
apply (*auto simp add*: *le add int-def Zero-int-def*)
apply (*rule-tac x=x−y* **in** *exI*, *simp*)
done

lemma *zero-less-imp-eq-int*: (0::*int*) < *k* ==> ∃ *n*>0. *k* = *of-nat n*
apply (*cases k*)
apply (*simp add*: *less int-def Zero-int-def*)
apply (*rule-tac x=x−y* **in** *exI*, *simp*)
done

lemma *zmult-zless-mono2*: [| *i*<*j*; (0::*int*) < *k* |] ==> *k*∗*i* < *k*∗*j*
apply (*drule zero-less-imp-eq-int*)
apply (*auto simp add*: *zmult-zless-mono2-lemma*)
done

The integers form an ordered integral domain

instance *int* :: *ordered-idom*
proof
  fix *i j k* :: *int*
  show $i < j \Longrightarrow 0 < k \Longrightarrow k * i < k * j$
    by (*rule zmult-zless-mono2*)
  show $|i| = (\mathit{if}\ i < 0\ \mathit{then} - i\ \mathit{else}\ i)$
    by (*simp only*: *zabs-def*)
  show *sgn* (*i*::*int*) = (*if i*=0 *then* 0 *else if* 0<*i then* 1 *else* − 1)
    by (*simp only*: *zsgn-def*)

qed

instance *int* :: *lordered-ring*
proof
  fix *k* :: *int*
  show *abs k* = *sup k* (− *k*)
    by (*auto simp add*: *sup-int-def zabs-def max-def less-minus-self-iff* [*symmetric*])
qed

lemma *zless-imp-add1-zle*: *w* < *z* ⟹ *w* + (*1*::*int*) ≤ *z*
apply (*cases w*, *cases z*)
apply (*simp add*: *less le add One-int-def*)
done

lemma *zless-iff-Suc-zadd*:
  (*w* :: *int*) < *z* ⟷ (∃ *n*. *z* = *w* + *of-nat* (*Suc n*))
apply (*cases z*, *cases w*)
apply (*auto simp add*: *less add int-def*)
apply (*rename-tac a b c d*)
apply (*rule-tac x*=*a*+*d* − *Suc*(*c*+*b*) **in** *exI*)
apply *arith*
done

lemmas *int-distrib* =
  left-distrib [*of z1*::*int z2 w*, *standard*]
  right-distrib [*of w*::*int z1 z2*, *standard*]
  left-diff-distrib [*of z1*::*int z2 w*, *standard*]
  right-diff-distrib [*of w*::*int z1 z2*, *standard*]

30.5 Embedding of the Integers into any *ring-1*: *of-int*

context *ring-1*
begin

definition
  of-int :: *int* ⇒ *'a*
where
  [*code del*]: *of-int z* = *contents* (⋃(*i*, *j*) ∈ *Rep-Integ z*. { *of-nat i* − *of-nat j* })

lemma *of-int*: *of-int* (*Abs-Integ* (*intrel* `` {(*i*,*j*)})) = *of-nat i* − *of-nat j*
proof −
  have (λ(*i*,*j*). { *of-nat i* − (*of-nat j* :: *'a*) }) *respects intrel*
    by (*simp add*: *congruent-def algebra-simps of-nat-add* [*symmetric*]
        del: *of-nat-add*)
  thus *?thesis*
    by (*simp add*: *of-int-def UN-equiv-class* [*OF equiv-intrel*])
qed

lemma *of-int-0* [*simp*]: *of-int 0* = *0*

by (*simp add*: *of-int Zero-int-def*)

lemma *of-int-1* [*simp*]: *of-int 1 = 1*
by (*simp add*: *of-int One-int-def*)

lemma *of-int-add* [*simp*]: *of-int* (*w*+*z*) = *of-int w* + *of-int z*
by (*cases w*, *cases z*, *simp add*: *algebra-simps of-int add*)

lemma *of-int-minus* [*simp*]: *of-int* (−*z*) = − (*of-int z*)
by (*cases z*, *simp add*: *algebra-simps of-int minus*)

lemma *of-int-diff* [*simp*]: *of-int* (*w* − *z*) = *of-int w* − *of-int z*
by (*simp add*: *OrderedGroup.diff-minus diff-minus*)

lemma *of-int-mult* [*simp*]: *of-int* (*w*∗*z*) = *of-int w* ∗ *of-int z*
apply (*cases w*, *cases z*)
apply (*simp add*: *algebra-simps of-int mult of-nat-mult*)
done

Collapse nested embeddings

lemma *of-int-of-nat-eq* [*simp*]: *of-int* (*of-nat n*) = *of-nat n*
by (*induct n*) *auto*

end

context *ordered-idom*
begin

lemma *of-int-le-iff* [*simp*]:
of-int w ≤ *of-int z* ⟷ *w* ≤ *z*
by (*cases w*, *cases z*, *simp add*: *of-int le minus algebra-simps of-nat-add* [*symmetric*] *del*: *of-nat-add*)

Special cases where either operand is zero

lemmas *of-int-0-le-iff* [*simp*] = *of-int-le-iff* [*of 0*, *simplified*]
lemmas *of-int-le-0-iff* [*simp*] = *of-int-le-iff* [*of - 0*, *simplified*]

lemma *of-int-less-iff* [*simp*]:
of-int w < *of-int z* ⟷ *w* < *z*
by (*simp add*: *not-le* [*symmetric*] *linorder-not-le* [*symmetric*])

Special cases where either operand is zero

lemmas *of-int-0-less-iff* [*simp*] = *of-int-less-iff* [*of 0*, *simplified*]
lemmas *of-int-less-0-iff* [*simp*] = *of-int-less-iff* [*of - 0*, *simplified*]

end

Class for unital rings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *ring-char-0* = *ring-1* + *semiring-char-0*
begin

lemma *of-int-eq-iff* [*simp*]:
  of-int w = *of-int z* ⟷ *w* = *z*
apply (*cases w*, *cases z*, *simp add*: *of-int*)
apply (*simp only*: *diff-eq-eq diff-add-eq eq-diff-eq*)
apply (*simp only*: *of-nat-add* [*symmetric*] *of-nat-eq-iff*)
done

Special cases where either operand is zero

lemmas *of-int-0-eq-iff* [*simp*] = *of-int-eq-iff* [*of 0*, *simplified*]
lemmas *of-int-eq-0-iff* [*simp*] = *of-int-eq-iff* [*of - 0*, *simplified*]

end

Every *ordered-idom* has characteristic zero.

subclass (**in** *ordered-idom*) *ring-char-0* **by** *intro-locales*

lemma *of-int-eq-id* [*simp*]: *of-int* = *id*
proof
  fix *z* **show** *of-int z* = *id z*
    by (*cases z*) (*simp add*: *of-int add minus int-def diff-minus*)
qed

30.6 Magnitude of an Integer, as a Natural Number: *nat*

definition
  nat :: *int* ⇒ *nat*
where
  [*code del*]: *nat z* = *contents* (⋃(*x*, *y*) ∈ *Rep-Integ z*. {*x*−*y*})

lemma *nat*: *nat* (*Abs-Integ* (*intrel*``{(*x*,*y*)})) = *x*−*y*
proof −
  have (λ(*x*,*y*). {*x*−*y*}) *respects intrel*
    by (*simp add*: *congruent-def*) *arith*
  thus *?thesis*
    by (*simp add*: *nat-def UN-equiv-class* [*OF equiv-intrel*])
qed

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

lemma *nat-zero* [*simp*]: *nat 0* = *0*
by (*simp add*: *Zero-int-def nat*)

lemma *int-nat-eq* [*simp*]: *of-nat* (*nat z*) = (*if 0* ≤ *z then z else 0*)
by (*cases z*, *simp add*: *nat le int-def Zero-int-def*)

corollary *nat-0-le*: $0 \leq z \Longrightarrow$ *of-nat* (*nat* z) $= z$
by *simp*

lemma *nat-le-0* [*simp*]: $z \leq 0 \Longrightarrow$ *nat* $z = 0$
by (*cases* z, *simp add*: *nat le Zero-int-def*)

lemma *nat-le-eq-zle*: $0 < w \mid 0 \leq z \Longrightarrow$ (*nat* $w \leq$ *nat* z) $= (w \leq z)$
apply (*cases* w, *cases* z)
apply (*simp add*: *nat le linorder-not-le* [*symmetric*] *Zero-int-def*, *arith*)
done

An alternative condition is $(0::'a) \leq w$

corollary *nat-mono-iff*: $0 < z \Longrightarrow$ (*nat* $w <$ *nat* z) $= (w < z)$
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

corollary *nat-less-eq-zless*: $0 \leq w \Longrightarrow$ (*nat* $w <$ *nat* z) $= (w < z)$
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

lemma *zless-nat-conj* [*simp*]: (*nat* $w <$ *nat* z) $= (0 < z \ \&\ w < z)$
apply (*cases* w, *cases* z)
apply (*simp add*: *nat le Zero-int-def linorder-not-le* [*symmetric*], *arith*)
done

lemma *nonneg-eq-int*:
fixes z :: *int*
assumes $0 \leq z$ **and** $\bigwedge m.\ z =$ *of-nat* $m \Longrightarrow P$
shows P
using *assms* **by** (*blast dest*: *nat-0-le sym*)

lemma *nat-eq-iff*: (*nat* $w = m$) $=$ (*if* $0 \leq w$ *then* $w =$ *of-nat* m *else* $m=0$)
by (*cases* w, *simp add*: *nat le int-def Zero-int-def*, *arith*)

corollary *nat-eq-iff2*: ($m =$ *nat* w) $=$ (*if* $0 \leq w$ *then* $w =$ *of-nat* m *else* $m=0$)
by (*simp only*: *eq-commute* [*of* m] *nat-eq-iff*)

lemma *nat-less-iff*: $0 \leq w \Longrightarrow$ (*nat* $w < m$) $= (w <$ *of-nat* $m)$
apply (*cases* w)
apply (*simp add*: *nat le int-def Zero-int-def linorder-not-le*[*symmetric*], *arith*)
done

lemma *nat-0-iff*[*simp*]: *nat*(i::*int*) $= 0 \longleftrightarrow i \leq 0$
by(*simp add*: *nat-eq-iff*) *arith*

lemma *int-eq-iff*: (*of-nat* $m = z$) $= (m =$ *nat* $z \ \&\ 0 \leq z)$
by (*auto simp add*: *nat-eq-iff2*)

lemma *zero-less-nat-eq* [*simp*]: $(0 <$ *nat* $z) = (0 < z)$
by (*insert zless-nat-conj* [*of* 0], *auto*)

lemma *nat-add-distrib*:
[| (0::*int*) ≤ *z*; 0 ≤ *z'* |] ==> *nat* (*z*+*z'*) = *nat z* + *nat z'*
by (*cases z*, *cases z'*, *simp add*: *nat add le Zero-int-def*)

lemma *nat-diff-distrib*:
[| (0::*int*) ≤ *z'*; *z'* ≤ *z* |] ==> *nat* (*z*−*z'*) = *nat z* − *nat z'*
by (*cases z*, *cases z'*,
simp add: *nat add minus diff-minus le Zero-int-def*)

lemma *nat-zminus-int* [*simp*]: *nat* (− (*of-nat n*)) = 0
by (*simp add*: *int-def minus nat Zero-int-def*)

lemma *zless-nat-eq-int-zless*: (*m* < *nat z*) = (*of-nat m* < *z*)
by (*cases z*, *simp add*: *nat less int-def*, *arith*)

context *ring-1*
begin

lemma *of-nat-nat*: 0 ≤ *z* ⟹ *of-nat* (*nat z*) = *of-int z*
by (*cases z rule*: *eq-Abs-Integ*)
(*simp add*: *nat le of-int Zero-int-def of-nat-diff*)

end

For termination proofs:

lemma *measure-function-int*[*measure-function*]: *is-measure* (*nat o abs*) **..**

30.7 Lemmas about the Function *of-nat* and Orderings

lemma *negative-zless-0*: − (*of-nat* (*Suc n*)) < (0 :: *int*)
by (*simp add*: *order-less-le del*: *of-nat-Suc*)

lemma *negative-zless* [*iff*]: − (*of-nat* (*Suc n*)) < (*of-nat m* :: *int*)
by (*rule negative-zless-0* [*THEN order-less-le-trans*], *simp*)

lemma *negative-zle-0*: − *of-nat n* ≤ (0 :: *int*)
by (*simp add*: *minus-le-iff*)

lemma *negative-zle* [*iff*]: − *of-nat n* ≤ (*of-nat m* :: *int*)
by (*rule order-trans* [*OF negative-zle-0 of-nat-0-le-iff*])

lemma *not-zle-0-negative* [*simp*]: ~ (0 ≤ − (*of-nat* (*Suc n*) :: *int*))
by (*subst le-minus-iff*, *simp del*: *of-nat-Suc*)

lemma *int-zle-neg*: ((*of-nat n* :: *int*) ≤ − *of-nat m*) = (*n* = 0 & *m* = 0)
by (*simp add*: *int-def le minus Zero-int-def*)

lemma *not-int-zless-negative* [*simp*]: ~ ((*of-nat n* :: *int*) < − *of-nat m*)
by (*simp add*: *linorder-not-less*)

lemma *negative-eq-positive* [*simp*]: $((-\ of\text{-}nat\ n :: int) = of\text{-}nat\ m) = (n = 0\ \&\ m = 0)$
by (*force simp add*: *order-eq-iff* [*of* $-$ *of-nat n*] *int-zle-neg*)

lemma *zle-iff-zadd*: $(w::int) \leq z \longleftrightarrow (\exists n.\ z = w + of\text{-}nat\ n)$
proof $-$
  have $(w \leq z) = (0 \leq z - w)$
    by (*simp only*: *le-diff-eq add-0-left*)
  also have $\ldots = (\exists n.\ z - w = of\text{-}nat\ n)$
    by (*auto elim*: *zero-le-imp-eq-int*)
  also have $\ldots = (\exists n.\ z = w + of\text{-}nat\ n)$
    by (*simp only*: *algebra-simps*)
  finally show *?thesis* **.**
qed

lemma *zadd-int-left*: *of-nat m* + (*of-nat n* + *z*) = *of-nat* (*m* + *n*) + (*z*::*int*)
by *simp*

lemma *int-Suc0-eq-1*: *of-nat* (*Suc 0*) = (*1*::*int*)
by *simp*

This version is proved for all ordered rings, not just integers! It is proved here because attribute *arith-split* is not available in theory *Ring-and-Field*. But is it really better than just rewriting with *abs-if*?

lemma *abs-split* [*arith-split*,*noatp*]:
    P(*abs*(*a*::'*a*::*ordered-idom*)) = ((*0* ≤ *a* −−> *P a*) & (*a* < *0* −−> *P*(−*a*)))
by (*force dest*: *order-less-le-trans simp add*: *abs-if linorder-not-less*)

lemma *negD*: $(x :: int) < 0 \Longrightarrow \exists n.\ x = -\ (of\text{-}nat\ (Suc\ n))$
apply (*cases x*)
apply (*auto simp add*: *le minus Zero-int-def int-def order-less-le*)
apply (*rule-tac x*=*y* − *Suc x* **in** *exI*, *arith*)
done

30.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

theorem *int-cases* [*cases type*: *int*, *case-names nonneg neg*]:
  [|!! *n*. (*z* :: *int*) = *of-nat n* ==> *P*; !! *n*. *z* = − (*of-nat* (*Suc n*)) ==> *P* |] ==> *P*
apply (*cases z* < *0*, *blast dest*!: *negD*)
apply (*simp add*: *linorder-not-less del*: *of-nat-Suc*)
apply *auto*
apply (*blast dest*: *nat-0-le* [*THEN sym*])
done

theorem *int-induct* [*induct type*: *int*, *case-names nonneg neg*]:
[|!! *n*. *P* (*of-nat n* :: *int*); !!*n*. *P* (− (*of-nat* (*Suc n*))) |] ==> *P z*
by (*cases z rule*: *int-cases*) *auto*

Contributed by Brian Huffman

theorem *int-diff-cases*:
obtains (*diff*) *m n* **where** (*z*::*int*) = *of-nat m* − *of-nat n*
apply (*cases z rule*: *eq-Abs-Integ*)
apply (*rule-tac m*=*x* **and** *n*=*y* **in** *diff*)
apply (*simp add*: *int-def diff-def minus add*)
done

30.9 Binary representation

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Tools/twos−compl.ML*, function *int-of-binary*, for the numerical interpretation.

The representation expects that (*m mod 2*) is 0 or 1, even if m is negative; For instance, *−5 div 2* = *−3* and *−5 mod 2* = *1*; thus *−5* = (*−3*)*2 + *1*.

This two's complement binary representation derives from the paper "An Efficient Representation of Arithmetic for Term Rewriting" by Dave Cohen and Phil Watson, Rewriting Techniques and Applications, Springer LNCS 488 (240-251), 1991.

30.9.1 The constructors *Bit0*, *Bit1*, *Pls* and *Min*

definition
Pls :: *int* **where**
[*code del*]: *Pls* = *0*

definition
Min :: *int* **where**
[*code del*]: *Min* = − *1*

definition
Bit0 :: *int* ⇒ *int* **where**
[*code del*]: *Bit0 k* = *k* + *k*

definition
Bit1 :: *int* ⇒ *int* **where**
[*code del*]: *Bit1 k* = *1* + *k* + *k*

class *number* = — for numeric types: nat, int, real, ...
fixes *number-of* :: *int* ⇒ *'a*

use *Tools/numeral.ML*

syntax
-Numeral :: *num-const* ⇒ *'a* (-)

use *Tools/numeral-syntax.ML*
setup *NumeralSyntax.setup*

abbreviation
Numeral0 ≡ *number-of Pls*

abbreviation
Numeral1 ≡ *number-of* (*Bit1 Pls*)

lemma *Let-number-of* [*simp*]: *Let* (*number-of v*) *f* = *f* (*number-of v*)
— Unfold all *let*s involving constants
unfolding *Let-def* **..**

definition
succ :: *int* ⇒ *int* **where**
[*code del*]: *succ k* = *k* + *1*

definition
pred :: *int* ⇒ *int* **where**
[*code del*]: *pred k* = *k* − *1*

lemmas
max-number-of [*simp*] = *max-def*
[*of number-of u number-of v*, *standard*, *simp*]
and
min-number-of [*simp*] = *min-def*
[*of number-of u number-of v*, *standard*, *simp*]
— unfolding *minx* and *max* on numerals

lemmas *numeral-simps* =
succ-def pred-def Pls-def Min-def Bit0-def Bit1-def

Removal of leading zeroes

lemma *Bit0-Pls* [*simp*, *code post*]:
Bit0 Pls = *Pls*
unfolding *numeral-simps* **by** *simp*

lemma *Bit1-Min* [*simp*, *code post*]:
Bit1 Min = *Min*
unfolding *numeral-simps* **by** *simp*

lemmas *normalize-bin-simps* =
Bit0-Pls Bit1-Min

30.9.2 Successor and predecessor functions

Successor

lemma *succ-Pls*:
succ Pls = Bit1 Pls
unfolding *numeral-simps* **by** *simp*

lemma *succ-Min*:
succ Min = Pls
unfolding *numeral-simps* **by** *simp*

lemma *succ-Bit0*:
succ (Bit0 k) = Bit1 k
unfolding *numeral-simps* **by** *simp*

lemma *succ-Bit1*:
succ (Bit1 k) = Bit0 (succ k)
unfolding *numeral-simps* **by** *simp*

lemmas *succ-bin-simps* [*simp*] =
succ-Pls succ-Min succ-Bit0 succ-Bit1

Predecessor

lemma *pred-Pls*:
pred Pls = Min
unfolding *numeral-simps* **by** *simp*

lemma *pred-Min*:
pred Min = Bit0 Min
unfolding *numeral-simps* **by** *simp*

lemma *pred-Bit0*:
pred (Bit0 k) = Bit1 (pred k)
unfolding *numeral-simps* **by** *simp*

lemma *pred-Bit1*:
pred (Bit1 k) = Bit0 k
unfolding *numeral-simps* **by** *simp*

lemmas *pred-bin-simps* [*simp*] =
pred-Pls pred-Min pred-Bit0 pred-Bit1

30.9.3 Binary arithmetic

Addition

lemma *add-Pls*:
Pls + k = k
unfolding *numeral-simps* **by** *simp*

lemma *add-Min*:
Min + k = pred k
unfolding *numeral-simps* **by** *simp*

lemma *add-Bit0-Bit0*:
(Bit0 k) + (Bit0 l) = Bit0 (k + l)
unfolding *numeral-simps* **by** *simp*

lemma *add-Bit0-Bit1*:
(Bit0 k) + (Bit1 l) = Bit1 (k + l)
unfolding *numeral-simps* **by** *simp*

lemma *add-Bit1-Bit0*:
(Bit1 k) + (Bit0 l) = Bit1 (k + l)
unfolding *numeral-simps* **by** *simp*

lemma *add-Bit1-Bit1*:
(Bit1 k) + (Bit1 l) = Bit0 (k + succ l)
unfolding *numeral-simps* **by** *simp*

lemma *add-Pls-right*:
k + Pls = k
unfolding *numeral-simps* **by** *simp*

lemma *add-Min-right*:
k + Min = pred k
unfolding *numeral-simps* **by** *simp*

lemmas *add-bin-simps* [*simp*] =
add-Pls add-Min add-Pls-right add-Min-right
add-Bit0-Bit0 add-Bit0-Bit1 add-Bit1-Bit0 add-Bit1-Bit1

Negation

lemma *minus-Pls*:
− Pls = Pls
unfolding *numeral-simps* **by** *simp*

lemma *minus-Min*:
− Min = Bit1 Pls
unfolding *numeral-simps* **by** *simp*

lemma *minus-Bit0*:
− (Bit0 k) = Bit0 (− k)
unfolding *numeral-simps* **by** *simp*

lemma *minus-Bit1*:
− (Bit1 k) = Bit1 (pred (− k))
unfolding *numeral-simps* **by** *simp*

lemmas *minus-bin-simps* [*simp*] =
minus-Pls minus-Min minus-Bit0 minus-Bit1

Subtraction

lemma *diff-bin-simps* [*simp*]:
$k - Pls = k$
$k - Min = succ\ k$
$Pls - (Bit0\ l) = Bit0\ (Pls - l)$
$Pls - (Bit1\ l) = Bit1\ (Min - l)$
$Min - (Bit0\ l) = Bit1\ (Min - l)$
$Min - (Bit1\ l) = Bit0\ (Min - l)$
$(Bit0\ k) - (Bit0\ l) = Bit0\ (k - l)$
$(Bit0\ k) - (Bit1\ l) = Bit1\ (pred\ k - l)$
$(Bit1\ k) - (Bit0\ l) = Bit1\ (k - l)$
$(Bit1\ k) - (Bit1\ l) = Bit0\ (k - l)$
unfolding *numeral-simps* **by** *simp-all*

Multiplication

lemma *mult-Pls*:
$Pls * w = Pls$
unfolding *numeral-simps* **by** *simp*

lemma *mult-Min*:
$Min * k = - k$
unfolding *numeral-simps* **by** *simp*

lemma *mult-Bit0*:
$(Bit0\ k) * l = Bit0\ (k * l)$
unfolding *numeral-simps int-distrib* **by** *simp*

lemma *mult-Bit1*:
$(Bit1\ k) * l = (Bit0\ (k * l)) + l$
unfolding *numeral-simps int-distrib* **by** *simp*

lemmas *mult-bin-simps* [*simp*] =
mult-Pls mult-Min mult-Bit0 mult-Bit1

30.9.4 Binary comparisons

Preliminaries

lemma *even-less-0-iff*:
$a + a < 0 \longleftrightarrow a < (0::'a::ordered\text{-}idom)$
proof −
have $a + a < 0 \longleftrightarrow (1+1)*a < 0$ **by** (*simp add*: *left-distrib*)
also have $(1+1)*a < 0 \longleftrightarrow a < 0$
by (*simp add*: *mult-less-0-iff zero-less-two*
order-less-not-sym [*OF zero-less-two*])

finally show *?thesis* **.**
qed

lemma *le-imp-0-less*:
assumes *le*: $0 \leq z$
shows $(0::int) < 1 + z$
proof −
have $0 \leq z$ **by** *fact*
also have $... < z + 1$ **by** (*rule less-add-one*)
also have $... = 1 + z$ **by** (*simp add*: *add-ac*)
finally show $0 < 1 + z$ **.**
qed

lemma *odd-less-0-iff*:
$(1 + z + z < 0) = (z < (0::int))$
proof (*cases z rule*: *int-cases*)
case (*nonneg n*)
thus *?thesis* **by** (*simp add*: *linorder-not-less add-assoc add-increasing*
le-imp-0-less [*THEN order-less-imp-le*])
next
case (*neg n*)
thus *?thesis* **by** (*simp del*: *of-nat-Suc of-nat-add of-nat-1*
add: *algebra-simps of-nat-1* [**where** *'a=int*, *symmetric*] *of-nat-add* [*symmetric*])
qed

lemma *bin-less-0-simps*:
$Pls < 0 \longleftrightarrow False$
$Min < 0 \longleftrightarrow True$
$Bit0\ w < 0 \longleftrightarrow w < 0$
$Bit1\ w < 0 \longleftrightarrow w < 0$
unfolding *numeral-simps*
by (*simp-all add*: *even-less-0-iff odd-less-0-iff*)

lemma *less-bin-lemma*: $k < l \longleftrightarrow k - l < (0::int)$
by *simp*

lemma *le-iff-pred-less*: $k \leq l \longleftrightarrow pred\ k < l$
unfolding *numeral-simps*
proof
have $k - 1 < k$ **by** *simp*
also assume $k \leq l$
finally show $k - 1 < l$ **.**
next
assume $k - 1 < l$
hence $(k - 1) + 1 \leq l$ **by** (*rule zless-imp-add1-zle*)
thus $k \leq l$ **by** *simp*
qed

lemma *succ-pred*: $succ\ (pred\ x) = x$

unfolding *numeral-simps* **by** *simp*

Less-than

lemma *less-bin-simps* [*simp*]:
Pls < *Pls* ⟷ *False*
Pls < *Min* ⟷ *False*
Pls < *Bit0 k* ⟷ *Pls* < *k*
Pls < *Bit1 k* ⟷ *Pls* ≤ *k*
Min < *Pls* ⟷ *True*
Min < *Min* ⟷ *False*
Min < *Bit0 k* ⟷ *Min* < *k*
Min < *Bit1 k* ⟷ *Min* < *k*
Bit0 k < *Pls* ⟷ *k* < *Pls*
Bit0 k < *Min* ⟷ *k* ≤ *Min*
Bit1 k < *Pls* ⟷ *k* < *Pls*
Bit1 k < *Min* ⟷ *k* < *Min*
Bit0 k < *Bit0 l* ⟷ *k* < *l*
Bit0 k < *Bit1 l* ⟷ *k* ≤ *l*
Bit1 k < *Bit0 l* ⟷ *k* < *l*
Bit1 k < *Bit1 l* ⟷ *k* < *l*
unfolding *le-iff-pred-less*
less-bin-lemma [*of Pls*]
less-bin-lemma [*of Min*]
less-bin-lemma [*of k*]
less-bin-lemma [*of Bit0 k*]
less-bin-lemma [*of Bit1 k*]
less-bin-lemma [*of pred Pls*]
less-bin-lemma [*of pred k*]
by (*simp-all add*: *bin-less-0-simps succ-pred*)

Less-than-or-equal

lemma *le-bin-simps* [*simp*]:
Pls ≤ *Pls* ⟷ *True*
Pls ≤ *Min* ⟷ *False*
Pls ≤ *Bit0 k* ⟷ *Pls* ≤ *k*
Pls ≤ *Bit1 k* ⟷ *Pls* ≤ *k*
Min ≤ *Pls* ⟷ *True*
Min ≤ *Min* ⟷ *True*
Min ≤ *Bit0 k* ⟷ *Min* < *k*
Min ≤ *Bit1 k* ⟷ *Min* ≤ *k*
Bit0 k ≤ *Pls* ⟷ *k* ≤ *Pls*
Bit0 k ≤ *Min* ⟷ *k* ≤ *Min*
Bit1 k ≤ *Pls* ⟷ *k* < *Pls*
Bit1 k ≤ *Min* ⟷ *k* ≤ *Min*
Bit0 k ≤ *Bit0 l* ⟷ *k* ≤ *l*
Bit0 k ≤ *Bit1 l* ⟷ *k* ≤ *l*
Bit1 k ≤ *Bit0 l* ⟷ *k* < *l*
Bit1 k ≤ *Bit1 l* ⟷ *k* ≤ *l*
unfolding *not-less* [*symmetric*]

by (*simp-all add*: *not-le*)

Equality

lemma *eq-bin-simps* [*simp*]:
Pls = *Pls* ⟷ *True*
Pls = *Min* ⟷ *False*
Pls = *Bit0 l* ⟷ *Pls* = *l*
Pls = *Bit1 l* ⟷ *False*
Min = *Pls* ⟷ *False*
Min = *Min* ⟷ *True*
Min = *Bit0 l* ⟷ *False*
Min = *Bit1 l* ⟷ *Min* = *l*
Bit0 k = *Pls* ⟷ *k* = *Pls*
Bit0 k = *Min* ⟷ *False*
Bit1 k = *Pls* ⟷ *False*
Bit1 k = *Min* ⟷ *k* = *Min*
Bit0 k = *Bit0 l* ⟷ *k* = *l*
Bit0 k = *Bit1 l* ⟷ *False*
Bit1 k = *Bit0 l* ⟷ *False*
Bit1 k = *Bit1 l* ⟷ *k* = *l*
unfolding *order-eq-iff* [**where** *'a*=*int*]
by (*simp-all add*: *not-less*)

30.10 Converting Numerals to Rings: *number-of*

class *number-ring* = *number* + *comm-ring-1* +
assumes *number-of-eq*: *number-of k* = *of-int k*

self-embedding of the integers

instantiation *int* :: *number-ring*
begin

definition *int-number-of-def* [*code del*]:
number-of w = (*of-int w* :: *int*)

instance proof
qed (*simp only*: *int-number-of-def*)

end

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

lemma *number-of-succ*:
number-of (*succ k*) = (*1* + *number-of k* ::*'a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-pred*:

number-of (*pred w*) = (− *1* + *number-of w* ::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-minus*:
number-of (*uminus w*) = (− (*number-of w*)::'*a*::*number-ring*)
unfolding *number-of-eq* **by** (*rule of-int-minus*)

lemma *number-of-add*:
number-of (*v* + *w*) = (*number-of v* + *number-of w*::'*a*::*number-ring*)
unfolding *number-of-eq* **by** (*rule of-int-add*)

lemma *number-of-diff*:
number-of (*v* − *w*) = (*number-of v* − *number-of w*::'*a*::*number-ring*)
unfolding *number-of-eq* **by** (*rule of-int-diff*)

lemma *number-of-mult*:
number-of (*v* ∗ *w*) = (*number-of v* ∗ *number-of w*::'*a*::*number-ring*)
unfolding *number-of-eq* **by** (*rule of-int-mult*)

The correctness of shifting. But it doesn't seem to give a measurable speed-up.

lemma *double-number-of-Bit0*:
(*1* + *1*) ∗ *number-of w* = (*number-of* (*Bit0 w*) ::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps left-distrib* **by** *simp*

Converting numerals 0 and 1 to their abstract versions.

lemma *numeral-0-eq-0* [*simp*]:
Numeral0 = (*0*::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *numeral-1-eq-1* [*simp*]:
Numeral1 = (*1*::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

Special-case simplification for small constants.

Unary minus for the abstract constant 1. Cannot be inserted as a simprule until later: it is *number-of-Min* re-oriented!

lemma *numeral-m1-eq-minus-1*:
(−*1*::'*a*::*number-ring*) = − *1*
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *mult-minus1* [*simp*]:
−*1* ∗ *z* = −(*z*::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *mult-minus1-right* [*simp*]:
z ∗ −*1* = −(*z*::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *minus-number-of-mult* [*simp*]:
  $-$ (*number-of w*) $*$ *z* = *number-of* (*uminus w*) $*$ (*z*::'*a*::*number-ring*)
  unfolding *number-of-eq* **by** *simp*

Subtraction

lemma *diff-number-of-eq*:
  number-of v $-$ *number-of w* =
    (*number-of* (*v* + *uminus w*)::'*a*::*number-ring*)
  unfolding *number-of-eq* **by** *simp*

lemma *number-of-Pls*:
  number-of Pls = (*0*::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-Min*:
  number-of Min = ($-$ *1*::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-Bit0*:
  number-of (*Bit0 w*) = (*0*::'*a*::*number-ring*) + (*number-of w*) + (*number-of w*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-Bit1*:
  number-of (*Bit1 w*) = (*1*::'*a*::*number-ring*) + (*number-of w*) + (*number-of w*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

30.10.1 Equality of Binary Numbers

First version by Norbert Voelker

definition
  iszero :: '*a*::*semiring-1* $\Rightarrow$ *bool*
where
  iszero z $\longleftrightarrow$ *z* = *0*

lemma *iszero-0*: *iszero 0*
by (*simp add*: *iszero-def*)

lemma *not-iszero-1*: ~ *iszero 1*
by (*simp add*: *iszero-def eq-commute*)

lemma *eq-number-of-eq*:
  ((*number-of x*::'*a*::*number-ring*) = *number-of y*) =
    iszero (*number-of* (*x* + *uminus y*) :: '*a*)
unfolding *iszero-def number-of-add number-of-minus*
by (*simp add*: *algebra-simps*)

lemma *iszero-number-of-Pls*:

iszero ((*number-of Pls*)::'*a*::*number-ring*)
unfolding *iszero-def numeral-0-eq-0* **..**

lemma *nonzero-number-of-Min*:
~ *iszero* ((*number-of Min*)::'*a*::*number-ring*)
unfolding *iszero-def numeral-m1-eq-minus-1* **by** *simp*

30.10.2 Comparisons, for Ordered Rings

lemmas *double-eq-0-iff* = *double-zero*

lemma *odd-nonzero*:
1 + *z* + *z* ≠ (*0*::*int*)
proof (*cases z rule*: *int-cases*)
case (*nonneg n*)
have *le*: *0* ≤ *z*+*z* **by** (*simp add*: *nonneg add-increasing*)
thus *?thesis* **using** *le-imp-0-less* [*OF le*]
by (*auto simp add*: *add-assoc*)
next
case (*neg n*)
show *?thesis*
proof
assume *eq*: *1* + *z* + *z* = *0*
have (*0*::*int*) < *1* + (*of-nat n* + *of-nat n*)
by (*simp add*: *le-imp-0-less add-increasing*)
also have ... = − (*1* + *z* + *z*)
by (*simp add*: *neg add-assoc* [*symmetric*])
also have ... = *0* **by** (*simp add*: *eq*)
finally have *0*<*0* **..**
thus *False* **by** *blast*
qed
qed

lemma *iszero-number-of-Bit0*:
iszero (*number-of* (*Bit0 w*)::'*a*) =
iszero (*number-of w*::'*a*::{*ring-char-0*,*number-ring*})
proof −
have (*of-int w* + *of-int w* = (*0*::'*a*)) ⟹ (*w* = *0*)
proof −
assume *eq*: *of-int w* + *of-int w* = (*0*::'*a*)
then have *of-int* (*w* + *w*) = (*of-int 0* :: '*a*) **by** *simp*
then have *w* + *w* = *0* **by** (*simp only*: *of-int-eq-iff*)
then show *w* = *0* **by** (*simp only*: *double-eq-0-iff*)
qed
thus *?thesis*
by (*auto simp add*: *iszero-def number-of-eq numeral-simps*)
qed

lemma *iszero-number-of-Bit1*:

~ *iszero* (*number-of* (*Bit1 w*)::′*a*::{*ring-char-0*,*number-ring*})
proof −
have *1 + of-int w + of-int w* ≠ (*0*::′*a*)
proof
assume *eq*: *1 + of-int w + of-int w* = (*0*::′*a*)
hence *of-int* (*1 + w + w*) = (*of-int 0* :: ′*a*) **by** *simp*
hence *1 + w + w* = *0* **by** (*simp only*: *of-int-eq-iff*)
with *odd-nonzero* **show** *False* **by** *blast*
qed
thus *?thesis*
by (*auto simp add*: *iszero-def number-of-eq numeral-simps*)
qed

lemmas *iszero-simps* =
iszero-0 not-iszero-1
iszero-number-of-Pls nonzero-number-of-Min
iszero-number-of-Bit0 iszero-number-of-Bit1

30.10.3 The Less-Than Relation

lemma *double-less-0-iff*:
(*a + a < 0*) = (*a* < (*0*::′*a*::*ordered-idom*))
proof −
have (*a + a < 0*) = ((*1*+*1*)∗*a < 0*) **by** (*simp add*: *left-distrib*)
also have ... = (*a < 0*)
by (*simp add*: *mult-less-0-iff zero-less-two*
order-less-not-sym [*OF zero-less-two*])
finally show *?thesis* **.**
qed

lemma *odd-less-0*:
(*1 + z + z < 0*) = (*z* < (*0*::*int*))
proof (*cases z rule*: *int-cases*)
case (*nonneg n*)
thus *?thesis* **by** (*simp add*: *linorder-not-less add-assoc add-increasing*
le-imp-0-less [*THEN order-less-imp-le*])
next
case (*neg n*)
thus *?thesis* **by** (*simp del*: *of-nat-Suc of-nat-add of-nat-1*
add: *algebra-simps of-nat-1* [**where** ′*a*=*int*, *symmetric*] *of-nat-add* [*symmetric*])
qed

Less-Than or Equals

Reduces $a \leq b$ to $\neg\ b < a$ for ALL numerals.

lemmas *le-number-of-eq-not-less* =
linorder-not-less [*of number-of w number-of v*, *symmetric*,
standard]

Absolute value (*abs*)

lemma *abs-number-of*:
abs(*number-of x*::'*a*::{*ordered-idom*,*number-ring*}) =
(*if number-of x* < (*0*::'*a*) *then* −*number-of x else number-of x*)
by (*simp add*: *abs-if*)

Re-orientation of the equation nnn=x

lemma *number-of-reorient*:
(*number-of w* = *x*) = (*x* = *number-of w*)
by *auto*

30.10.4 Simplification of arithmetic operations on integer constants.

lemmas *arith-extra-simps* [*standard*, *simp*] =
number-of-add [*symmetric*]
number-of-minus [*symmetric*]
numeral-m1-eq-minus-1 [*symmetric*]
number-of-mult [*symmetric*]
diff-number-of-eq abs-number-of

For making a minimal simpset, one must include these default simprules. Also include *simp-thms*.

lemmas *arith-simps* =
normalize-bin-simps pred-bin-simps succ-bin-simps
add-bin-simps minus-bin-simps mult-bin-simps
abs-zero abs-one arith-extra-simps

Simplification of relational operations

lemma *less-number-of* [*simp*]:
(*number-of x*::'*a*::{*ordered-idom*,*number-ring*}) < *number-of y* ⟷ *x* < *y*
unfolding *number-of-eq* **by** (*rule of-int-less-iff*)

lemma *le-number-of* [*simp*]:
(*number-of x*::'*a*::{*ordered-idom*,*number-ring*}) ≤ *number-of y* ⟷ *x* ≤ *y*
unfolding *number-of-eq* **by** (*rule of-int-le-iff*)

lemma *eq-number-of* [*simp*]:
(*number-of x*::'*a*::{*ring-char-0*,*number-ring*}) = *number-of y* ⟷ *x* = *y*
unfolding *number-of-eq* **by** (*rule of-int-eq-iff*)

lemmas *rel-simps* [*simp*] =
less-number-of less-bin-simps
le-number-of le-bin-simps
eq-number-of-eq eq-bin-simps
iszero-simps

30.10.5 Simplification of arithmetic when nested to the right.

lemma *add-number-of-left* [*simp*]:

number-of v + (number-of w + z) =
(number-of(v + w) + z::'a::number-ring)
by (*simp add*: *add-assoc* [*symmetric*])

lemma *mult-number-of-left* [*simp*]:
*number-of v * (number-of w * z) =*
*(number-of(v * w) * z::'a::number-ring)*
by (*simp add*: *mult-assoc* [*symmetric*])

lemma *add-number-of-diff1*:
number-of v + (number-of w - c) =
number-of(v + w) - (c::'a::number-ring)
by (*simp add*: *diff-minus add-number-of-left*)

lemma *add-number-of-diff2* [*simp*]:
number-of v + (c - number-of w) =
number-of (v + uminus w) + (c::'a::number-ring)
by (*simp add*: *algebra-simps diff-number-of-eq* [*symmetric*])

30.11 The Set of Integers

context *ring-1*
begin

definition *Ints* :: *'a set* **where**
[*code del*]: *Ints = range of-int*

end

notation (*xsymbols*)
Ints (ℤ)

context *ring-1*
begin

lemma *Ints-0* [*simp*]: *0 ∈ ℤ*
apply (*simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-0* [*symmetric*])
done

lemma *Ints-1* [*simp*]: *1 ∈ ℤ*
apply (*simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-1* [*symmetric*])
done

lemma *Ints-add* [*simp*]: *a ∈ ℤ ⟹ b ∈ ℤ ⟹ a + b ∈ ℤ*
apply (*auto simp add*: *Ints-def*)

apply (*rule range-eqI*)
apply (*rule of-int-add* [*symmetric*])
done

lemma *Ints-minus* [*simp*]: $a \in \mathbb{Z} \Longrightarrow -a \in \mathbb{Z}$
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-minus* [*symmetric*])
done

lemma *Ints-mult* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a * b \in \mathbb{Z}$
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-mult* [*symmetric*])
done

lemma *Ints-cases* [*cases set*: *Ints*]:
  assumes $q \in \mathbb{Z}$
  obtains (*of-int*) z **where** $q = of\text{-}int\ z$
  unfolding *Ints-def*
proof −
  from ‹$q \in \mathbb{Z}$› **have** $q \in range\ of\text{-}int$ **unfolding** *Ints-def* **.**
  then obtain z **where** $q = of\text{-}int\ z$ **..**
  then show *thesis* **..**
qed

lemma *Ints-induct* [*case-names of-int*, *induct set*: *Ints*]:
  $q \in \mathbb{Z} \Longrightarrow (\bigwedge z.\ P\ (of\text{-}int\ z)) \Longrightarrow P\ q$
  by (*rule Ints-cases*) *auto*

end

lemma *Ints-diff* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a-b \in \mathbb{Z}$
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-diff* [*symmetric*])
done

The premise involving $\mathbb{Z}$ prevents $a = (1::'a)\ /\ (2::'a)$.

lemma *Ints-double-eq-0-iff*:
  assumes *in-Ints*: $a \in Ints$
  shows $(a + a = 0) = (a = (0::'a::ring\text{-}char\text{-}0))$
proof −
  from *in-Ints* **have** $a \in range\ of\text{-}int$ **unfolding** *Ints-def* [*symmetric*] **.**
  then obtain z **where** *a*: $a = of\text{-}int\ z$ **..**
  show *?thesis*
  proof
    assume $a = 0$
    thus $a + a = 0$ **by** *simp*

```
  next
    assume eq: a + a = 0
    hence of-int (z + z) = (of-int 0 :: 'a) by (simp add: a)
    hence z + z = 0 by (simp only: of-int-eq-iff)
    hence z = 0 by (simp only: double-eq-0-iff)
    thus a = 0 by (simp add: a)
  qed
qed

lemma Ints-odd-nonzero:
  assumes in-Ints: a ∈ Ints
  shows 1 + a + a ≠ (0::'a::ring-char-0)
proof −
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  show ?thesis
  proof
    assume eq: 1 + a + a = 0
    hence of-int (1 + z + z) = (of-int 0 :: 'a) by (simp add: a)
    hence 1 + z + z = 0 by (simp only: of-int-eq-iff)
    with odd-nonzero show False by blast
  qed
qed

lemma Ints-number-of:
  (number-of w :: 'a::number-ring) ∈ Ints
  unfolding number-of-eq Ints-def by simp

lemma Ints-odd-less-0:
  assumes in-Ints: a ∈ Ints
  shows (1 + a + a < 0) = (a < (0::'a::ordered-idom))
proof −
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  hence ((1::'a) + a + a < 0) = (of-int (1 + z + z) < (of-int 0 :: 'a))
    by (simp add: a)
  also have ... = (z < 0) by (simp only: of-int-less-iff odd-less-0)
  also have ... = (a < 0) by (simp add: a)
  finally show ?thesis .
qed
```

30.12 *setsum* and *setprod*

```
lemma of-nat-setsum: of-nat (setsum f A) = (∑x∈A. of-nat(f x))
  apply (cases finite A)
  apply (erule finite-induct, auto)
  done

lemma of-int-setsum: of-int (setsum f A) = (∑x∈A. of-int(f x))
```

apply (*cases finite A*)
apply (*erule finite-induct*, *auto*)
done

lemma *of-nat-setprod*: *of-nat* (*setprod f A*) = ($\prod x{\in}A$. *of-nat*(*f x*))
apply (*cases finite A*)
apply (*erule finite-induct*, *auto simp add*: *of-nat-mult*)
done

lemma *of-int-setprod*: *of-int* (*setprod f A*) = ($\prod x{\in}A$. *of-int*(*f x*))
apply (*cases finite A*)
apply (*erule finite-induct*, *auto*)
done

lemmas *int-setsum* = *of-nat-setsum* [**where** *'a*=*int*]
lemmas *int-setprod* = *of-nat-setprod* [**where** *'a*=*int*]

30.13 Inequality Reasoning for the Arithmetic Simproc

lemma *add-numeral-0*: *Numeral0* + *a* = (*a*::*'a*::*number-ring*)
by *simp*

lemma *add-numeral-0-right*: *a* + *Numeral0* = (*a*::*'a*::*number-ring*)
by *simp*

lemma *mult-numeral-1*: *Numeral1* * *a* = (*a*::*'a*::*number-ring*)
by *simp*

lemma *mult-numeral-1-right*: *a* * *Numeral1* = (*a*::*'a*::*number-ring*)
by *simp*

lemma *divide-numeral-1*: *a* / *Numeral1* = (*a*::*'a*::{*number-ring*,*field*})
by *simp*

lemma *inverse-numeral-1*:
inverse Numeral1 = (*Numeral1*::*'a*::{*number-ring*,*field*})
by *simp*

Theorem lists for the cancellation simprocs. The use of binary numerals for 0 and 1 reduces the number of special cases.

lemmas *add-0s* = *add-numeral-0 add-numeral-0-right*
lemmas *mult-1s* = *mult-numeral-1 mult-numeral-1-right*
mult-minus1 mult-minus1-right

30.14 Special Arithmetic Rules for Abstract 0 and 1

Arithmetic computations are defined for binary literals, which leaves 0 and 1 as special cases. Addition already has rules for 0, but not 1. Multiplication and unary minus already have rules for both 0 and 1.

lemma *binop-eq*: $[|f\ x\ y = g\ x\ y;\ x = x';\ y = y'|] ==> f\ x'\ y' = g\ x'\ y'$
by *simp*

lemmas *add-number-of-eq* = *number-of-add* [*symmetric*]

Allow 1 on either or both sides

lemma *one-add-one-is-two*: *1 + 1 = (2::'a::number-ring)*
by (*simp del*: *numeral-1-eq-1 add*: *numeral-1-eq-1* [*symmetric*] *add-number-of-eq*)

lemmas *add-special* =
one-add-one-is-two
binop-eq [*of op +*, *OF add-number-of-eq numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op +*, *OF add-number-of-eq refl numeral-1-eq-1*, *standard*]

Allow 1 on either or both sides (1-1 already simplifies to 0)

lemmas *diff-special* =
binop-eq [*of op −*, *OF diff-number-of-eq numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op −*, *OF diff-number-of-eq refl numeral-1-eq-1*, *standard*]

Allow 0 or 1 on either side with a binary numeral on the other

lemmas *eq-special* =
binop-eq [*of op =*, *OF eq-number-of-eq numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op =*, *OF eq-number-of-eq numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op =*, *OF eq-number-of-eq refl numeral-0-eq-0*, *standard*]
binop-eq [*of op =*, *OF eq-number-of-eq refl numeral-1-eq-1*, *standard*]

Allow 0 or 1 on either side with a binary numeral on the other

lemmas *less-special* =
binop-eq [*of op <*, *OF less-number-of numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op <*, *OF less-number-of numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op <*, *OF less-number-of refl numeral-0-eq-0*, *standard*]
binop-eq [*of op <*, *OF less-number-of refl numeral-1-eq-1*, *standard*]

Allow 0 or 1 on either side with a binary numeral on the other

lemmas *le-special* =
binop-eq [*of op ≤*, *OF le-number-of numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op ≤*, *OF le-number-of numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op ≤*, *OF le-number-of refl numeral-0-eq-0*, *standard*]
binop-eq [*of op ≤*, *OF le-number-of refl numeral-1-eq-1*, *standard*]

lemmas *arith-special*[*simp*] =
add-special diff-special eq-special less-special le-special

lemma *min-max-01*: *min (0::int) 1 = 0 & min (1::int) 0 = 0 &*
max (0::int) 1 = 1 & max (1::int) 0 = 1
by(*simp add*:*min-def max-def*)

lemmas *min-max-special*[*simp*] =
min-max-01
max-def[*of 0*::*int number-of v*, *standard*, *simp*]
min-def[*of 0*::*int number-of v*, *standard*, *simp*]
max-def[*of number-of u 0*::*int*, *standard*, *simp*]
min-def[*of number-of u 0*::*int*, *standard*, *simp*]
max-def[*of 1*::*int number-of v*, *standard*, *simp*]
min-def[*of 1*::*int number-of v*, *standard*, *simp*]
max-def[*of number-of u 1*::*int*, *standard*, *simp*]
min-def[*of number-of u 1*::*int*, *standard*, *simp*]

Legacy theorems

lemmas *zle-int* = *of-nat-le-iff* [**where** *'a*=*int*]
lemmas *int-int-eq* = *of-nat-eq-iff* [**where** *'a*=*int*]

30.15 Setting up simplification procedures

lemmas *int-arith-rules* =
neg-le-iff-le numeral-0-eq-0 numeral-1-eq-1
minus-zero diff-minus left-minus right-minus
mult-zero-left mult-zero-right mult-Bit1 mult-1-right
mult-minus-left mult-minus-right
minus-add-distrib minus-minus mult-assoc
of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult
of-int-0 of-int-1 of-int-add of-int-mult

use *Tools/int-arith.ML*
declaration ⟪ *K Int-Arith.setup* ⟫

30.16 Lemmas About Small Numerals

lemma *of-int-m1* [*simp*]: *of-int −1* = (*−1* :: *'a* :: *number-ring*)
proof −
have (*of-int −1* :: *'a*) = *of-int* (*− 1*) **by** *simp*
also have ... = *− of-int 1* **by** (*simp only*: *of-int-minus*)
also have ... = *−1* **by** *simp*
finally show *?thesis* **.**
qed

lemma *abs-minus-one* [*simp*]: *abs* (*−1*) = (*1*::*'a*::{*ordered-idom*,*number-ring*})
by (*simp add*: *abs-if*)

lemma *abs-power-minus-one* [*simp*]:
abs(*−1* ^ *n*) = (*1*::*'a*::{*ordered-idom*,*number-ring*,*recpower*})
by (*simp add*: *power-abs*)

lemma *of-int-number-of-eq* [*simp*]:
of-int (*number-of v*) = (*number-of v* :: *'a* :: *number-ring*)

by (*simp add*: *number-of-eq*)

Lemmas for specialist use, NOT as default simprules

lemma *mult-2*: $2 * z = (z+z::'a::number\text{-}ring)$
proof −
  have $2*z = (1 + 1)*z$ **by** *simp*
  also have ... $= z+z$ **by** (*simp add*: *left-distrib*)
  finally show *?thesis* **.**
qed

lemma *mult-2-right*: $z * 2 = (z+z::'a::number\text{-}ring)$
by (*subst mult-commute*, *rule mult-2*)

30.17 More Inequality Reasoning

lemma *zless-add1-eq*: $(w < z + (1::int)) = (w<z \mid w=z)$
by *arith*

lemma *add1-zle-eq*: $(w + (1::int) \leq z) = (w<z)$
by *arith*

lemma *zle-diff1-eq* [*simp*]: $(w \leq z - (1::int)) = (w<z)$
by *arith*

lemma *zle-add1-eq-le* [*simp*]: $(w < z + (1::int)) = (w\leq z)$
by *arith*

lemma *int-one-le-iff-zero-less*: $((1::int) \leq z) = (0 < z)$
by *arith*

30.18 The functions *nat* and *int*

Simplify the terms *int* $(0::'a)$, *int* (*Suc 0*) and $w + - z$

declare *Zero-int-def* [*symmetric*, *simp*]
declare *One-int-def* [*symmetric*, *simp*]

lemmas *diff-int-def-symmetric* = *diff-int-def* [*symmetric*, *simp*]

lemma *nat-0*: *nat 0* = *0*
by (*simp add*: *nat-eq-iff*)

lemma *nat-1*: *nat 1* = *Suc 0*
by (*subst nat-eq-iff*, *simp*)

lemma *nat-2*: *nat 2* = *Suc* (*Suc 0*)
by (*subst nat-eq-iff*, *simp*)

lemma *one-less-nat-eq* [*simp*]: (*Suc 0* < *nat z*) = (*1* < *z*)
apply (*insert zless-nat-conj* [*of 1 z*])

apply (*auto simp add*: *nat-1*)
done

This simplifies expressions of the form *int n = z* where z is an integer literal.

lemmas *int-eq-iff-number-of* [*simp*] = *int-eq-iff* [*of - number-of v*, *standard*]

lemma *split-nat* [*arith-split*]:
  $P(nat(i::int)) = ((\forall n.\ i = of\text{-}nat\ n \longrightarrow P\ n)\ \&\ (i < 0 \longrightarrow P\ 0))$
  (**is** *?P* = (*?L* & *?R*))
proof (*cases i < 0*)
  case *True* **thus** *?thesis* **by** *auto*
next
  case *False*
  have *?P* = *?L*
  proof
    assume *?P* **thus** *?L* **using** *False* **by** *clarsimp*
  next
    assume *?L* **thus** *?P* **using** *False* **by** *simp*
  qed
  with *False* **show** *?thesis* **by** *simp*
qed

context *ring-1*
begin

lemma *of-int-of-nat* [*nitpick-const-simp*]:
  of-int k = (*if k < 0 then* − *of-nat* (*nat* (− *k*)) *else of-nat* (*nat k*))
proof (*cases k < 0*)
  case *True* **then have** *0* ≤ − *k* **by** *simp*
  then have *of-nat* (*nat* (− *k*)) = *of-int* (− *k*) **by** (*rule of-nat-nat*)
  with *True* **show** *?thesis* **by** *simp*
next
  case *False* **then show** *?thesis* **by** (*simp add*: *not-less of-nat-nat*)
qed

end

lemma *nat-mult-distrib*:
  fixes *z z′* :: *int*
  assumes *0* ≤ *z*
  shows *nat* (*z* ∗ *z′*) = *nat z* ∗ *nat z′*
proof (*cases 0* ≤ *z′*)
  case *False* **with** *assms* **have** *z* ∗ *z′* ≤ *0*
    by (*simp add*: *not-le mult-le-0-iff*)
  then have *nat* (*z* ∗ *z′*) = *0* **by** *simp*
  moreover from *False* **have** *nat z′* = *0* **by** *simp*
  ultimately show *?thesis* **by** *simp*
next
  case *True* **with** *assms* **have** *ge-0*: *z* ∗ *z′* ≥ *0* **by** (*simp add*: *zero-le-mult-iff*)

show *?thesis*
by (*rule injD* [*of of-nat* :: *nat* ⇒ *int*, *OF inj-of-nat*])
(*simp only*: *of-nat-mult of-nat-nat* [*OF True*]
of-nat-nat [*OF assms*] *of-nat-nat* [*OF ge-0*], *simp*)
qed

lemma *nat-mult-distrib-neg*: $z \leq (0::int) ==> nat(z*z') = nat(-z) * nat(-z')$
apply (*rule trans*)
apply (*rule-tac* [*2*] *nat-mult-distrib*, *auto*)
done

lemma *nat-abs-mult-distrib*: *nat* (*abs* (*w* * *z*)) = *nat* (*abs w*) * *nat* (*abs z*)
apply (*cases z=0* | *w=0*)
apply (*auto simp add*: *abs-if nat-mult-distrib* [*symmetric*]
nat-mult-distrib-neg [*symmetric*] *mult-less-0-iff*)
done

30.19 Induction principles for int

Well-founded segments of the integers

definition
int-ge-less-than :: *int* => (*int* * *int*) *set*
where
int-ge-less-than d = {(*z'*,*z*). *d* ≤ *z'* & *z'* < *z*}

theorem *wf-int-ge-less-than*: *wf* (*int-ge-less-than d*)
proof −
have *int-ge-less-than d* ⊆ *measure* (%*z*. *nat* (*z*−*d*))
by (*auto simp add*: *int-ge-less-than-def*)
thus *?thesis*
by (*rule wf-subset* [*OF wf-measure*])
qed

This variant looks odd, but is typical of the relations suggested by RankFinder.

definition
int-ge-less-than2 :: *int* => (*int* * *int*) *set*
where
int-ge-less-than2 d = {(*z'*,*z*). *d* ≤ *z* & *z'* < *z*}

theorem *wf-int-ge-less-than2*: *wf* (*int-ge-less-than2 d*)
proof −
have *int-ge-less-than2 d* ⊆ *measure* (%*z*. *nat* (*1*+*z*−*d*))
by (*auto simp add*: *int-ge-less-than2-def*)
thus *?thesis*
by (*rule wf-subset* [*OF wf-measure*])
qed

abbreviation
int :: *nat* ⇒ *int*
where
int ≡ *of-nat*

theorem *int-ge-induct* [*case-names base step*, *induct set*: *int*]:
fixes *i* :: *int*
assumes *ge*: $k \leq i$ **and**
base: *P k* **and**
step: $\bigwedge i.\ k \leq i \Longrightarrow P\ i \Longrightarrow P\ (i + 1)$
shows *P i*
proof −
{ **fix** *n* **have** $\bigwedge i{::}int.\ n = nat(i-k) \Longrightarrow k \leq i \Longrightarrow P\ i$
proof (*induct n*)
case *0*
hence $i = k$ **by** *arith*
thus *P i* **using** *base* **by** *simp*
next
case (*Suc n*)
then have $n = nat((i - 1) - k)$ **by** *arith*
moreover
have *ki1*: $k \leq i - 1$ **using** *Suc.prems* **by** *arith*
ultimately
have $P(i - 1)$ **by**(*rule Suc.hyps*)
from *step*[*OF ki1 this*] **show** *?case* **by** *simp*
qed
}
with *ge* **show** *?thesis* **by** *fast*
qed

theorem *int-gr-induct* [*case-names base step*, *induct set*: *int*]:
assumes *gr*: $k < (i{::}int)$ **and**
base: $P(k+1)$ **and**
step: $\bigwedge i.\ [\![k < i;\ P\ i]\!] \Longrightarrow P(i+1)$
shows *P i*
apply(*rule int-ge-induct*[*of k + 1*])
using *gr* **apply** *arith*
apply(*rule base*)
apply (*rule step*, *simp+*)
done

theorem *int-le-induct*[*consumes 1*,*case-names base step*]:
assumes *le*: $i \leq (k{::}int)$ **and**
base: $P(k)$ **and**
step: $\bigwedge i.\ [\![i \leq k;\ P\ i]\!] \Longrightarrow P(i - 1)$
shows *P i*
proof −

```
  { fix n have ⋀i::int. n = nat(k−i) ⟹ i ≤ k ⟹ P i
    proof (induct n)
      case 0
      hence i = k by arith
      thus P i using base by simp
    next
      case (Suc n)
      hence n = nat(k − (i+1)) by arith
      moreover
      have ki1: i + 1 ≤ k using Suc.prems by arith
      ultimately
      have P(i+1) by(rule Suc.hyps)
      from step[OF ki1 this] show ?case by simp
    qed
  }
  with le show ?thesis by fast
qed

theorem int-less-induct [consumes 1,case-names base step]:
  assumes less: (i::int) < k and
        base: P(k − 1) and
        step: ⋀i. ⟦i < k; P i⟧ ⟹ P(i − 1)
  shows P i
apply(rule int-le-induct[of - k − 1])
  using less apply arith
 apply(rule base)
apply (rule step, simp+)
done
```

30.20 Intermediate value theorems

```
lemma int-val-lemma:
     (∀ i<n::nat. abs(f(i+1) − f i) ≤ 1) −−>
      f 0 ≤ k −−> k ≤ f n −−> (∃ i ≤ n. f i = (k::int))
unfolding One-nat-def
apply (induct n, simp)
apply (intro strip)
apply (erule impE, simp)
apply (erule-tac x = n in allE, simp)
apply (case-tac k = f (Suc n))
apply force
apply (erule impE)
 apply (simp add: abs-if split add: split-if-asm)
apply (blast intro: le-SucI)
done

lemmas nat0-intermed-int-val = int-val-lemma [rule-format (no-asm)]

lemma nat-intermed-int-val:
```

```
    [| ∀i. m ≤ i & i < n --> abs(f(i + 1::nat) - f i) ≤ 1; m < n;
       f m ≤ k; k ≤ f n |] ==> ? i. m ≤ i & i ≤ n & f i = (k::int)
apply (cut-tac n = n-m and f = %i. f (i+m)  and k = k
      in int-val-lemma)
unfolding One-nat-def
apply simp
apply (erule exE)
apply (rule-tac x = i+m in exI, arith)
done
```

30.21 Products and 1, by T. M. Rasmussen

```
lemma zabs-less-one-iff [simp]: (|z| < 1) = (z = (0::int))
by arith

lemma abs-zmult-eq-1: (|m * n| = 1) ==> |m| = (1::int)
apply (cases |n|=1)
apply (simp add: abs-mult)
apply (rule ccontr)
apply (auto simp add: linorder-neq-iff abs-mult)
apply (subgoal-tac 2 ≤ |m| & 2 ≤ |n|)
 prefer 2 apply arith
apply (subgoal-tac 2*2 ≤ |m| * |n|, simp)
apply (rule mult-mono, auto)
done

lemma pos-zmult-eq-1-iff-lemma: (m * n = 1) ==> m = (1::int) | m = -1
by (insert abs-zmult-eq-1 [of m n], arith)

lemma pos-zmult-eq-1-iff: 0 < (m::int) ==> (m * n = 1) = (m = 1 & n = 1)
apply (auto dest: pos-zmult-eq-1-iff-lemma)
apply (simp add: mult-commute [of m])
apply (frule pos-zmult-eq-1-iff-lemma, auto)
done

lemma zmult-eq-1-iff: (m*n = (1::int)) = ((m = 1 & n = 1) | (m = -1 & n =
-1))
apply (rule iffI)
 apply (frule pos-zmult-eq-1-iff-lemma)
 apply (simp add: mult-commute [of m])
 apply (frule pos-zmult-eq-1-iff-lemma, auto)
done


lemma infinite-UNIV-int: ~finite(UNIV::int set)
proof
  assume finite(UNIV::int set)
  moreover have ~(EX i::int. 2*i = 1)
    by (auto simp: pos-zmult-eq-1-iff)
```

ultimately show *False* **using** *finite-UNIV-inj-surj*[*of* %*n*::*int*. *n*+*n*]
by (*simp add*:*inj-on-def surj-def*) (*blast intro*:*sym*)
qed

30.22 Integer Powers

instantiation *int* :: *recpower*
begin

primrec *power-int* **where**
p ^ *0* = (*1*::*int*)
| *p* ^ (*Suc n*) = (*p*::*int*) * (*p* ^ *n*)

instance proof
fix *z* :: *int*
fix *n* :: *nat*
show *z* ^ *0* = *1* **by** *simp*
show *z* ^ *Suc n* = *z* * (*z* ^ *n*) **by** *simp*
qed

declare *power-int.simps* [*simp del*]

end

lemma *zpower-zadd-distrib*: *x* ^ (*y* + *z*) = ((*x* ^ *y*) * (*x* ^ *z*)::*int*)
by (*rule Power.power-add*)

lemma *zpower-zpower*: (*x* ^ *y*) ^ *z* = (*x* ^ (*y* * *z*)::*int*)
by (*rule Power.power-mult* [*symmetric*])

lemma *zero-less-zpower-abs-iff* [*simp*]:
(*0* < *abs x* ^ *n*) ⟷ (*x* ≠ (*0*::*int*) | *n* = *0*)
by (*induct n*) (*auto simp add*: *zero-less-mult-iff*)

lemma *zero-le-zpower-abs* [*simp*]: (*0*::*int*) ≤ *abs x* ^ *n*
by (*induct n*) (*auto simp add*: *zero-le-mult-iff*)

lemma *of-int-power*:
of-int (*z* ^ *n*) = (*of-int z* ^ *n* :: *'a*::{*recpower*, *ring-1*})
by (*induct n*) *simp-all*

lemma *int-power*: *int* (*m*^*n*) = (*int m*) ^ *n*
by (*rule of-nat-power*)

lemmas *zpower-int* = *int-power* [*symmetric*]

30.23 Further theorems on numerals

30.23.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

lemmas *left-distrib-number-of* [*simp*] =
left-distrib [*of - - number-of v*, *standard*]

lemmas *right-distrib-number-of* [*simp*] =
right-distrib [*of number-of v*, *standard*]

lemmas *left-diff-distrib-number-of* [*simp*] =
left-diff-distrib [*of - - number-of v*, *standard*]

lemmas *right-diff-distrib-number-of* [*simp*] =
right-diff-distrib [*of number-of v*, *standard*]

These are actually for fields, like real: but where else to put them?

lemmas *zero-less-divide-iff-number-of* [*simp*, *noatp*] =
zero-less-divide-iff [*of number-of w*, *standard*]

lemmas *divide-less-0-iff-number-of* [*simp*, *noatp*] =
divide-less-0-iff [*of number-of w*, *standard*]

lemmas *zero-le-divide-iff-number-of* [*simp*, *noatp*] =
zero-le-divide-iff [*of number-of w*, *standard*]

lemmas *divide-le-0-iff-number-of* [*simp*, *noatp*] =
divide-le-0-iff [*of number-of w*, *standard*]

Replaces *inverse #nn* by *1/#nn*. It looks strange, but then other simprocs simplify the quotient.

lemmas *inverse-eq-divide-number-of* [*simp*] =
inverse-eq-divide [*of number-of w*, *standard*]

These laws simplify inequalities, moving unary minus from a term into the literal.

lemmas *less-minus-iff-number-of* [*simp*, *noatp*] =
less-minus-iff [*of number-of v*, *standard*]

lemmas *le-minus-iff-number-of* [*simp*, *noatp*] =
le-minus-iff [*of number-of v*, *standard*]

lemmas *equation-minus-iff-number-of* [*simp*, *noatp*] =
equation-minus-iff [*of number-of v*, *standard*]

lemmas *minus-less-iff-number-of* [*simp*, *noatp*] =
minus-less-iff [*of - number-of v*, *standard*]

lemmas *minus-le-iff-number-of* [*simp*, *noatp*] =
minus-le-iff [*of* - *number-of v*, *standard*]

lemmas *minus-equation-iff-number-of* [*simp*, *noatp*] =
minus-equation-iff [*of* - *number-of v*, *standard*]

To Simplify Inequalities Where One Side is the Constant 1

lemma *less-minus-iff-1* [*simp*,*noatp*]:
fixes *b*::'*b*::{*ordered-idom*,*number-ring*}
shows (*1* < − *b*) = (*b* < −*1*)
by *auto*

lemma *le-minus-iff-1* [*simp*,*noatp*]:
fixes *b*::'*b*::{*ordered-idom*,*number-ring*}
shows (*1* ≤ − *b*) = (*b* ≤ −*1*)
by *auto*

lemma *equation-minus-iff-1* [*simp*,*noatp*]:
fixes *b*::'*b*::*number-ring*
shows (*1* = − *b*) = (*b* = −*1*)
by (*subst equation-minus-iff*, *auto*)

lemma *minus-less-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::{*ordered-idom*,*number-ring*}
shows (− *a* < *1*) = (−*1* < *a*)
by *auto*

lemma *minus-le-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::{*ordered-idom*,*number-ring*}
shows (− *a* ≤ *1*) = (−*1* ≤ *a*)
by *auto*

lemma *minus-equation-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::*number-ring*
shows (− *a* = *1*) = (*a* = −*1*)
by (*subst minus-equation-iff*, *auto*)

Cancellation of constant factors in comparisons (< and ≤)

lemmas *mult-less-cancel-left-number-of* [*simp*, *noatp*] =
mult-less-cancel-left [*of number-of v*, *standard*]

lemmas *mult-less-cancel-right-number-of* [*simp*, *noatp*] =
mult-less-cancel-right [*of* - *number-of v*, *standard*]

lemmas *mult-le-cancel-left-number-of* [*simp*, *noatp*] =
mult-le-cancel-left [*of number-of v*, *standard*]

lemmas *mult-le-cancel-right-number-of* [*simp*, *noatp*] =

mult-le-cancel-right [*of* - *number-of v*, *standard*]

Multiplying out constant divisors in comparisons ($<$, $\leq$ and $=$)

lemmas *le-divide-eq-number-of1* [*simp*] = *le-divide-eq* [*of* - - *number-of w*, *standard*]
lemmas *divide-le-eq-number-of1* [*simp*] = *divide-le-eq* [*of* - *number-of w*, *standard*]
lemmas *less-divide-eq-number-of1* [*simp*] = *less-divide-eq* [*of* - - *number-of w*, *standard*]
lemmas *divide-less-eq-number-of1* [*simp*] = *divide-less-eq* [*of* - *number-of w*, *standard*]
lemmas *eq-divide-eq-number-of1* [*simp*] = *eq-divide-eq* [*of* - - *number-of w*, *standard*]
lemmas *divide-eq-eq-number-of1* [*simp*] = *divide-eq-eq* [*of* - *number-of w*, *standard*]

30.23.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas *le-divide-eq-number-of* = *le-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-le-eq-number-of* = *divide-le-eq* [*of* - - *number-of w*, *standard*]
lemmas *less-divide-eq-number-of* = *less-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-less-eq-number-of* = *divide-less-eq* [*of* - - *number-of w*, *standard*]
lemmas *eq-divide-eq-number-of* = *eq-divide-eq* [*of number-of w*, *standard*]
lemmas *divide-eq-eq-number-of* = *divide-eq-eq* [*of* - - *number-of w*, *standard*]

Not good as automatic simprules because they cause case splits.

lemmas *divide-const-simps* =
le-divide-eq-number-of divide-le-eq-number-of less-divide-eq-number-of
divide-less-eq-number-of eq-divide-eq-number-of divide-eq-eq-number-of
le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1

Division By -1

lemma *divide-minus1* [*simp*]:
$x/-1 = -(x::'a::\{field,division\text{-}by\text{-}zero,number\text{-}ring\})$
by *simp*

lemma *minus1-divide* [*simp*]:
$-1 \;/\; (x::'a::\{field,division\text{-}by\text{-}zero,number\text{-}ring\}) = -\,(1/x)$
by (*simp add*: *divide-inverse inverse-minus-eq*)

lemma *half-gt-zero-iff*:
$(0 < r/2) = (0 < (r::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero,number\text{-}ring\}))$
by *auto*

lemmas *half-gt-zero* [*simp*] = *half-gt-zero-iff* [*THEN iffD2*, *standard*]

30.24 Configuration of the code generator

code-datatype *Pls Min Bit0 Bit1 number-of* :: *int* ⇒ *int*

lemmas *pred-succ-numeral-code* [*code*] =
pred-bin-simps succ-bin-simps

lemmas *plus-numeral-code* [*code*] =
add-bin-simps
arith-extra-simps(*1*) [**where** *'a* = *int*]

lemmas *minus-numeral-code* [*code*] =
minus-bin-simps
arith-extra-simps(*2*) [**where** *'a* = *int*]
arith-extra-simps(*5*) [**where** *'a* = *int*]

lemmas *times-numeral-code* [*code*] =
mult-bin-simps
arith-extra-simps(*4*) [**where** *'a* = *int*]

instantiation *int* :: *eq*
begin

definition [*code del*]: *eq-class.eq k l* ⟷ *k* − *l* = (*0*::*int*)

instance by *default* (*simp add*: *eq-int-def*)

end

lemma *eq-number-of-int-code* [*code*]:
eq-class.eq (*number-of k* :: *int*) (*number-of l*) ⟷ *eq-class.eq k l*
unfolding *eq-int-def number-of-is-id* **..**

lemma *eq-int-code* [*code*]:
eq-class.eq Int.Pls Int.Pls ⟷ *True*
eq-class.eq Int.Pls Int.Min ⟷ *False*
eq-class.eq Int.Pls (*Int.Bit0 k2*) ⟷ *eq-class.eq Int.Pls k2*
eq-class.eq Int.Pls (*Int.Bit1 k2*) ⟷ *False*
eq-class.eq Int.Min Int.Pls ⟷ *False*
eq-class.eq Int.Min Int.Min ⟷ *True*
eq-class.eq Int.Min (*Int.Bit0 k2*) ⟷ *False*
eq-class.eq Int.Min (*Int.Bit1 k2*) ⟷ *eq-class.eq Int.Min k2*
eq-class.eq (*Int.Bit0 k1*) *Int.Pls* ⟷ *eq-class.eq k1 Int.Pls*
eq-class.eq (*Int.Bit1 k1*) *Int.Pls* ⟷ *False*
eq-class.eq (*Int.Bit0 k1*) *Int.Min* ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) *Int.Min* ⟷ *eq-class.eq k1 Int.Min*
eq-class.eq (*Int.Bit0 k1*) (*Int.Bit0 k2*) ⟷ *eq-class.eq k1 k2*
eq-class.eq (*Int.Bit0 k1*) (*Int.Bit1 k2*) ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) (*Int.Bit0 k2*) ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) (*Int.Bit1 k2*) ⟷ *eq-class.eq k1 k2*
unfolding *eq-equals* **by** *simp-all*

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

lemma *less-eq-number-of-int-code* [*code*]:
(*number-of k* :: *int*) ≤ *number-of l* ⟷ *k* ≤ *l*
unfolding *number-of-is-id* **..**

lemma *less-eq-int-code* [*code*]:
Int.Pls ≤ *Int.Pls* ⟷ *True*
Int.Pls ≤ *Int.Min* ⟷ *False*
Int.Pls ≤ *Int.Bit0 k* ⟷ *Int.Pls* ≤ *k*
Int.Pls ≤ *Int.Bit1 k* ⟷ *Int.Pls* ≤ *k*
Int.Min ≤ *Int.Pls* ⟷ *True*
Int.Min ≤ *Int.Min* ⟷ *True*
Int.Min ≤ *Int.Bit0 k* ⟷ *Int.Min* < *k*
Int.Min ≤ *Int.Bit1 k* ⟷ *Int.Min* ≤ *k*
Int.Bit0 k ≤ *Int.Pls* ⟷ *k* ≤ *Int.Pls*
Int.Bit1 k ≤ *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit0 k ≤ *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit1 k ≤ *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit0 k1 ≤ *Int.Bit0 k2* ⟷ *k1* ≤ *k2*
Int.Bit0 k1 ≤ *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
Int.Bit1 k1 ≤ *Int.Bit0 k2* ⟷ *k1* < *k2*
Int.Bit1 k1 ≤ *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
by *simp-all*

lemma *less-number-of-int-code* [*code*]:
(*number-of k* :: *int*) < *number-of l* ⟷ *k* < *l*
unfolding *number-of-is-id* **..**

lemma *less-int-code* [*code*]:
Int.Pls < *Int.Pls* ⟷ *False*
Int.Pls < *Int.Min* ⟷ *False*
Int.Pls < *Int.Bit0 k* ⟷ *Int.Pls* < *k*
Int.Pls < *Int.Bit1 k* ⟷ *Int.Pls* ≤ *k*
Int.Min < *Int.Pls* ⟷ *True*
Int.Min < *Int.Min* ⟷ *False*
Int.Min < *Int.Bit0 k* ⟷ *Int.Min* < *k*
Int.Min < *Int.Bit1 k* ⟷ *Int.Min* < *k*
Int.Bit0 k < *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit1 k < *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit0 k < *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit1 k < *Int.Min* ⟷ *k* < *Int.Min*
Int.Bit0 k1 < *Int.Bit0 k2* ⟷ *k1* < *k2*
Int.Bit0 k1 < *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
Int.Bit1 k1 < *Int.Bit0 k2* ⟷ *k1* < *k2*
Int.Bit1 k1 < *Int.Bit1 k2* ⟷ *k1* < *k2*
by *simp-all*

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

lemma [*code*]:
nat-aux i n = (*if i* ≤ *0 then n else nat-aux* (*i* − *1*) (*Suc n*)) — tail recursive
by (*auto simp add*: *nat-aux-def nat-eq-iff linorder-not-le order-less-imp-le*
dest: *zless-imp-add1-zle*)

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

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

lemma *zero-is-num-zero* [*code*, *code inline*, *symmetric*, *code post*]:
(*0*::*int*) = *Numeral0*
by *simp*

lemma *one-is-num-one* [*code*, *code inline*, *symmetric*, *code post*]:
(*1*::*int*) = *Numeral1*
by *simp*

code-modulename *SML*
Int Integer

code-modulename *OCaml*
Int Integer

code-modulename *Haskell*
Int Integer

types-code
int (*int*)
attach (*term-of*) ⟪
val term-of-int = *HOLogic.mk-number HOLogic.intT*;
⟫
attach (*test*) ⟪
fun gen-int i =
let val j = *one-of* [~*1*, *1*] ∗ *random-range 0 i*
in (*j*, *fn* () => *term-of-int j*) *end*;
⟫

setup ⟪
let

fun strip-number-of (@{*term Int.number-of* :: *int* => *int*} $ *t*) = *t*
| *strip-number-of t* = *t*;

```
fun numeral-codegen thy defs dep module b t gr =
  let val i = HOLogic.dest-numeral (strip-number-of t)
  in
    SOME (Codegen.str (string-of-int i),
      snd (Codegen.invoke-tycodegen thy defs dep module false HOLogic.intT gr))
  end handle TERM - => NONE;

in

Codegen.add-codegen numeral-codegen numeral-codegen

end
⟩⟩
```

consts-code
number-of :: *int* ⇒ *int* ((-))
0 :: *int* (*0*)
1 :: *int* (*1*)
uminus :: *int* => *int* (~)
op + :: *int* => *int* => *int* ((- +/ -))
op ∗ :: *int* => *int* => *int* ((- */ -))
op ≤ :: *int* => *int* => *bool* ((- <=/ -))
op < :: *int* => *int* => *bool* ((- </ -))

quickcheck-params [*default-type* = *int*]

hide (**open**) *const Pls Min Bit0 Bit1 succ pred*

30.25 Legacy theorems

lemmas *zminus-zminus* = *minus-minus* [*of z*::*int*, *standard*]
lemmas *zminus-0* = *minus-zero* [**where** *'a*=*int*]
lemmas *zminus-zadd-distrib* = *minus-add-distrib* [*of z*::*int w*, *standard*]
lemmas *zadd-commute* = *add-commute* [*of z*::*int w*, *standard*]
lemmas *zadd-assoc* = *add-assoc* [*of z1*::*int z2 z3*, *standard*]
lemmas *zadd-left-commute* = *add-left-commute* [*of x*::*int y z*, *standard*]
lemmas *zadd-ac* = *zadd-assoc zadd-commute zadd-left-commute*
lemmas *zmult-ac* = *OrderedGroup.mult-ac*
lemmas *zadd-0* = *OrderedGroup.add-0-left* [*of z*::*int*, *standard*]
lemmas *zadd-0-right* = *OrderedGroup.add-0-left* [*of z*::*int*, *standard*]
lemmas *zadd-zminus-inverse2* = *left-minus* [*of z*::*int*, *standard*]
lemmas *zmult-zminus* = *mult-minus-left* [*of z*::*int w*, *standard*]
lemmas *zmult-commute* = *mult-commute* [*of z*::*int w*, *standard*]
lemmas *zmult-assoc* = *mult-assoc* [*of z1*::*int z2 z3*, *standard*]
lemmas *zadd-zmult-distrib* = *left-distrib* [*of z1*::*int z2 w*, *standard*]
lemmas *zadd-zmult-distrib2* = *right-distrib* [*of w*::*int z1 z2*, *standard*]
lemmas *zdiff-zmult-distrib* = *left-diff-distrib* [*of z1*::*int z2 w*, *standard*]
lemmas *zdiff-zmult-distrib2* = *right-diff-distrib* [*of w*::*int z1 z2*, *standard*]

lemmas *zmult-1 = mult-1-left [of z::int, standard]*
lemmas *zmult-1-right = mult-1-right [of z::int, standard]*

lemmas *zle-refl = order-refl [of w::int, standard]*
lemmas *zle-trans = order-trans [***where** *'a=int* **and** *x=i* **and** *y=j* **and** *z=k, standard]*
lemmas *zle-anti-sym = order-antisym [of z::int w, standard]*
lemmas *zle-linear = linorder-linear [of z::int w, standard]*
lemmas *zless-linear = linorder-less-linear [***where** *'a = int]*

lemmas *zadd-left-mono = add-left-mono [of i::int j k, standard]*
lemmas *zadd-strict-right-mono = add-strict-right-mono [of i::int j k, standard]*
lemmas *zadd-zless-mono = add-less-le-mono [of w'::int w z' z, standard]*

lemmas *int-0-less-1 = zero-less-one [***where** *'a=int]*
lemmas *int-0-neq-1 = zero-neq-one [***where** *'a=int]*

lemmas *inj-int = inj-of-nat [***where** *'a=int]*
lemmas *zadd-int = of-nat-add [***where** *'a=int, symmetric]*
lemmas *int-mult = of-nat-mult [***where** *'a=int]*
lemmas *zmult-int = of-nat-mult [***where** *'a=int, symmetric]*
lemmas *int-eq-0-conv = of-nat-eq-0-iff [***where** *'a=int* **and** *m=n, standard]*
lemmas *zless-int = of-nat-less-iff [***where** *'a=int]*
lemmas *int-less-0-conv = of-nat-less-0-iff [***where** *'a=int* **and** *m=k, standard]*
lemmas *zero-less-int-conv = of-nat-0-less-iff [***where** *'a=int]*
lemmas *zero-zle-int = of-nat-0-le-iff [***where** *'a=int]*
lemmas *int-le-0-conv = of-nat-le-0-iff [***where** *'a=int* **and** *m=n, standard]*
lemmas *int-0 = of-nat-0 [***where** *'a=int]*
lemmas *int-1 = of-nat-1 [***where** *'a=int]*
lemmas *int-Suc = of-nat-Suc [***where** *'a=int]*
lemmas *abs-int-eq = abs-of-nat [***where** *'a=int* **and** *n=m, standard]*
lemmas *of-int-int-eq = of-int-of-nat-eq [***where** *'a=int]*
lemmas *zdiff-int = of-nat-diff [***where** *'a=int, symmetric]*
lemmas *zless-le = less-int-def*
lemmas *int-eq-of-nat = TrueI*

end

31 IntDiv: The Division Operators div and mod

theory *IntDiv*
imports *Int Divides FunDef*
uses
~~/src/Provers/Arith/cancel-numeral-factor.ML
~~/src/Provers/Arith/extract-common-term.ML
(Tools/int-factor-simprocs.ML)
begin

definition *divmod-rel* :: *int* ⇒ *int* ⇒ *int* × *int* ⇒ *bool* **where**
— definition of quotient and remainder
[*code*]: *divmod-rel a b* = (λ(*q*, *r*). *a* = *b* ∗ *q* + *r* ∧
(*if* 0 < *b then* 0 ≤ *r* ∧ *r* < *b else b* < *r* ∧ *r* ≤ 0))

definition *adjust* :: *int* ⇒ *int* × *int* ⇒ *int* × *int* **where**
— for the division algorithm
[*code*]: *adjust b* = (λ(*q*, *r*). *if* 0 ≤ *r* − *b then* (2 ∗ *q* + 1, *r* − *b*)
else (2 ∗ *q*, *r*))

algorithm for the case $a \geq 0$, $b > 0$

function *posDivAlg* :: *int* ⇒ *int* ⇒ *int* × *int* **where**
posDivAlg a b = (*if a* < *b* ∨ *b* ≤ 0 *then* (0, *a*)
else adjust b (*posDivAlg a* (2 ∗ *b*)))
by *auto*
termination by (*relation measure* (λ(*a*, *b*). *nat* (*a* − *b* + 1))) *auto*

algorithm for the case $a < 0$, $b > 0$

function *negDivAlg* :: *int* ⇒ *int* ⇒ *int* × *int* **where**
negDivAlg a b = (*if* 0 ≤*a* + *b* ∨ *b* ≤ 0 *then* (−1, *a* + *b*)
else adjust b (*negDivAlg a* (2 ∗ *b*)))
by *auto*
termination by (*relation measure* (λ(*a*, *b*). *nat* (− *a* − *b*))) *auto*

algorithm for the general case $b \neq (0::'a)$

definition *negateSnd* :: *int* × *int* ⇒ *int* × *int* **where**
[*code inline*]: *negateSnd* = *apsnd uminus*

definition *divmod* :: *int* ⇒ *int* ⇒ *int* × *int* **where**
— The full division algorithm considers all possible signs for a, b including the special case $a=0$, $b<0$ because *negDivAlg* requires $a < (0::'a)$.
divmod a b = (*if* 0 ≤ *a then if* 0 ≤ *b then posDivAlg a b*
else if a = 0 *then* (0, 0)
else negateSnd (*negDivAlg* (−*a*) (−*b*))
else
if 0 < *b then negDivAlg a b*
else negateSnd (*posDivAlg* (−*a*) (−*b*)))

instantiation *int* :: *Divides.div*
begin

definition
div-def: *a div b* = *fst* (*divmod a b*)

definition
mod-def: *a mod b* = *snd* (*divmod a b*)

instance ..

end

lemma *divmod-mod-div*:
divmod p q = (*p div q*, *p mod q*)
by (*auto simp add*: *div-def mod-def*)

Here is the division algorithm in ML:

```
    fun posDivAlg (a,b) =
      if a<b then (0,a)
      else let val (q,r) = posDivAlg(a, 2*b)
       in  if 0\<le>r-b then (2*q+1, r-b) else (2*q, r)
   end

    fun negDivAlg (a,b) =
      if 0\<le>a+b then (~1,a+b)
      else let val (q,r) = negDivAlg(a, 2*b)
       in  if 0\<le>r-b then (2*q+1, r-b) else (2*q, r)
   end;

    fun negateSnd (q,r:int) = (q,~r);

    fun divmod (a,b) = if 0\<le>a then
  if b>0 then posDivAlg (a,b)
   else if a=0 then (0,0)
else negateSnd (negDivAlg (~a,~b))
       else
  if 0<b then negDivAlg (a,b)
  else        negateSnd (posDivAlg (~a,~b));
```

31.1 Uniqueness and Monotonicity of Quotients and Remainders

lemma *unique-quotient-lemma*:
[| $b*q' + r' \le b*q + r$; $0 \le r'$; $r' < b$; $r < b$ |]
==> $q' \le (q::int)$
apply (*subgoal-tac* $r' + b * (q'-q) \le r$)
prefer *2* **apply** (*simp add*: *right-diff-distrib*)
apply (*subgoal-tac* $0 < b * (1 + q - q')$)
apply (*erule-tac* [*2*] *order-le-less-trans*)
prefer *2* **apply** (*simp add*: *right-diff-distrib right-distrib*)
apply (*subgoal-tac* $b * q' < b * (1 + q)$)
prefer *2* **apply** (*simp add*: *right-diff-distrib right-distrib*)
apply (*simp add*: *mult-less-cancel-left*)
done

lemma *unique-quotient-lemma-neg*:
[| $b*q' + r' \leq b*q + r$; $r \leq 0$; $b < r$; $b < r'$ |]
==> $q \leq (q'::int)$
by (*rule-tac* $b = -b$ **and** $r = -r'$ **and** $r' = -r$ **in** *unique-quotient-lemma*, *auto*)

lemma *unique-quotient*:
[| *divmod-rel a b* (q, r); *divmod-rel a b* (q', r'); $b \neq 0$ |]
==> $q = q'$
apply (*simp add*: *divmod-rel-def linorder-neq-iff split*: *split-if-asm*)
apply (*blast intro*: *order-antisym*
dest: *order-eq-refl* [*THEN unique-quotient-lemma*]
order-eq-refl [*THEN unique-quotient-lemma-neg*] *sym*)+
done

lemma *unique-remainder*:
[| *divmod-rel a b* (q, r); *divmod-rel a b* (q', r'); $b \neq 0$ |]
==> $r = r'$
apply (*subgoal-tac* $q = q'$)
apply (*simp add*: *divmod-rel-def*)
apply (*blast intro*: *unique-quotient*)
done

31.2 Correctness of *posDivAlg*, the Algorithm for Non-Negative Dividends

And positive divisors

lemma *adjust-eq* [*simp*]:
adjust b (q,r) =
(*let diff* = $r-b$ *in*
if $0 \leq$ *diff then* ($2*q + 1$, *diff*)
else ($2*q$, r))
by (*simp add*: *Let-def adjust-def*)

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

use with a simproc to avoid repeatedly proving the premise

lemma *posDivAlg-eqn*:
$0 < b$ ==>
posDivAlg a b = (*if* $a<b$ *then* (0,a) *else adjust b* (*posDivAlg a* ($2*b$)))
by (*rule posDivAlg.simps* [*THEN trans*], *simp*)

Correctness of *posDivAlg*: it computes quotients correctly

theorem *posDivAlg-correct*:
assumes $0 \leq a$ **and** $0 < b$
shows *divmod-rel a b* (*posDivAlg a b*)

using *prems* **apply** (*induct a b rule*: *posDivAlg.induct*)
apply *auto*
apply (*simp add*: *divmod-rel-def*)
apply (*subst posDivAlg-eqn*, *simp add*: *right-distrib*)
apply (*case-tac a < b*)
apply *simp-all*
apply (*erule splitE*)
apply (*auto simp add*: *right-distrib Let-def*)
done

31.3 Correctness of *negDivAlg*, the Algorithm for Negative Dividends

And positive divisors

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

use with a simproc to avoid repeatedly proving the premise

lemma *negDivAlg-eqn*:
0 < b ==>
negDivAlg a b =
(if 0≤a+b then (−1,a+b) else adjust b (negDivAlg a (2∗b)))
by (*rule negDivAlg.simps* [*THEN trans*], *simp*)

lemma *negDivAlg-correct*:
assumes *a < 0* **and** *b > 0*
shows *divmod-rel a b* (*negDivAlg a b*)
using *prems* **apply** (*induct a b rule*: *negDivAlg.induct*)
apply (*auto simp add*: *linorder-not-le*)
apply (*simp add*: *divmod-rel-def*)
apply (*subst negDivAlg-eqn*, *assumption*)
apply (*case-tac a + b < (0*::*int)*)
apply *simp-all*
apply (*erule splitE*)
apply (*auto simp add*: *right-distrib Let-def*)
done

31.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma *divmod-rel-0*: *b ≠ 0 ==> divmod-rel 0 b* (*0*, *0*)
by (*auto simp add*: *divmod-rel-def linorder-neq-iff*)

lemma *posDivAlg-0* [*simp*]: *posDivAlg 0 b* = (*0*, *0*)
by (*subst posDivAlg.simps*, *auto*)

lemma *negDivAlg-minus1* [*simp*]: *negDivAlg −1 b* = (*−1*, *b − 1*)
by (*subst negDivAlg.simps*, *auto*)

lemma *negateSnd-eq* [*simp*]: *negateSnd*(*q*,*r*) = (*q*,−*r*)
by (*simp add*: *negateSnd-def*)

lemma *divmod-rel-neg*: *divmod-rel* (−*a*) (−*b*) *qr* ==> *divmod-rel a b* (*negateSnd qr*)
by (*auto simp add*: *split-ifs divmod-rel-def*)

lemma *divmod-correct*: *b* ≠ *0* ==> *divmod-rel a b* (*divmod a b*)
by (*force simp add*: *linorder-neq-iff divmod-rel-0 divmod-def divmod-rel-neg*
posDivAlg-correct negDivAlg-correct)

Arbitrary definitions for division by zero. Useful to simplify certain equations.

lemma *DIVISION-BY-ZERO* [*simp*]: *a div* (*0*::*int*) = *0* & *a mod* (*0*::*int*) = *a*
by (*simp add*: *div-def mod-def divmod-def posDivAlg.simps*)

Basic laws about division and remainder

lemma *zmod-zdiv-equality*: (*a*::*int*) = *b* ∗ (*a div b*) + (*a mod b*)
apply (*case-tac b* = *0*, *simp*)
apply (*cut-tac a* = *a* **and** *b* = *b* **in** *divmod-correct*)
apply (*auto simp add*: *divmod-rel-def div-def mod-def*)
done

lemma *zdiv-zmod-equality*: (*b* ∗ (*a div b*) + (*a mod b*)) + *k* = (*a*::*int*)+*k*
by(*simp add*: *zmod-zdiv-equality*[*symmetric*])

lemma *zdiv-zmod-equality2*: ((*a div b*) ∗ *b* + (*a mod b*)) + *k* = (*a*::*int*)+*k*
by(*simp add*: *mult-commute zmod-zdiv-equality*[*symmetric*])

Tool setup

ML ⟪
local

structure CancelDivMod = *CancelDivModFun*(
struct
val div-name = @{*const-name Divides.div*};
val mod-name = @{*const-name Divides.mod*};
val mk-binop = *HOLogic.mk-binop*;
val mk-sum = *Int-Numeral-Simprocs.mk-sum HOLogic.intT*;
val dest-sum = *Int-Numeral-Simprocs.dest-sum*;
val div-mod-eqs =
map mk-meta-eq [@{*thm zdiv-zmod-equality*},
@{*thm zdiv-zmod-equality2*}];
val trans = *trans*;
val prove-eq-sums =
let
val simps = @{*thm diff-int-def*} :: *Int-Numeral-Simprocs.add-0s* @ @{*thms zadd-ac*}

```
    in Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac simps) end;
end)

in

val cancel-zdiv-zmod-proc = Simplifier.simproc (the-context ())
  cancel-zdiv-zmod [(m::int) + n] (K CancelDivMod.proc)

end;

Addsimprocs [cancel-zdiv-zmod-proc]
⟩⟩

lemma pos-mod-conj : (0::int) < b ==> 0 ≤ a mod b & a mod b < b
apply (cut-tac a = a and b = b in divmod-correct)
apply (auto simp add: divmod-rel-def mod-def)
done

lemmas pos-mod-sign  [simp] = pos-mod-conj [THEN conjunct1, standard]
   and pos-mod-bound [simp] = pos-mod-conj [THEN conjunct2, standard]

lemma neg-mod-conj : b < (0::int) ==> a mod b ≤ 0 & b < a mod b
apply (cut-tac a = a and b = b in divmod-correct)
apply (auto simp add: divmod-rel-def div-def mod-def)
done

lemmas neg-mod-sign  [simp] = neg-mod-conj [THEN conjunct1, standard]
   and neg-mod-bound [simp] = neg-mod-conj [THEN conjunct2, standard]
```

31.5 General Properties of div and mod

```
lemma divmod-rel-div-mod: b ≠ 0 ==> divmod-rel a b (a div b, a mod b)
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (force simp add: divmod-rel-def linorder-neq-iff)
done

lemma divmod-rel-div: [| divmod-rel a b (q, r);  b ≠ 0 |] ==> a div b = q
by (simp add: divmod-rel-div-mod [THEN unique-quotient])

lemma divmod-rel-mod: [| divmod-rel a b (q, r);  b ≠ 0 |] ==> a mod b = r
by (simp add: divmod-rel-div-mod [THEN unique-remainder])

lemma div-pos-pos-trivial: [| (0::int) ≤ a;  a < b |] ==> a div b = 0
apply (rule divmod-rel-div)
apply (auto simp add: divmod-rel-def)
done

lemma div-neg-neg-trivial: [| a ≤ (0::int);  b < a |] ==> a div b = 0
apply (rule divmod-rel-div)
```

apply (*auto simp add*: *divmod-rel-def*)
done

lemma *div-pos-neg-trivial*: [| (*0*::*int*) < *a*; *a*+*b* ≤ *0* |] ==> *a div b* = −*1*
apply (*rule divmod-rel-div*)
apply (*auto simp add*: *divmod-rel-def*)
done

lemma *mod-pos-pos-trivial*: [| (*0*::*int*) ≤ *a*; *a* < *b* |] ==> *a mod b* = *a*
apply (*rule-tac q* = *0* **in** *divmod-rel-mod*)
apply (*auto simp add*: *divmod-rel-def*)
done

lemma *mod-neg-neg-trivial*: [| *a* ≤ (*0*::*int*); *b* < *a* |] ==> *a mod b* = *a*
apply (*rule-tac q* = *0* **in** *divmod-rel-mod*)
apply (*auto simp add*: *divmod-rel-def*)
done

lemma *mod-pos-neg-trivial*: [| (*0*::*int*) < *a*; *a*+*b* ≤ *0* |] ==> *a mod b* = *a*+*b*
apply (*rule-tac q* = −*1* **in** *divmod-rel-mod*)
apply (*auto simp add*: *divmod-rel-def*)
done

There is no *mod-neg-pos-trivial*.

lemma *zdiv-zminus-zminus* [*simp*]: (−*a*) *div* (−*b*) = *a div* (*b*::*int*)
apply (*case-tac b* = *0*, *simp*)
apply (*simp add*: *divmod-rel-div-mod* [*THEN divmod-rel-neg*, *simplified*,
THEN divmod-rel-div, *THEN sym*])

done

lemma *zmod-zminus-zminus* [*simp*]: (−*a*) *mod* (−*b*) = − (*a mod* (*b*::*int*))
apply (*case-tac b* = *0*, *simp*)
apply (*subst divmod-rel-div-mod* [*THEN divmod-rel-neg*, *simplified*, *THEN divmod-rel-mod*],
auto)
done

31.6 Laws for div and mod with Unary Minus

lemma *zminus1-lemma*:
divmod-rel a b (*q*, *r*)
==> *divmod-rel* (−*a*) *b* (*if r*=*0 then* −*q else* −*q* − *1*,
if r=*0 then 0 else b*−*r*)
by (*force simp add*: *split-ifs divmod-rel-def linorder-neq-iff right-diff-distrib*)

lemma *zdiv-zminus1-eq-if*:
$b \neq (0::int)$
$\Longrightarrow (-a)\ div\ b =$
$(if\ a\ mod\ b = 0\ then\ -\ (a\ div\ b)\ else\ -\ (a\ div\ b) - 1)$
by (*blast intro*: *divmod-rel-div-mod* [*THEN zminus1-lemma*, *THEN divmod-rel-div*])

lemma *zmod-zminus1-eq-if*:
$(-a::int)\ mod\ b = (if\ a\ mod\ b = 0\ then\ 0\ else\ b - (a\ mod\ b))$
apply (*case-tac* $b = 0$, *simp*)
apply (*blast intro*: *divmod-rel-div-mod* [*THEN zminus1-lemma*, *THEN divmod-rel-mod*])
done

lemma *zmod-zminus1-not-zero*:
fixes $k\ l :: int$
shows $-\ k\ mod\ l \neq 0 \Longrightarrow k\ mod\ l \neq 0$
unfolding *zmod-zminus1-eq-if* **by** *auto*

lemma *zdiv-zminus2*: $a\ div\ (-b) = (-a::int)\ div\ b$
by (*cut-tac* $a = -a$ **in** *zdiv-zminus-zminus*, *auto*)

lemma *zmod-zminus2*: $a\ mod\ (-b) = -\ ((-a::int)\ mod\ b)$
by (*cut-tac* $a = -a$ **and** $b = b$ **in** *zmod-zminus-zminus*, *auto*)

lemma *zdiv-zminus2-eq-if*:
$b \neq (0::int)$
$\Longrightarrow a\ div\ (-b) =$
$(if\ a\ mod\ b = 0\ then\ -\ (a\ div\ b)\ else\ -\ (a\ div\ b) - 1)$
by (*simp add*: *zdiv-zminus1-eq-if zdiv-zminus2*)

lemma *zmod-zminus2-eq-if*:
$a\ mod\ (-b::int) = (if\ a\ mod\ b = 0\ then\ 0\ else\ (a\ mod\ b) - b)$
by (*simp add*: *zmod-zminus1-eq-if zmod-zminus2*)

lemma *zmod-zminus2-not-zero*:
fixes $k\ l :: int$
shows $k\ mod\ -\ l \neq 0 \Longrightarrow k\ mod\ l \neq 0$
unfolding *zmod-zminus2-eq-if* **by** *auto*

31.7 Division of a Number by Itself

lemma *self-quotient-aux1*: $[|\ (0::int) < a;\ a = r + a*q;\ r < a\ |] \Longrightarrow 1 \leq q$
apply (*subgoal-tac* $0 < a*q$)
apply (*simp add*: *zero-less-mult-iff*, *arith*)
done

lemma *self-quotient-aux2*: $[|\ (0::int) < a;\ a = r + a*q;\ 0 \leq r\ |] \Longrightarrow q \leq 1$
apply (*subgoal-tac* $0 \leq a*\ (1-q)$)
apply (*simp add*: *zero-le-mult-iff*)
apply (*simp add*: *right-diff-distrib*)

done

lemma *self-quotient*: [| *divmod-rel a a* (*q*, *r*); *a* ≠ (*0*::*int*) |] ==> *q* = *1*
apply (*simp add*: *split-ifs divmod-rel-def linorder-neq-iff*)
apply (*rule order-antisym*, *safe*, *simp-all*)
apply (*rule-tac* [*3*] *a* = −*a* **and** *r* = −*r* **in** *self-quotient-aux1*)
apply (*rule-tac a* = −*a* **and** *r* = −*r* **in** *self-quotient-aux2*)
apply (*force intro*: *self-quotient-aux1 self-quotient-aux2 simp add*: *add-commute*)+
done

lemma *self-remainder*: [| *divmod-rel a a* (*q*, *r*); *a* ≠ (*0*::*int*) |] ==> *r* = *0*
apply (*frule self-quotient*, *assumption*)
apply (*simp add*: *divmod-rel-def*)
done

lemma *zdiv-self* [*simp*]: *a* ≠ *0* ==> *a div a* = (*1*::*int*)
by (*simp add*: *divmod-rel-div-mod* [*THEN self-quotient*])

lemma *zmod-self* [*simp*]: *a mod a* = (*0*::*int*)
apply (*case-tac a* = *0*, *simp*)
apply (*simp add*: *divmod-rel-div-mod* [*THEN self-remainder*])
done

31.8 Computation of Division and Remainder

lemma *zdiv-zero* [*simp*]: (*0*::*int*) *div b* = *0*
by (*simp add*: *div-def divmod-def*)

lemma *div-eq-minus1*: (*0*::*int*) < *b* ==> −*1 div b* = −*1*
by (*simp add*: *div-def divmod-def*)

lemma *zmod-zero* [*simp*]: (*0*::*int*) *mod b* = *0*
by (*simp add*: *mod-def divmod-def*)

lemma *zmod-minus1*: (*0*::*int*) < *b* ==> −*1 mod b* = *b* − *1*
by (*simp add*: *mod-def divmod-def*)

a positive, b positive

lemma *div-pos-pos*: [| *0* < *a*; *0* ≤ *b* |] ==> *a div b* = *fst* (*posDivAlg a b*)
by (*simp add*: *div-def divmod-def*)

lemma *mod-pos-pos*: [| *0* < *a*; *0* ≤ *b* |] ==> *a mod b* = *snd* (*posDivAlg a b*)
by (*simp add*: *mod-def divmod-def*)

a negative, b positive

lemma *div-neg-pos*: [| *a* < *0*; *0* < *b* |] ==> *a div b* = *fst* (*negDivAlg a b*)
by (*simp add*: *div-def divmod-def*)

lemma *mod-neg-pos*: [| $a < 0$; $0 < b$ |] ==> a *mod* b = *snd* (*negDivAlg* a b)
by (*simp add*: *mod-def divmod-def*)

a positive, b negative

lemma *div-pos-neg*:
[| $0 < a$; $b < 0$ |] ==> a *div* b = *fst* (*negateSnd* (*negDivAlg* $(-a)$ $(-b)$))
by (*simp add*: *div-def divmod-def*)

lemma *mod-pos-neg*:
[| $0 < a$; $b < 0$ |] ==> a *mod* b = *snd* (*negateSnd* (*negDivAlg* $(-a)$ $(-b)$))
by (*simp add*: *mod-def divmod-def*)

a negative, b negative

lemma *div-neg-neg*:
[| $a < 0$; $b \leq 0$ |] ==> a *div* b = *fst* (*negateSnd* (*posDivAlg* $(-a)$ $(-b)$))
by (*simp add*: *div-def divmod-def*)

lemma *mod-neg-neg*:
[| $a < 0$; $b \leq 0$ |] ==> a *mod* b = *snd* (*negateSnd* (*posDivAlg* $(-a)$ $(-b)$))
by (*simp add*: *mod-def divmod-def*)

Simplify expresions in which div and mod combine numerical constants

lemma *divmod-relI*:
⟦$a == b * q + r$; *if* $0 < b$ *then* $0 \leq r \wedge r < b$ *else* $b < r \wedge r \leq 0$⟧
$\Longrightarrow$ *divmod-rel* a b (q, r)
unfolding *divmod-rel-def* **by** *simp*

lemmas *divmod-rel-div-eq* = *divmod-relI* [*THEN divmod-rel-div*, *THEN eq-reflection*]
lemmas *divmod-rel-mod-eq* = *divmod-relI* [*THEN divmod-rel-mod*, *THEN eq-reflection*]
lemmas *arithmetic-simps* =
arith-simps
add-special
OrderedGroup.add-0-left
OrderedGroup.add-0-right
mult-zero-left
mult-zero-right
mult-1-left
mult-1-right

ML ⟪
local
val mk-number = *HOLogic.mk-number HOLogic.intT*;
fun mk-cert u k l = @{*term plus* :: *int* ⇒ *int* ⇒ *int*} $
(@{*term times* :: *int* ⇒ *int* ⇒ *int*} $ *u* $ *mk-number k*) $
mk-number l;
fun prove ctxt prop = *Goal.prove ctxt* [] [] *prop*
(*K* (*ALLGOALS* (*full-simp-tac* (*HOL-basic-ss addsimps* @{*thms arithmetic-simps*}))));
fun binary-proc proc ss ct =

```
    (case Thm.term-of ct of
      - $ t $ u =>
      (case try (pairself (`(snd o HOLogic.dest-number))) (t, u) of
        SOME args => proc (Simplifier.the-context ss) args
      | NONE => NONE)
    | - => NONE);
in
  fun divmod-proc rule = binary-proc (fn ctxt => fn ((m, t), (n, u)) =>
    if n = 0 then NONE
    else let val (k, l) = Integer.div-mod m n;
    in SOME (rule OF [prove ctxt (Logic.mk-equals (t, mk-cert u k l))]) end);
end
⟫

simproc-setup binary-int-div (number-of m div number-of n :: int) =
  ⟪ K (divmod-proc (@{thm divmod-rel-div-eq})) ⟫

simproc-setup binary-int-mod (number-of m mod number-of n :: int) =
  ⟪ K (divmod-proc (@{thm divmod-rel-mod-eq})) ⟫

lemmas posDivAlg-eqn-number-of [simp] =
    posDivAlg-eqn [of number-of v number-of w, standard]

lemmas negDivAlg-eqn-number-of [simp] =
    negDivAlg-eqn [of number-of v number-of w, standard]
```

Special-case simplification

```
lemma zmod-minus1-right [simp]: a mod (−1::int) = 0
apply (cut-tac a = a and b = −1 in neg-mod-sign)
apply (cut-tac [2] a = a and b = −1 in neg-mod-bound)
apply (auto simp del: neg-mod-sign neg-mod-bound)
done

lemma zdiv-minus1-right [simp]: a div (−1::int) = −a
by (cut-tac a = a and b = −1 in zmod-zdiv-equality, auto)



lemmas div-pos-pos-1-number-of [simp] =
    div-pos-pos [OF int-0-less-1, of number-of w, standard]

lemmas div-pos-neg-1-number-of [simp] =
    div-pos-neg [OF int-0-less-1, of number-of w, standard]

lemmas mod-pos-pos-1-number-of [simp] =
    mod-pos-pos [OF int-0-less-1, of number-of w, standard]

lemmas mod-pos-neg-1-number-of [simp] =
    mod-pos-neg [OF int-0-less-1, of number-of w, standard]
```

lemmas *posDivAlg-eqn-1-number-of* [*simp*] =
  posDivAlg-eqn [*of concl*: *1 number-of w*, *standard*]

lemmas *negDivAlg-eqn-1-number-of* [*simp*] =
  negDivAlg-eqn [*of concl*: *1 number-of w*, *standard*]

31.9 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*: [| $a \leq a'$; $0 < (b::int)$ |] ==> a *div* $b \leq a'$ *div* b
apply (*cut-tac* $a = a$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*cut-tac* $a = a'$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*rule unique-quotient-lemma*)
apply (*erule subst*)
apply (*erule subst*, *simp-all*)
done

lemma *zdiv-mono1-neg*: [| $a \leq a'$; $(b::int) < 0$ |] ==> a' *div* $b \leq a$ *div* b
apply (*cut-tac* $a = a$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*cut-tac* $a = a'$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*rule unique-quotient-lemma-neg*)
apply (*erule subst*)
apply (*erule subst*, *simp-all*)
done

31.10 Monotonicity in the Second Argument (Divisor)

lemma *q-pos-lemma*:
  [| $0 \leq b'*q' + r'$; $r' < b'$; $0 < b'$ |] ==> $0 \leq (q'::int)$
apply (*subgoal-tac* $0 < b'* (q' + 1)$)
 apply (*simp add*: *zero-less-mult-iff*)
apply (*simp add*: *right-distrib*)
done

lemma *zdiv-mono2-lemma*:
  [| $b*q + r = b'*q' + r'$; $0 \leq b'*q' + r'$;
    $r' < b'$; $0 \leq r$; $0 < b'$; $b' \leq b$ |]
  ==> $q \leq (q'::int)$
apply (*frule q-pos-lemma*, *assumption*+)
apply (*subgoal-tac* $b*q < b* (q' + 1)$)
 apply (*simp add*: *mult-less-cancel-left*)
apply (*subgoal-tac* $b*q = r' - r + b'*q'$)
 prefer *2* **apply** *simp*
apply (*simp* (*no-asm-simp*) *add*: *right-distrib*)
apply (*subst add-commute*, *rule zadd-zless-mono*, *arith*)
apply (*rule mult-right-mono*, *auto*)
done

lemma *zdiv-mono2*:

```
    [| (0::int) ≤ a;  0 < b';  b' ≤ b |] ==> a div b ≤ a div b'
apply (subgoal-tac b ≠ 0)
 prefer 2 apply arith
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (cut-tac a = a and b = b' in zmod-zdiv-equality)
apply (rule zdiv-mono2-lemma)
apply (erule subst)
apply (erule subst, simp-all)
done

lemma q-neg-lemma:
    [| b'*q' + r' < 0;  0 ≤ r';  0 < b' |] ==> q' ≤ (0::int)
apply (subgoal-tac b'*q' < 0)
 apply (simp add: mult-less-0-iff, arith)
done

lemma zdiv-mono2-neg-lemma:
    [| b*q + r = b'*q' + r';  b'*q' + r' < 0;
       r < b;  0 ≤ r';  0 < b';  b' ≤ b |]
    ==> q' ≤ (q::int)
apply (frule q-neg-lemma, assumption+)
apply (subgoal-tac b*q' < b* (q + 1) )
 apply (simp add: mult-less-cancel-left)
apply (simp add: right-distrib)
apply (subgoal-tac b*q' ≤ b'*q')
 prefer 2 apply (simp add: mult-right-mono-neg, arith)
done

lemma zdiv-mono2-neg:
    [| a < (0::int);  0 < b';  b' ≤ b |] ==> a div b' ≤ a div b
apply (cut-tac a = a and b = b in zmod-zdiv-equality)
apply (cut-tac a = a and b = b' in zmod-zdiv-equality)
apply (rule zdiv-mono2-neg-lemma)
apply (erule subst)
apply (erule subst, simp-all)
done
```

31.11 More Algebraic Laws for div and mod

proving (a*b) div c = a * (b div c) + a * (b mod c)

```
lemma zmult1-lemma:
    [| divmod-rel b c (q, r);  c ≠ 0 |]
    ==> divmod-rel (a * b) c (a*q + a*r div c, a*r mod c)
by (force simp add: split-ifs divmod-rel-def linorder-neq-iff right-distrib)

lemma zdiv-zmult1-eq: (a*b) div c = a*(b div c) + a*(b mod c) div (c::int)
apply (case-tac c = 0, simp)
apply (blast intro: divmod-rel-div-mod [THEN zmult1-lemma, THEN divmod-rel-div])
done
```

lemma *zmod-zmult1-eq*: $(a*b)\ mod\ c = a*(b\ mod\ c)\ mod\ (c::int)$
apply (*case-tac* $c = 0$, *simp*)
apply (*blast intro*: *divmod-rel-div-mod* [*THEN zmult1-lemma*, *THEN divmod-rel-mod*])
done

lemma *zmod-zdiv-trivial*: $(a\ mod\ b)\ div\ b = (0::int)$
apply (*case-tac* $b = 0$, *simp*)
apply (*auto simp add*: *linorder-neq-iff div-pos-pos-trivial div-neg-neg-trivial*)
done

proving (a+b) div c = a div c + b div c + ((a mod c + b mod c) div c)

lemma *zadd1-lemma*:
[| *divmod-rel a c* (*aq*, *ar*); *divmod-rel b c* (*bq*, *br*); $c \neq 0$ |]
==> *divmod-rel* $(a+b)\ c\ (aq + bq + (ar+br)\ div\ c, (ar+br)\ mod\ c)$
by (*force simp add*: *split-ifs divmod-rel-def linorder-neq-iff right-distrib*)

lemma *zdiv-zadd1-eq*:
$(a+b)\ div\ (c::int) = a\ div\ c + b\ div\ c + ((a\ mod\ c + b\ mod\ c)\ div\ c)$
apply (*case-tac* $c = 0$, *simp*)
apply (*blast intro*: *zadd1-lemma* [*OF divmod-rel-div-mod divmod-rel-div-mod*] *divmod-rel-div*)
done

instance *int* :: *ring-div*
proof
fix *a b c* :: *int*
assume *not0*: $b \neq 0$
show $(a + c * b)\ div\ b = c + a\ div\ b$
unfolding *zdiv-zadd1-eq* [*of a c* $* b$] **using** *not0*
by (*simp add*: *zmod-zmult1-eq zmod-zdiv-trivial zdiv-zmult1-eq*)
qed *auto*

lemma *posDivAlg-div-mod*:
assumes $k \geq 0$
and $l \geq 0$
shows *posDivAlg k l* = (*k div l*, *k mod l*)
proof (*cases* $l = 0$)
case *True* **then show** *?thesis* **by** (*simp add*: *posDivAlg.simps*)
next
case *False* **with** *assms posDivAlg-correct*
have *divmod-rel k l* (*fst* (*posDivAlg k l*), *snd* (*posDivAlg k l*))
by *simp*
from *divmod-rel-div* [*OF this* ‹$l \neq 0$›] *divmod-rel-mod* [*OF this* ‹$l \neq 0$›]
show *?thesis* **by** *simp*
qed

lemma *negDivAlg-div-mod*:
assumes $k < 0$

and $l > 0$
shows *negDivAlg k l* = (*k div l*, *k mod l*)
proof −
from *assms* **have** $l \neq 0$ **by** *simp*
from *assms negDivAlg-correct*
have *divmod-rel k l* (*fst* (*negDivAlg k l*), *snd* (*negDivAlg k l*))
by *simp*
from *divmod-rel-div* [*OF this* ‹$l \neq 0$›] *divmod-rel-mod* [*OF this* ‹$l \neq 0$›]
show *?thesis* **by** *simp*
qed

lemma *zmod-eq-0-iff*: (*m mod d* = 0) = (*EX q*::*int*. *m* = *d*∗*q*)
by (*simp add*: *dvd-eq-mod-eq-0* [*symmetric*] *dvd-def*)

lemmas *zmod-eq-0D* [*dest*!] = *zmod-eq-0-iff* [*THEN iffD1*]

31.12 Proving *a div* (*b* ∗ *c*) = *a div b div c*

first, four lemmas to bound the remainder for the cases b¡0 and b¿0

lemma *zmult2-lemma-aux1*: [| (0::*int*) < *c*; *b* < *r*; *r* ≤ 0 |] ==> *b*∗*c* < *b*∗(*q mod c*) + *r*
apply (*subgoal-tac b* ∗ (*c* − *q mod c*) < *r* ∗ 1)
apply (*simp add*: *algebra-simps*)
apply (*rule order-le-less-trans*)
apply (*erule-tac* [2] *mult-strict-right-mono*)
apply (*rule mult-left-mono-neg*)
using *add1-zle-eq*[*of q mod c*]**apply**(*simp add*: *algebra-simps pos-mod-bound*)
apply (*simp*)
apply (*simp*)
done

lemma *zmult2-lemma-aux2*:
[| (0::*int*) < *c*; *b* < *r*; *r* ≤ 0 |] ==> *b* ∗ (*q mod c*) + *r* ≤ 0
apply (*subgoal-tac b* ∗ (*q mod c*) ≤ 0)
apply *arith*
apply (*simp add*: *mult-le-0-iff*)
done

lemma *zmult2-lemma-aux3*: [| (0::*int*) < *c*; 0 ≤ *r*; *r* < *b* |] ==> 0 ≤ *b* ∗ (*q mod c*) + *r*
apply (*subgoal-tac* 0 ≤ *b* ∗ (*q mod c*))
apply *arith*
apply (*simp add*: *zero-le-mult-iff*)
done

lemma *zmult2-lemma-aux4*: [| (0::*int*) < *c*; 0 ≤ *r*; *r* < *b* |] ==> *b* ∗ (*q mod c*) + *r* < *b* ∗ *c*
apply (*subgoal-tac r* ∗ 1 < *b* ∗ (*c* − *q mod c*))

apply (*simp add*: *right-diff-distrib*)
apply (*rule order-less-le-trans*)
apply (*erule mult-strict-right-mono*)
apply (*rule-tac* [*2*] *mult-left-mono*)
apply *simp*
using *add1-zle-eq*[*of q mod c*] **apply** (*simp add*: *algebra-simps pos-mod-bound*)
apply *simp*
done

lemma *zmult2-lemma*: [| *divmod-rel a b* (*q*, *r*); *b* ≠ *0*; *0* < *c* |]
==> *divmod-rel a* (*b* * *c*) (*q div c*, *b**(*q mod c*) + *r*)
by (*auto simp add*: *mult-ac divmod-rel-def linorder-neq-iff*
zero-less-mult-iff right-distrib [*symmetric*]
zmult2-lemma-aux1 zmult2-lemma-aux2 zmult2-lemma-aux3
zmult2-lemma-aux4)

lemma *zdiv-zmult2-eq*: (*0*::*int*) < *c* ==> *a div* (*b***c*) = (*a div b*) *div c*
apply (*case-tac b* = *0*, *simp*)
apply (*force simp add*: *divmod-rel-div-mod* [*THEN zmult2-lemma*, *THEN divmod-rel-div*])
done

lemma *zmod-zmult2-eq*:
(*0*::*int*) < *c* ==> *a mod* (*b***c*) = *b**(*a div b mod c*) + *a mod b*
apply (*case-tac b* = *0*, *simp*)
apply (*force simp add*: *divmod-rel-div-mod* [*THEN zmult2-lemma*, *THEN divmod-rel-mod*])
done

31.13 Cancellation of Common Factors in div

lemma *zdiv-zmult-zmult1-aux1*:
[| (*0*::*int*) < *b*; *c* ≠ *0* |] ==> (*c***a*) *div* (*c***b*) = *a div b*
by (*subst zdiv-zmult2-eq*, *auto*)

lemma *zdiv-zmult-zmult1-aux2*:
[| *b* < (*0*::*int*); *c* ≠ *0* |] ==> (*c***a*) *div* (*c***b*) = *a div b*
apply (*subgoal-tac* (*c* * (−*a*)) *div* (*c* * (−*b*)) = (−*a*) *div* (−*b*))
apply (*rule-tac* [*2*] *zdiv-zmult-zmult1-aux1*, *auto*)
done

lemma *zdiv-zmult-zmult1*: *c* ≠ (*0*::*int*) ==> (*c***a*) *div* (*c***b*) = *a div b*
apply (*case-tac b* = *0*, *simp*)
apply (*auto simp add*: *linorder-neq-iff zdiv-zmult-zmult1-aux1 zdiv-zmult-zmult1-aux2*)
done

lemma *zdiv-zmult-zmult1-if*[*simp*]:
(*k***m*) *div* (*k***n*) = (*if k* = (*0*::*int*) *then 0 else m div n*)
by (*simp add*:*zdiv-zmult-zmult1*)

31.14 Distribution of Factors over mod

lemma *zmod-zmult-zmult1-aux1*:
[| (*0*::*int*) < *b*; *c* ≠ *0* |] ==> (*c*∗*a*) *mod* (*c*∗*b*) = *c* ∗ (*a mod b*)
by (*subst zmod-zmult2-eq*, *auto*)

lemma *zmod-zmult-zmult1-aux2*:
[| *b* < (*0*::*int*); *c* ≠ *0* |] ==> (*c*∗*a*) *mod* (*c*∗*b*) = *c* ∗ (*a mod b*)
apply (*subgoal-tac* (*c* ∗ (−*a*)) *mod* (*c* ∗ (−*b*)) = *c* ∗ ((−*a*) *mod* (−*b*)))
apply (*rule-tac* [*2*] *zmod-zmult-zmult1-aux1*, *auto*)
done

lemma *zmod-zmult-zmult1*: (*c*∗*a*) *mod* (*c*∗*b*) = (*c*::*int*) ∗ (*a mod b*)
apply (*case-tac b = 0*, *simp*)
apply (*case-tac c = 0*, *simp*)
apply (*auto simp add*: *linorder-neg-iff zmod-zmult-zmult1-aux1 zmod-zmult-zmult1-aux2*)
done

lemma *zmod-zmult-zmult2*: (*a*∗*c*) *mod* (*b*∗*c*) = (*a mod b*) ∗ (*c*::*int*)
apply (*cut-tac c = c* **in** *zmod-zmult-zmult1*)
apply (*auto simp add*: *mult-commute*)
done

31.15 Splitting Rules for div and mod

The proofs of the two lemmas below are essentially identical

lemma *split-pos-lemma*:
0<*k* ==>
P(*n div k* :: *int*)(*n mod k*) = (∀ *i j*. *0*≤*j* & *j*<*k* & *n* = *k*∗*i* + *j* −−> *P i j*)
apply (*rule iffI*, *clarify*)
apply (*erule-tac P*=*P ?x ?y* **in** *rev-mp*)
apply (*subst mod-add-eq*)
apply (*subst zdiv-zadd1-eq*)
apply (*simp add*: *div-pos-pos-trivial mod-pos-pos-trivial*)

converse direction

apply (*drule-tac x = n div k* **in** *spec*)
apply (*drule-tac x = n mod k* **in** *spec*, *simp*)
done

lemma *split-neg-lemma*:
k<*0* ==>
P(*n div k* :: *int*)(*n mod k*) = (∀ *i j*. *k*<*j* & *j*≤*0* & *n* = *k*∗*i* + *j* −−> *P i j*)
apply (*rule iffI*, *clarify*)
apply (*erule-tac P*=*P ?x ?y* **in** *rev-mp*)
apply (*subst mod-add-eq*)
apply (*subst zdiv-zadd1-eq*)
apply (*simp add*: *div-neg-neg-trivial mod-neg-neg-trivial*)

converse direction

apply (*drule-tac* $x = n\ div\ k$ **in** *spec*)
apply (*drule-tac* $x = n\ mod\ k$ **in** *spec*, *simp*)
done

lemma *split-zdiv*:
$P(n\ div\ k :: int) =$
$((k = 0 \longrightarrow P\ 0)\ \&$
$(0<k \longrightarrow (\forall i\ j.\ 0\leq j\ \&\ j<k\ \&\ n = k*i + j \longrightarrow P\ i))\ \&$
$(k<0 \longrightarrow (\forall i\ j.\ k<j\ \&\ j\leq 0\ \&\ n = k*i + j \longrightarrow P\ i)))$
apply (*case-tac* $k=0$, *simp*)
apply (*simp only*: *linorder-neq-iff*)
apply (*erule disjE*)
apply (*simp-all add*: *split-pos-lemma* [*of concl*: $\%x\ y.\ P\ x$]
split-neg-lemma [*of concl*: $\%x\ y.\ P\ x$])
done

lemma *split-zmod*:
$P(n\ mod\ k :: int) =$
$((k = 0 \longrightarrow P\ n)\ \&$
$(0<k \longrightarrow (\forall i\ j.\ 0\leq j\ \&\ j<k\ \&\ n = k*i + j \longrightarrow P\ j))\ \&$
$(k<0 \longrightarrow (\forall i\ j.\ k<j\ \&\ j\leq 0\ \&\ n = k*i + j \longrightarrow P\ j)))$
apply (*case-tac* $k=0$, *simp*)
apply (*simp only*: *linorder-neq-iff*)
apply (*erule disjE*)
apply (*simp-all add*: *split-pos-lemma* [*of concl*: $\%x\ y.\ P\ y$]
split-neg-lemma [*of concl*: $\%x\ y.\ P\ y$])
done

declare *split-zdiv* [*of* - - *number-of k*, *simplified*, *standard*, *arith-split*]
declare *split-zmod* [*of* - - *number-of k*, *simplified*, *standard*, *arith-split*]

31.16 Speeding up the Division Algorithm with Shifting

computing div by shifting

lemma *pos-zdiv-mult-2*: $(0::int) \leq a \Longrightarrow (1 + 2*b)\ div\ (2*a) = b\ div\ a$
proof *cases*
assume $a=0$
thus *?thesis* **by** *simp*
next
assume $a\neq 0$ **and** *le-a*: $0\leq a$
hence *a-pos*: $1 \leq a$ **by** *arith*
hence *one-less-a2*: $1 < 2 * a$ **by** *arith*
hence *le-2a*: $2 * (1 + b\ mod\ a) \leq 2 * a$
unfolding *mult-le-cancel-left*
by (*simp add*: *add1-zle-eq add-commute* [*of 1*])
with *a-pos* **have** $0 \leq b\ mod\ a$ **by** *simp*
hence *le-addm*: $0 \leq 1\ mod\ (2*a) + 2*(b\ mod\ a)$
by (*simp add*: *mod-pos-pos-trivial one-less-a2*)

with *le-2a*
have *(1 mod (2∗a) + 2∗(b mod a)) div (2∗a) = 0*
by (*simp add*: *div-pos-pos-trivial le-addm mod-pos-pos-trivial one-less-a2 right-distrib*)
thus *?thesis*
by (*subst zdiv-zadd1-eq*,
simp add: *zdiv-zmult-zmult1 zmod-zmult-zmult1 one-less-a2 div-pos-pos-trivial*)
qed

lemma *neg-zdiv-mult-2*: *a ≤ (0::int) ==> (1 + 2∗b) div (2∗a) = (b+1) div a*
apply (*subgoal-tac (1 + 2∗ (−b − 1)) div (2 ∗ (−a)) = (−b − 1) div (−a)*)
apply (*rule-tac* [*2*] *pos-zdiv-mult-2*)
apply (*auto simp add*: *minus-mult-right* [*symmetric*] *right-diff-distrib*)
apply (*subgoal-tac (−1 − (2 ∗ b)) = − (1 + (2 ∗ b))*)
apply (*simp only*: *zdiv-zminus-zminus diff-minus minus-add-distrib* [*symmetric*], *simp*)
done

lemma *zdiv-number-of-Bit0* [*simp*]:
number-of (Int.Bit0 v) div number-of (Int.Bit0 w) =
number-of v div (number-of w :: int)
by (*simp only*: *number-of-eq numeral-simps*) *simp*

lemma *zdiv-number-of-Bit1* [*simp*]:
number-of (Int.Bit1 v) div number-of (Int.Bit0 w) =
(if (0::int) ≤ number-of w
then number-of v div (number-of w)
else (number-of v + (1::int)) div (number-of w))
apply (*simp only*: *number-of-eq numeral-simps UNIV-I split*: *split-if*)
apply (*simp add*: *zdiv-zmult-zmult1 pos-zdiv-mult-2 neg-zdiv-mult-2 add-ac*)
done

31.17 Computing mod by Shifting (proofs resemble those for div)

lemma *pos-zmod-mult-2*:
(0::int) ≤ a ==> (1 + 2∗b) mod (2∗a) = 1 + 2 ∗ (b mod a)
apply (*case-tac a = 0*, *simp*)
apply (*subgoal-tac 1 < a ∗ 2*)
prefer *2* **apply** *arith*
apply (*subgoal-tac 2∗ (1 + b mod a) ≤ 2∗a*)
apply (*rule-tac* [*2*] *mult-left-mono*)
apply (*auto simp add*: *add-commute* [*of 1*] *mult-commute add1-zle-eq pos-mod-bound*)
apply (*subst mod-add-eq*)
apply (*simp add*: *zmod-zmult-zmult2 mod-pos-pos-trivial*)
apply (*rule mod-pos-pos-trivial*)
apply (*auto simp add*: *mod-pos-pos-trivial ring-distribs*)

apply (*subgoal-tac* $0 \leq b \bmod a$, *arith*, *simp*)
done

lemma *neg-zmod-mult-2*:
$a \leq (0::int) \Longrightarrow (1 + 2*b) \bmod (2*a) = 2 * ((b+1) \bmod a) - 1$
apply (*subgoal-tac* $(1 + 2* (-b - 1)) \bmod (2* (-a)) =$
$1 + 2* ((-b - 1) \bmod (-a))$)
apply (*rule-tac* [2] *pos-zmod-mult-2*)
apply (*auto simp add*: *right-diff-distrib*)
apply (*subgoal-tac* $(-1 - (2 * b)) = - (1 + (2 * b))$)
prefer 2 **apply** *simp*
apply (*simp only*: *zmod-zminus-zminus diff-minus minus-add-distrib* [*symmetric*])
done

lemma *zmod-number-of-Bit0* [*simp*]:
number-of (*Int.Bit0 v*) *mod number-of* (*Int.Bit0 w*) =
(2::*int*) * (*number-of v mod number-of w*)
apply (*simp only*: *number-of-eq numeral-simps*)
apply (*simp add*: *zmod-zmult-zmult1 pos-zmod-mult-2*
neg-zmod-mult-2 add-ac)
done

lemma *zmod-number-of-Bit1* [*simp*]:
number-of (*Int.Bit1 v*) *mod number-of* (*Int.Bit0 w*) =
(*if* $(0::int) \leq$ *number-of w*
then 2 * (*number-of v mod number-of w*) + 1
else 2 * ((*number-of v* + (1::*int*)) *mod number-of w*) - 1)
apply (*simp only*: *number-of-eq numeral-simps*)
apply (*simp add*: *zmod-zmult-zmult1 pos-zmod-mult-2*
neg-zmod-mult-2 add-ac)
done

31.18 Quotients of Signs

lemma *div-neg-pos-less0*: [| $a < (0::int)$; $0 < b$ |] ==> $a \operatorname{div} b < 0$
apply (*subgoal-tac* $a \operatorname{div} b \leq -1$, *force*)
apply (*rule order-trans*)
apply (*rule-tac* $a' = -1$ **in** *zdiv-mono1*)
apply (*auto simp add*: *div-eq-minus1*)
done

lemma *div-nonneg-neg-le0*: [| $(0::int) \leq a$; $b < 0$ |] ==> $a \operatorname{div} b \leq 0$
by (*drule zdiv-mono1-neg*, *auto*)

lemma *div-nonpos-pos-le0*: [| $(a::int) \leq 0$; $b > 0$ |] ==> $a \operatorname{div} b \leq 0$
by (*drule zdiv-mono1*, *auto*)

lemma *pos-imp-zdiv-nonneg-iff*: $(0::int) < b \Longrightarrow (0 \leq a \operatorname{div} b) = (0 \leq a)$
apply *auto*

apply (*drule-tac* [*2*] *zdiv-mono1*)
apply (*auto simp add*: *linorder-neq-iff*)
apply (*simp* (*no-asm-use*) *add*: *linorder-not-less* [*symmetric*])
apply (*blast intro*: *div-neg-pos-less0*)
done

lemma *neg-imp-zdiv-nonneg-iff*:
  $b < (0::int) \Longrightarrow (0 \leq a \ div\ b) = (a \leq (0::int))$
apply (*subst zdiv-zminus-zminus* [*symmetric*])
apply (*subst pos-imp-zdiv-nonneg-iff*, *auto*)
done

lemma *pos-imp-zdiv-neg-iff*: $(0::int) < b \Longrightarrow (a \ div\ b < 0) = (a < 0)$
by (*simp add*: *linorder-not-le* [*symmetric*] *pos-imp-zdiv-nonneg-iff*)

lemma *neg-imp-zdiv-neg-iff*: $b < (0::int) \Longrightarrow (a \ div\ b < 0) = (0 < a)$
by (*simp add*: *linorder-not-le* [*symmetric*] *neg-imp-zdiv-nonneg-iff*)

31.19 The Divides Relation

lemmas *zdvd-iff-zmod-eq-0-number-of* [*simp*] =
  dvd-eq-mod-eq-0 [*of number-of x*::*int number-of y*::*int*, *standard*]

lemma *zdvd-anti-sym*:
  $0 < m \Longrightarrow 0 < n \Longrightarrow m\ dvd\ n \Longrightarrow n\ dvd\ m \Longrightarrow m = (n::int)$
 apply (*simp add*: *dvd-def*, *auto*)
 apply (*simp add*: *mult-assoc zero-less-mult-iff zmult-eq-1-iff*)
 done

lemma *zdvd-dvd-eq*: **assumes** $a \neq 0$ **and** $(a::int)\ dvd\ b$ **and** $b\ dvd\ a$
 shows $|a| = |b|$
proof−
 from ‹*a dvd b*› **obtain** *k* **where** *k*:$b = a*k$ **unfolding** *dvd-def* **by** *blast*
 from ‹*b dvd a*› **obtain** k' **where** k':$a = b*k'$ **unfolding** *dvd-def* **by** *blast*
 from *k k'* **have** $a = a*k*k'$ **by** *simp*
 with *mult-cancel-left1*[**where** *c*=*a* **and** *b*=$k*k'$]
 have *kk'*:$k*k' = 1$ **using** ‹$a \neq 0$› **by** (*simp add*: *mult-assoc*)
 hence $k = 1 \wedge k' = 1 \vee k = -1 \wedge k' = -1$ **by** (*simp add*: *zmult-eq-1-iff*)
 thus *?thesis* **using** *k k'* **by** *auto*
qed

lemma *zdvd-zdiffD*: *k dvd m* − *n* ⟹ *k dvd n* ⟹ *k dvd* $(m::int)$
 apply (*subgoal-tac* $m = n + (m - n)$)
  apply (*erule ssubst*)
  apply (*blast intro*: *dvd-add*, *simp*)
 done

lemma *zdvd-reduce*: $(k\ dvd\ n + k * m) = (k\ dvd\ (n::int))$
apply (*rule iffI*)
apply (*erule-tac* [*2*] *dvd-add*)
apply (*subgoal-tac* $n = (n + k * m) - k * m$)
apply (*erule ssubst*)
apply (*erule dvd-diff*)
apply(*simp-all*)
done

lemma *zdvd-zmod*: $f\ dvd\ m \Longrightarrow f\ dvd\ (n::int) \Longrightarrow f\ dvd\ m\ mod\ n$
apply (*simp add*: *dvd-def*)
apply (*auto simp add*: *zmod-zmult-zmult1*)
done

lemma *zdvd-zmod-imp-zdvd*: $k\ dvd\ m\ mod\ n \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ (m::int)$
apply (*subgoal-tac* $k\ dvd\ n * (m\ div\ n) + m\ mod\ n$)
apply (*simp add*: *zmod-zdiv-equality* [*symmetric*])
apply (*simp only*: *dvd-add dvd-mult2*)
done

lemma *zdvd-not-zless*: $0 < m \Longrightarrow m < n \Longrightarrow \neg\ n\ dvd\ (m::int)$
apply (*auto elim*!: *dvdE*)
apply (*subgoal-tac* $0 < n$)
prefer *2*
apply (*blast intro*: *order-less-trans*)
apply (*simp add*: *zero-less-mult-iff*)
apply (*subgoal-tac* $n * k < n * 1$)
apply (*drule mult-less-cancel-left* [*THEN iffD1*], *auto*)
done

lemma *zmult-div-cancel*: $(n::int) * (m\ div\ n) = m - (m\ mod\ n)$
using *zmod-zdiv-equality*[**where** $a{=}m$ **and** $b{=}n$]
by (*simp add*: *algebra-simps*)

lemma *zdvd-mult-div-cancel*:$(n::int)\ dvd\ m \Longrightarrow n * (m\ div\ n) = m$
apply (*subgoal-tac* $m\ mod\ n = 0$)
apply (*simp add*: *zmult-div-cancel*)
apply (*simp only*: *dvd-eq-mod-eq-0*)
done

lemma *zdvd-mult-cancel*: **assumes** $d{:}k * m\ dvd\ k * n$ **and** $kz{:}k \neq (0::int)$
shows $m\ dvd\ n$
proof−
from d **obtain** h **where** $h{:}\ k{*}n = k{*}m * h$ **unfolding** *dvd-def* **by** *blast*
{assume $n \neq m{*}h$ **hence** $k{*}\ n \neq k{*}\ (m{*}h)$ **using** kz **by** *simp*
with h **have** *False* **by** (*simp add*: *mult-assoc*)**}**
hence $n = m * h$ **by** *blast*
thus *?thesis* **by** *simp*
qed

theorem *ex-nat*: $(\exists\, x{::}nat.\ P\ x) = (\exists\, x{::}int.\ 0 <= x \wedge P\ (nat\ x))$
apply (*simp split add*: *split-nat*)
apply (*rule iffI*)
apply (*erule exE*)
apply (*rule-tac* $x = int\ x$ **in** *exI*)
apply *simp*
apply (*erule exE*)
apply (*rule-tac* $x = nat\ x$ **in** *exI*)
apply (*erule conjE*)
apply (*erule-tac* $x = nat\ x$ **in** *allE*)
apply *simp*
done

theorem *zdvd-int*: $(x\ dvd\ y) = (int\ x\ dvd\ int\ y)$
proof −
  have $\bigwedge k.\ int\ y = int\ x * k \Longrightarrow x\ dvd\ y$
  proof −
    fix *k*
    assume *A*: $int\ y = int\ x * k$
    then show $x\ dvd\ y$ **proof** (*cases k*)
      case (*1 n*) **with** *A* **have** $y = x * n$ **by** (*simp add*: *zmult-int*)
      then show *?thesis* **..**
    next
      case (*2 n*) **with** *A* **have** $int\ y = int\ x * (-\ int\ (Suc\ n))$ **by** *simp*
      also have $\ldots = -\ (int\ x * int\ (Suc\ n))$ **by** (*simp only*: *mult-minus-right*)
      also have $\ldots = -\ int\ (x * Suc\ n)$ **by** (*simp only*: *zmult-int*)
      finally have $-\ int\ (x * Suc\ n) = int\ y$ **..**
      then show *?thesis* **by** (*simp only*: *negative-eq-positive*) *auto*
    qed
  qed
  then show *?thesis* **by** (*auto elim*!: *dvdE simp only*: *dvd-triv-left int-mult*)
qed

lemma *zdvd1-eq*[*simp*]: $(x{::}int)\ dvd\ 1 = (\ |x| = 1)$
proof
  assume *d*: $x\ dvd\ 1$ **hence** $int\ (nat\ |x|)\ dvd\ int\ (nat\ 1)$ **by** *simp*
  hence $nat\ |x|\ dvd\ 1$ **by** (*simp add*: *zdvd-int*)
  hence $nat\ |x| = 1$ **by** *simp*
  thus $|x| = 1$ **by** (*cases* $x < 0$, *auto*)
next
  assume $|x|{=}1$ **thus** $x\ dvd\ 1$
    by(*cases* $x < 0$,*simp-all add*: *minus-equation-iff dvd-eq-mod-eq-0*)
qed
lemma *zdvd-mult-cancel1*:
  assumes *mp*:$m \neq (0{::}int)$ **shows** $(m * n\ dvd\ m) = (|n| = 1)$
proof
  assume *n1*: $|n| = 1$ **thus** $m * n\ dvd\ m$

by (*cases n >0*, *auto simp add*: *minus-dvd-iff minus-equation-iff*)
next
assume *H*: *m * n dvd m* **hence** *H2*: *m * n dvd m * 1* **by** *simp*
from *zdvd-mult-cancel*[*OF H2 mp*] **show** |*n*| = *1* **by** (*simp only*: *zdvd1-eq*)
qed

lemma *int-dvd-iff*: (*int m dvd z*) = (*m dvd nat* (*abs z*))
unfolding *zdvd-int* **by** (*cases z ≥ 0*) *simp-all*

lemma *dvd-int-iff*: (*z dvd int m*) = (*nat* (*abs z*) *dvd m*)
unfolding *zdvd-int* **by** (*cases z ≥ 0*) *simp-all*

lemma *nat-dvd-iff*: (*nat z dvd m*) = (*if 0 ≤ z then* (*z dvd int m*) *else m = 0*)
by (*auto simp add*: *dvd-int-iff*)

lemma *zdvd-imp-le*: [| *z dvd n*; *0 < n* |] ==> *z ≤* (*n*::*int*)
apply (*rule-tac z*=*n* **in** *int-cases*)
apply (*auto simp add*: *dvd-int-iff*)
apply (*rule-tac z*=*z* **in** *int-cases*)
apply (*auto simp add*: *dvd-imp-le*)
done

lemma *zpower-zmod*: ((*x*::*int*) *mod m*) ^*y mod m* = *x* ^*y mod m*
apply (*induct y*, *auto*)
apply (*rule zmod-zmult1-eq* [*THEN trans*])
apply (*simp* (*no-asm-simp*))
apply (*rule mod-mult-eq* [*symmetric*])
done

lemma *zdiv-int*: *int* (*a div b*) = (*int a*) *div* (*int b*)
apply (*subst split-div*, *auto*)
apply (*subst split-zdiv*, *auto*)
apply (*rule-tac a*=*int* (*b * i*) + *int j* **and** *b*=*int b* **and** *r*=*int j* **and** *r′*=*ja* **in**
IntDiv.unique-quotient)
apply (*auto simp add*: *IntDiv.divmod-rel-def of-nat-mult*)
done

lemma *zmod-int*: *int* (*a mod b*) = (*int a*) *mod* (*int b*)
apply (*subst split-mod*, *auto*)
apply (*subst split-zmod*, *auto*)
apply (*rule-tac a*=*int* (*b * i*) + *int j* **and** *b*=*int b* **and** *q*=*int i* **and** *q′*=*ia*
in *unique-remainder*)
apply (*auto simp add*: *IntDiv.divmod-rel-def of-nat-mult*)
done

lemma *abs-div*: (*y*::*int*) *dvd x* ⟹ *abs* (*x div y*) = *abs x div abs y*
by (*unfold dvd-def*, *cases y*=*0*, *auto simp add*: *abs-mult*)

Suggested by Matthias Daum

lemma *int-power-div-base*:
  $[\![0 < m;\ 0 < k]\!] \Longrightarrow k \hat{}\ m\ div\ k = (k::int) \hat{}\ (m - Suc\ 0)$
apply (*subgoal-tac* $k \hat{}\ m = k \hat{}\ ((m - Suc\ 0) + Suc\ 0)$)
 apply (*erule ssubst*)
 apply (*simp only*: *power-add*)
 apply *simp-all*
done

by Brian Huffman

lemma *zminus-zmod*: $-((x::int)\ mod\ m)\ mod\ m = -\ x\ mod\ m$
by (*rule mod-minus-eq* [*symmetric*])

lemma *zdiff-zmod-left*: $(x\ mod\ m - y)\ mod\ m = (x - y)\ mod\ (m::int)$
by (*rule mod-diff-left-eq* [*symmetric*])

lemma *zdiff-zmod-right*: $(x - y\ mod\ m)\ mod\ m = (x - y)\ mod\ (m::int)$
by (*rule mod-diff-right-eq* [*symmetric*])

lemmas *zmod-simps* =
  mod-add-left-eq [*symmetric*]
  mod-add-right-eq [*symmetric*]
  IntDiv.zmod-zmult1-eq [*symmetric*]
  mod-mult-left-eq [*symmetric*]
  IntDiv.zpower-zmod
  zminus-zmod zdiff-zmod-left zdiff-zmod-right

Distributive laws for function *nat*.

lemma *nat-div-distrib*: $0 \le x \Longrightarrow nat\ (x\ div\ y) = nat\ x\ div\ nat\ y$
apply (*rule linorder-cases* [*of y 0*])
apply (*simp add*: *div-nonneg-neg-le0*)
apply *simp*
apply (*simp add*: *nat-eq-iff pos-imp-zdiv-nonneg-iff zdiv-int*)
done

lemma *nat-mod-distrib*:
  $[\![0 \le x;\ 0 \le y]\!] \Longrightarrow nat\ (x\ mod\ y) = nat\ x\ mod\ nat\ y$
apply (*case-tac* $y = 0$, *simp add*: *DIVISION-BY-ZERO*)
apply (*simp add*: *nat-eq-iff zmod-int*)
done

Suggested by Matthias Daum

lemma *int-div-less-self*: $[\![0 < x;\ 1 < k]\!] \Longrightarrow x\ div\ k < (x::int)$
apply (*subgoal-tac nat x div nat k* $<$ *nat x*)
 apply (*simp* (*asm-lr*) *add*: *nat-div-distrib* [*symmetric*])
apply (*rule Divides.div-less-dividend*, *simp-all*)
done

code generator setup

context *ring-1*
begin

lemma *of-int-num* [*code*]:
  of-int k = (*if k* = *0 then 0 else if k* < *0 then*
    − *of-int* (− *k*) *else let*
      (*l*, *m*) = *divmod k 2*;
      l' = *of-int l*
    in if m = *0 then l'* + *l'* *else l'* + *l'* + *1*)
proof −
  have *aux1*: *k mod* (*2*::*int*) ≠ (*0*::*int*) ⟹
    of-int k = *of-int* (*k div 2* ∗ *2* + *1*)
  proof −
    have *k mod 2* < *2* **by** (*auto intro*: *pos-mod-bound*)
    moreover have *0* ≤ *k mod 2* **by** (*auto intro*: *pos-mod-sign*)
    moreover assume *k mod 2* ≠ *0*
    ultimately have *k mod 2* = *1* **by** *arith*
    moreover have *of-int k* = *of-int* (*k div 2* ∗ *2* + *k mod 2*) **by** *simp*
    ultimately show *?thesis* **by** *auto*
  qed
  have *aux2*: ⋀*x*. *of-int 2* ∗ *x* = *x* + *x*
  proof −
    fix *x*
    have *int2*: (*2*::*int*) = *1* + *1* **by** *arith*
    show *of-int 2* ∗ *x* = *x* + *x*
    unfolding *int2 of-int-add left-distrib* **by** *simp*
  qed
  have *aux3*: ⋀*x*. *x* ∗ *of-int 2* = *x* + *x*
  proof −
    fix *x*
    have *int2*: (*2*::*int*) = *1* + *1* **by** *arith*
    show *x* ∗ *of-int 2* = *x* + *x*
    unfolding *int2 of-int-add right-distrib* **by** *simp*
  qed
  from *aux1* **show** *?thesis* **by** (*auto simp add*: *divmod-mod-div Let-def aux2 aux3*)
qed

end

lemma *zmod-eq-dvd-iff*: (*x*::*int*) *mod n* = *y mod n* ⟷ *n dvd x* − *y*
proof
  assume *H*: *x mod n* = *y mod n*
  hence *x mod n* − *y mod n* = *0* **by** *simp*
  hence (*x mod n* − *y mod n*) *mod n* = *0* **by** *simp*
  hence (*x* − *y*) *mod n* = *0* **by** (*simp add*: *mod-diff-eq*[*symmetric*])
  thus *n dvd x* − *y* **by** (*simp add*: *dvd-eq-mod-eq-0*)
next
  assume *H*: *n dvd x* − *y*
  then obtain *k* **where** *k*: *x*−*y* = *n*∗*k* **unfolding** *dvd-def* **by** *blast*

hence $x = n*k + y$ **by** *simp*
hence $x\ mod\ n = (n*k + y)\ mod\ n$ **by** *simp*
thus $x\ mod\ n = y\ mod\ n$ **by** (*simp add*: *mod-add-left-eq*)
qed

lemma *nat-mod-eq-lemma*: **assumes** *xyn*: $(x::nat)\ mod\ n = y\ \ mod\ n$ **and** *xy*:$y \leq x$
shows $\exists\, q.\ x = y + n * q$
proof−
from *xy* **have** *th*: $int\ x - int\ y = int\ (x - y)$ **by** *simp*
from *xyn* **have** $int\ x\ mod\ int\ n = int\ y\ mod\ int\ n$
by (*simp add*: *zmod-int*[*symmetric*])
hence $int\ n\ dvd\ int\ x - int\ y$ **by** (*simp only*: *zmod-eq-dvd-iff*[*symmetric*])
hence $n\ dvd\ x - y$ **by** (*simp add*: *th zdvd-int*)
then show *?thesis* **using** *xy* **unfolding** *dvd-def* **apply** *clarsimp* **apply** (*rule-tac* $x{=}k$ **in** *exI*) **by** *arith*
qed

lemma *nat-mod-eq-iff*: $(x::nat)\ mod\ n = y\ mod\ n \longleftrightarrow (\exists\, q1\ q2.\ x + n * q1 = y + n * q2)$
(**is** *?lhs* = *?rhs*)
proof
assume *H*: $x\ mod\ n = y\ mod\ n$
{assume *xy*: $x \leq y$
from *H* **have** *th*: $y\ mod\ n = x\ mod\ n$ **by** *simp*
from *nat-mod-eq-lemma*[*OF th xy*] **have** *?rhs*
apply *clarify* **apply** (*rule-tac* $x{=}q$ **in** *exI*) **by** (*rule exI*[**where** $x{=}0$], *simp*)**}**
moreover
{assume *xy*: $y \leq x$
from *nat-mod-eq-lemma*[*OF H xy*] **have** *?rhs*
apply *clarify* **apply** (*rule-tac* $x{=}0$ **in** *exI*) **by** (*rule-tac* $x{=}q$ **in** *exI*, *simp*)**}**
ultimately show *?rhs* **using** *linear*[*of x y*] **by** *blast*
next
assume *?rhs* **then obtain** *q1 q2* **where** *q12*: $x + n * q1 = y + n * q2$ **by** *blast*
hence $(x + n * q1)\ mod\ n = (y + n * q2)\ mod\ n$ **by** *simp*
thus *?lhs* **by** *simp*
qed

31.20 Simproc setup

use *Tools/int-factor-simprocs.ML*

31.21 Code generation

definition *pdivmod* :: $int \Rightarrow int \Rightarrow int \times int$ **where**
$pdivmod\ k\ l = (|k|\ div\ |l|, |k|\ mod\ |l|)$

lemma *pdivmod-posDivAlg* [*code*]:
$pdivmod\ k\ l = (if\ l = 0\ then\ (0, |k|)\ else\ posDivAlg\ |k|\ |l|)$
by (*subst posDivAlg-div-mod*) (*simp-all add*: *pdivmod-def*)

lemma *divmod-pdivmod*: *divmod k l* = (*if k* = *0 then* (*0*, *0*) *else if l* = *0 then* (*0*, *k*) *else*
apsnd ((*op* ∗) (*sgn l*)) (*if* 0 < *l* ∧ 0 ≤ *k* ∨ *l* < 0 ∧ *k* < 0
then pdivmod k l
else (*let* (*r*, *s*) = *pdivmod k l in*
if s = *0 then* (− *r*, *0*) *else* (− *r* − *1*, |*l*| − *s*))))
proof −
have *aux*: ⋀*q*::*int*. − *k* = *l* ∗ *q* ⟷ *k* = *l* ∗ − *q* **by** *auto*
show *?thesis*
by (*simp add*: *divmod-mod-div pdivmod-def*)
(*auto simp add*: *aux not-less not-le zdiv-zminus1-eq-if*
zmod-zminus1-eq-if zdiv-zminus2-eq-if zmod-zminus2-eq-if)
qed

lemma *divmod-code* [*code*]: *divmod k l* = (*if k* = *0 then* (*0*, *0*) *else if l* = *0 then* (*0*, *k*) *else*
apsnd ((*op* ∗) (*sgn l*)) (*if sgn k* = *sgn l*
then pdivmod k l
else (*let* (*r*, *s*) = *pdivmod k l in*
if s = *0 then* (− *r*, *0*) *else* (− *r* − *1*, |*l*| − *s*))))
proof −
have *k* ≠ *0* ⟹ *l* ≠ *0* ⟹ 0 < *l* ∧ 0 ≤ *k* ∨ *l* < 0 ∧ *k* < 0 ⟷ *sgn k* = *sgn l*
by (*auto simp add*: *not-less sgn-if*)
then show *?thesis* **by** (*simp add*: *divmod-pdivmod*)
qed

code-modulename *SML*
IntDiv Integer

code-modulename *OCaml*
IntDiv Integer

code-modulename *Haskell*
IntDiv Integer

end

32 NatBin: Binary arithmetic for the natural numbers

theory *NatBin*
imports *IntDiv*
uses (*Tools/nat-simprocs.ML*)
begin

Arithmetic for naturals is reduced to that for the non-negative integers.

instantiation *nat* :: *number*
begin

definition
nat-number-of-def [*code inline*, *code del*]: *number-of v* = *nat* (*number-of v*)

instance ..

end

lemma [*code post*]:
nat (*number-of v*) = *number-of v*
unfolding *nat-number-of-def* **..**

abbreviation (*xsymbols*)
power2 :: *'a*::*power* => *'a* ((-^{2}) [*1000*] *999*) **where**
x^2 == *x^2*

notation (*latex* **output**)
power2 ((-^{2}) [*1000*] *999*)

notation (*HTML* **output**)
power2 ((-^{2}) [*1000*] *999*)

32.1 Predicate for negative binary numbers

definition *neg* :: *int* ⇒ *bool* **where**
neg Z ⟷ *Z* < *0*

lemma *not-neg-int* [*simp*]: ~ *neg* (*of-nat n*)
by (*simp add*: *neg-def*)

lemma *neg-zminus-int* [*simp*]: *neg* (− (*of-nat* (*Suc n*)))
by (*simp add*: *neg-def neg-less-0-iff-less del*: *of-nat-Suc*)

lemmas *neg-eq-less-0* = *neg-def*

lemma *not-neg-eq-ge-0*: (~*neg x*) = (*0* ≤ *x*)
by (*simp add*: *neg-def linorder-not-less*)

To simplify inequalities when Numeral1 can get simplified to 1

lemma *not-neg-0*: ~ *neg 0*
by (*simp add*: *One-int-def neg-def*)

lemma *not-neg-1*: ~ *neg 1*
by (*simp add*: *neg-def linorder-not-less zero-le-one*)

lemma *neg-nat*: *neg z* ==> *nat z* = *0*
by (*simp add*: *neg-def order-less-imp-le*)

lemma *not-neg-nat*: ~ *neg z* ==> *of-nat* (*nat z*) = *z*
by (*simp add*: *linorder-not-less neg-def*)

If *Numeral0* is rewritten to 0 then this rule can't be applied: *Numeral0* IS *number-of Pls*

lemma *not-neg-number-of-Pls*: ~ *neg* (*number-of Int.Pls*)
by (*simp add*: *neg-def*)

lemma *neg-number-of-Min*: *neg* (*number-of Int.Min*)
by (*simp add*: *neg-def*)

lemma *neg-number-of-Bit0*:
neg (*number-of* (*Int.Bit0 w*)) = *neg* (*number-of w*)
by (*simp add*: *neg-def*)

lemma *neg-number-of-Bit1*:
neg (*number-of* (*Int.Bit1 w*)) = *neg* (*number-of w*)
by (*simp add*: *neg-def*)

lemmas *neg-simps* [*simp*] =
not-neg-0 not-neg-1
not-neg-number-of-Pls neg-number-of-Min
neg-number-of-Bit0 neg-number-of-Bit1

32.2 Function *nat*: Coercion from Type *int* to *nat*

declare *nat-0* [*simp*] *nat-1* [*simp*]

lemma *nat-number-of* [*simp*]: *nat* (*number-of w*) = *number-of w*
by (*simp add*: *nat-number-of-def*)

lemma *nat-numeral-0-eq-0* [*simp*]: *Numeral0* = (*0*::*nat*)
by (*simp add*: *nat-number-of-def*)

lemma *nat-numeral-1-eq-1* [*simp*]: *Numeral1* = (*1*::*nat*)
by (*simp add*: *nat-1 nat-number-of-def*)

lemma *numeral-1-eq-Suc-0*: *Numeral1* = *Suc 0*
by (*simp add*: *nat-numeral-1-eq-1*)

lemma *numeral-2-eq-2*: *2* = *Suc* (*Suc 0*)
apply (*unfold nat-number-of-def*)
apply (*rule nat-2*)
done

32.3 Function *int*: Coercion from Type *nat* to *int*

lemma *int-nat-number-of* [*simp*]:

```
    int (number-of v) =
        (if neg (number-of v :: int) then 0
         else (number-of v :: int))
  unfolding nat-number-of-def number-of-is-id neg-def
  by simp
```

32.3.1 Successor

```
lemma Suc-nat-eq-nat-zadd1: (0::int) <= z ==> Suc (nat z) = nat (1 + z)
apply (rule sym)
apply (simp add: nat-eq-iff int-Suc)
done

lemma Suc-nat-number-of-add:
    Suc (number-of v + n) =
      (if neg (number-of v :: int) then 1+n else number-of (Int.succ v) + n)
  unfolding nat-number-of-def number-of-is-id neg-def numeral-simps
  by (simp add: Suc-nat-eq-nat-zadd1 add-ac)

lemma Suc-nat-number-of [simp]:
    Suc (number-of v) =
      (if neg (number-of v :: int) then 1 else number-of (Int.succ v))
apply (cut-tac n = 0 in Suc-nat-number-of-add)
apply (simp cong del: if-weak-cong)
done
```

32.3.2 Addition

```
lemma add-nat-number-of [simp]:
    (number-of v :: nat) + number-of v' =
        (if v < Int.Pls then number-of v'
         else if v' < Int.Pls then number-of v
         else number-of (v + v'))
  unfolding nat-number-of-def number-of-is-id numeral-simps
  by (simp add: nat-add-distrib)

lemma nat-number-of-add-1 [simp]:
  number-of v + (1::nat) =
    (if v < Int.Pls then 1 else number-of (Int.succ v))
  unfolding nat-number-of-def number-of-is-id numeral-simps
  by (simp add: nat-add-distrib)

lemma nat-1-add-number-of [simp]:
  (1::nat) + number-of v =
    (if v < Int.Pls then 1 else number-of (Int.succ v))
  unfolding nat-number-of-def number-of-is-id numeral-simps
  by (simp add: nat-add-distrib)

lemma nat-1-add-1 [simp]: 1 + 1 = (2::nat)
  by (rule int-int-eq [THEN iffD1]) simp
```

32.3.3 Subtraction

lemma *diff-nat-eq-if*:
nat z − nat z′ =
(if neg z′ then nat z
else let d = z−z′ in
if neg d then 0 else nat d)
by (*simp add*: *Let-def nat-diff-distrib* [*symmetric*] *neg-eq-less-0 not-neg-eq-ge-0*)

lemma *diff-nat-number-of* [*simp*]:
(number-of v :: nat) − number-of v′ =
(if v′ < Int.Pls then number-of v
else let d = number-of (v + uminus v′) in
if neg d then 0 else nat d)
unfolding *nat-number-of-def number-of-is-id numeral-simps neg-def*
by *auto*

lemma *nat-number-of-diff-1* [*simp*]:
number-of v − (1::nat) =
(if v ≤ Int.Pls then 0 else number-of (Int.pred v))
unfolding *nat-number-of-def number-of-is-id numeral-simps*
by *auto*

32.3.4 Multiplication

lemma *mult-nat-number-of* [*simp*]:
(number-of v :: nat) ∗ number-of v′ =
(if v < Int.Pls then 0 else number-of (v ∗ v′))
unfolding *nat-number-of-def number-of-is-id numeral-simps*
by (*simp add*: *nat-mult-distrib*)

32.3.5 Quotient

lemma *div-nat-number-of* [*simp*]:
(number-of v :: nat) div number-of v′ =
(if neg (number-of v :: int) then 0
else nat (number-of v div number-of v′))
unfolding *nat-number-of-def number-of-is-id neg-def*
by (*simp add*: *nat-div-distrib*)

lemma *one-div-nat-number-of* [*simp*]:
Suc 0 div number-of v′ = nat (1 div number-of v′)
by (*simp del*: *nat-numeral-1-eq-1 add*: *numeral-1-eq-Suc-0* [*symmetric*])

32.3.6 Remainder

lemma *mod-nat-number-of* [*simp*]:
(number-of v :: nat) mod number-of v′ =
(if neg (number-of v :: int) then 0

else if neg (*number-of v′* :: *int*) *then number-of v*
else nat (*number-of v mod number-of v′*))
unfolding *nat-number-of-def number-of-is-id neg-def*
by (*simp add*: *nat-mod-distrib*)

lemma *one-mod-nat-number-of* [*simp*]:
Suc 0 mod number-of v′ =
(*if neg* (*number-of v′* :: *int*) *then Suc 0*
else nat (*1 mod number-of v′*))
by (*simp del*: *nat-numeral-1-eq-1 add*: *numeral-1-eq-Suc-0* [*symmetric*])

32.3.7 Divisibility

lemmas *dvd-eq-mod-eq-0-number-of* =
dvd-eq-mod-eq-0 [*of number-of x number-of y*, *standard*]

declare *dvd-eq-mod-eq-0-number-of* [*simp*]

ML
⟪
val nat-number-of-def = *thmnat-number-of-def*;

val nat-number-of = *thmnat-number-of*;
val nat-numeral-0-eq-0 = *thmnat-numeral-0-eq-0*;
val nat-numeral-1-eq-1 = *thmnat-numeral-1-eq-1*;
val numeral-1-eq-Suc-0 = *thmnumeral-1-eq-Suc-0*;
val numeral-2-eq-2 = *thmnumeral-2-eq-2*;
val nat-div-distrib = *thmnat-div-distrib*;
val nat-mod-distrib = *thmnat-mod-distrib*;
val int-nat-number-of = *thmint-nat-number-of*;
val Suc-nat-eq-nat-zadd1 = *thmSuc-nat-eq-nat-zadd1*;
val Suc-nat-number-of-add = *thmSuc-nat-number-of-add*;
val Suc-nat-number-of = *thmSuc-nat-number-of*;
val add-nat-number-of = *thmadd-nat-number-of*;
val diff-nat-eq-if = *thmdiff-nat-eq-if*;
val diff-nat-number-of = *thmdiff-nat-number-of*;
val mult-nat-number-of = *thmmult-nat-number-of*;
val div-nat-number-of = *thmdiv-nat-number-of*;
val mod-nat-number-of = *thmmod-nat-number-of*;
⟫

32.4 Comparisons

32.4.1 Equals (=)

lemma *eq-nat-nat-iff*:
[| (*0*::*int*) <= *z*; *0* <= *z′* |] ==> (*nat z* = *nat z′*) = (*z*=*z′*)
by (*auto elim*!: *nonneg-eq-int*)

lemma *eq-nat-number-of* [*simp*]:

$((number\text{-}of\ v :: nat) = number\text{-}of\ v') =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ (number\text{-}of\ v' :: int) \leq 0$
$\ else\ if\ neg\ (number\text{-}of\ v' :: int)\ then\ (number\text{-}of\ v :: int) = 0$
$\ else\ v = v')$
unfolding *nat-number-of-def number-of-is-id neg-def*
by *auto*

32.4.2 Less-than (¡)

lemma *less-nat-number-of* [*simp*]:
$(number\text{-}of\ v :: nat) < number\text{-}of\ v' \longleftrightarrow$
$(if\ v < v'\ then\ Int.Pls < v'\ else\ False)$
unfolding *nat-number-of-def number-of-is-id numeral-simps*
by *auto*

32.4.3 Less-than-or-equal

lemma *le-nat-number-of* [*simp*]:
$(number\text{-}of\ v :: nat) \leq number\text{-}of\ v' \longleftrightarrow$
$(if\ v \leq v'\ then\ True\ else\ v \leq Int.Pls)$
unfolding *nat-number-of-def number-of-is-id numeral-simps*
by *auto*

lemmas *numerals* = *nat-numeral-0-eq-0 nat-numeral-1-eq-1 numeral-2-eq-2*

32.5 Powers with Numeric Exponents

We cannot refer to the number $2::'a$ in *Ring-and-Field.thy*. We cannot prove general results about the numeral $-1::'a$, so we have to use $-(1::'a)$ instead.

lemma *power2-eq-square*: $(a::'a::recpower)^2 = a * a$
by (*simp add*: *numeral-2-eq-2 Power.power-Suc*)

lemma *zero-power2* [*simp*]: $(0::'a::\{semiring\text{-}1,recpower\})^2 = 0$
by (*simp add*: *power2-eq-square*)

lemma *one-power2* [*simp*]: $(1::'a::\{semiring\text{-}1,recpower\})^2 = 1$
by (*simp add*: *power2-eq-square*)

lemma *power3-eq-cube*: $(x::'a::recpower)\ \hat{}\ 3 = x * x * x$
apply (*subgoal-tac* $3 = Suc\ (Suc\ (Suc\ 0))$)
apply (*erule ssubst*)
apply (*simp add*: *power-Suc mult-ac*)
apply (*unfold nat-number-of-def*)
apply (*subst nat-eq-iff*)
apply *simp*
done

Squares of literal numerals will be evaluated.

lemmas *power2-eq-square-number-of* =
  power2-eq-square [*of number-of w*, *standard*]
declare *power2-eq-square-number-of* [*simp*]

lemma *zero-le-power2*[*simp*]: $0 \leq (a^2::'a::\{ordered\text{-}idom,recpower\})$
  by (*simp add*: *power2-eq-square*)

lemma *zero-less-power2*[*simp*]:
  $(0 < a^2) = (a \neq (0::'a::\{ordered\text{-}idom,recpower\}))$
  by (*force simp add*: *power2-eq-square zero-less-mult-iff linorder-neq-iff*)

lemma *power2-less-0*[*simp*]:
  fixes $a :: 'a::\{ordered\text{-}idom,recpower\}$
  shows $\sim (a^2 < 0)$
by (*force simp add*: *power2-eq-square mult-less-0-iff*)

lemma *zero-eq-power2*[*simp*]:
  $(a^2 = 0) = (a = (0::'a::\{ordered\text{-}idom,recpower\}))$
  by (*force simp add*: *power2-eq-square mult-eq-0-iff*)

lemma *abs-power2*[*simp*]:
  $abs(a^2) = (a^2::'a::\{ordered\text{-}idom,recpower\})$
  by (*simp add*: *power2-eq-square abs-mult abs-mult-self*)

lemma *power2-abs*[*simp*]:
  $(abs\ a)^2 = (a^2::'a::\{ordered\text{-}idom,recpower\})$
  by (*simp add*: *power2-eq-square abs-mult-self*)

lemma *power2-minus*[*simp*]:
  $(-\ a)^2 = (a^2::'a::\{comm\text{-}ring\text{-}1,recpower\})$
  by (*simp add*: *power2-eq-square*)

lemma *power2-le-imp-le*:
  fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
  shows $[\![x^2 \leq y^2;\ 0 \leq y]\!] \Longrightarrow x \leq y$
unfolding *numeral-2-eq-2* **by** (*rule power-le-imp-le-base*)

lemma *power2-less-imp-less*:
  fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
  shows $[\![x^2 < y^2;\ 0 \leq y]\!] \Longrightarrow x < y$
by (*rule power-less-imp-less-base*)

lemma *power2-eq-imp-eq*:
  fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
  shows $[\![x^2 = y^2;\ 0 \leq x;\ 0 \leq y]\!] \Longrightarrow x = y$
unfolding *numeral-2-eq-2* **by** (*erule* (2) *power-eq-imp-eq-base*, *simp*)

lemma *power-minus1-even*[*simp*]: $(-1) \hat{} (2*n) = (1::'a::\{comm\text{-}ring\text{-}1,recpower\})$
proof (*induct n*)
  case *0* **show** *?case* **by** *simp*
next
  case (*Suc n*) **then show** *?case* **by** (*simp add*: *power-Suc power-add*)
qed

lemma *power-minus1-odd*: $(-1) \hat{} Suc(2*n) = -(1::'a::\{comm\text{-}ring\text{-}1,recpower\})$
  by (*simp add*: *power-Suc*)

lemma *power-even-eq*: $(a::'a::recpower) \hat{} (2*n) = (a \hat{} n) \hat{} 2$
by (*subst mult-commute*) (*simp add*: *power-mult*)

lemma *power-odd-eq*: $(a::int) \hat{} Suc(2*n) = a * (a \hat{} n) \hat{} 2$
by (*simp add*: *power-even-eq*)

lemma *power-minus-even* [*simp*]:
    $(-a) \hat{} (2*n) = (a::'a::\{comm\text{-}ring\text{-}1,recpower\}) \hat{} (2*n)$
by (*simp add*: *power-minus1-even power-minus* [*of a*])

lemma *zero-le-even-power'*[*simp*]:
    $0 \leq (a::'a::\{ordered\text{-}idom,recpower\}) \hat{} (2*n)$
proof (*induct n*)
  case *0*
    show *?case* **by** (*simp add*: *zero-le-one*)
next
  case (*Suc n*)
    have $a \hat{} (2 * Suc\ n) = (a*a) * a \hat{} (2*n)$
      by (*simp add*: *mult-ac power-add power2-eq-square*)
    thus *?case*
      by (*simp add*: *prems zero-le-mult-iff*)
qed

lemma *odd-power-less-zero*:
    $(a::'a::\{ordered\text{-}idom,recpower\}) < 0 \Longrightarrow a \hat{} Suc(2*n) < 0$
proof (*induct n*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  have $a \hat{} Suc\ (2 * Suc\ n) = (a*a) * a \hat{} Suc(2*n)$
    by (*simp add*: *mult-ac power-add power2-eq-square*)
  thus *?case*
    by (*simp del*: *power-Suc add*: *prems mult-less-0-iff mult-neg-neg*)
qed

lemma *odd-0-le-power-imp-0-le*:
    $0 \leq a \hat{} Suc(2*n) \Longrightarrow 0 \leq (a::'a::\{ordered\text{-}idom,recpower\})$
apply (*insert odd-power-less-zero* [*of a n*])

apply (*force simp add*: *linorder-not-less* [*symmetric*])
done

Simprules for comparisons where common factors can be cancelled.

lemmas *zero-compare-simps* =
add-strict-increasing add-strict-increasing2 add-increasing
zero-le-mult-iff zero-le-divide-iff
zero-less-mult-iff zero-less-divide-iff
mult-le-0-iff divide-le-0-iff
mult-less-0-iff divide-less-0-iff
zero-le-power2 power2-less-0

32.5.1 Nat

lemma *Suc-pred'*: $0 < n \Longrightarrow n = Suc(n - 1)$
by (*simp add*: *numerals*)

lemmas *expand-Suc* = *Suc-pred'* [*of number-of v*, *standard*]

32.5.2 Arith

lemma *Suc-eq-add-numeral-1*: $Suc\ n = n + 1$
by (*simp add*: *numerals*)

lemma *Suc-eq-add-numeral-1-left*: $Suc\ n = 1 + n$
by (*simp add*: *numerals*)

lemma *add-eq-if*: $(m::nat) + n = (if\ m=0\ then\ n\ else\ Suc\ ((m - 1) + n))$
unfolding *One-nat-def* **by** (*cases m*) *simp-all*

lemma *mult-eq-if*: $(m::nat) * n = (if\ m=0\ then\ 0\ else\ n + ((m - 1) * n))$
unfolding *One-nat-def* **by** (*cases m*) *simp-all*

lemma *power-eq-if*: $(p\ \hat{}\ m :: nat) = (if\ m=0\ then\ 1\ else\ p * (p\ \hat{}\ (m - 1)))$
unfolding *One-nat-def* **by** (*cases m*) *simp-all*

32.6 Comparisons involving (0::nat)

Simplification already does $n < (0::'a)$, $n \leq (0::'a)$ and $(0::'a) \leq n$.

lemma *eq-number-of-0* [*simp*]:
$number\text{-}of\ v = (0::nat) \longleftrightarrow v \leq Int.Pls$
unfolding *nat-number-of-def number-of-is-id numeral-simps*
by *auto*

lemma *eq-0-number-of* [*simp*]:
$(0::nat) = number\text{-}of\ v \longleftrightarrow v \leq Int.Pls$

by (*rule trans* [*OF eq-sym-conv eq-number-of-0*])

lemma *less-0-number-of* [*simp*]:
(*0*::*nat*) < *number-of v* ⟷ *Int.Pls* < *v*
unfolding *nat-number-of-def number-of-is-id numeral-simps*
by *simp*

lemma *neg-imp-number-of-eq-0*: *neg* (*number-of v* :: *int*) ==> *number-of v* = (*0*::*nat*)
by (*simp del*: *nat-numeral-0-eq-0 add*: *nat-numeral-0-eq-0* [*symmetric*])

32.7 Comparisons involving *Suc*

lemma *eq-number-of-Suc* [*simp*]:
(*number-of v* = *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else nat pv = *n*)
apply (*simp only*: *simp-thms Let-def neg-eq-less-0 linorder-not-less*
number-of-pred nat-number-of-def
split add: *split-if*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply (*auto simp add*: *nat-eq-iff*)
done

lemma *Suc-eq-number-of* [*simp*]:
(*Suc n* = *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else nat pv = *n*)
by (*rule trans* [*OF eq-sym-conv eq-number-of-Suc*])

lemma *less-number-of-Suc* [*simp*]:
(*number-of v* < *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then True else nat pv < *n*)
apply (*simp only*: *simp-thms Let-def neg-eq-less-0 linorder-not-less*
number-of-pred nat-number-of-def
split add: *split-if*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply (*auto simp add*: *nat-less-iff*)
done

lemma *less-Suc-number-of* [*simp*]:
(*Suc n* < *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else n < *nat pv*)
apply (*simp only*: *simp-thms Let-def neg-eq-less-0 linorder-not-less*
number-of-pred nat-number-of-def
split add: *split-if*)
apply (*rule-tac x* = *number-of v* **in** *spec*)

apply (*auto simp add*: *zless-nat-eq-int-zless*)
done

lemma *le-number-of-Suc* [*simp*]:
(*number-of v <= Suc n*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then True else nat pv <= n)
by (*simp add*: *Let-def less-Suc-number-of linorder-not-less* [*symmetric*])

lemma *le-Suc-number-of* [*simp*]:
(*Suc n <= number-of v*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then False else n <= nat pv)
by (*simp add*: *Let-def less-number-of-Suc linorder-not-less* [*symmetric*])

lemma *eq-number-of-Pls-Min*: (*Numeral0* ::*int*) ~= *number-of Int.Min*
by *auto*

32.8 Max and Min Combined with *Suc*

lemma *max-number-of-Suc* [*simp*]:
max (*Suc n*) (*number-of v*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then Suc n else Suc(*max n* (*nat pv*)))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
split add: *split-if nat.split*)
apply (*rule-tac x = number-of v* **in** *spec*)
apply *auto*
done

lemma *max-Suc-number-of* [*simp*]:
max (*number-of v*) (*Suc n*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then Suc n else Suc(*max* (*nat pv*) *n*))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
split add: *split-if nat.split*)
apply (*rule-tac x = number-of v* **in** *spec*)
apply *auto*
done

lemma *min-number-of-Suc* [*simp*]:
min (*Suc n*) (*number-of v*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then 0 else Suc(*min n* (*nat pv*)))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
split add: *split-if nat.split*)
apply (*rule-tac x = number-of v* **in** *spec*)
apply *auto*

done

lemma *min-Suc-number-of* [*simp*]:
  min (*number-of v*) (*Suc n*) =
    (*let pv* = *number-of* (*Int.pred v*) *in*
     if neg pv then 0 else Suc(*min* (*nat pv*) *n*))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
      split add: *split-if nat.split*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply *auto*
done

32.9 Literal arithmetic involving powers

lemma *nat-power-eq*: (*0*::*int*) <= *z* ==> *nat* (*z*^*n*) = *nat z* ^ *n*
apply (*induct n*)
apply (*simp-all* (*no-asm-simp*) *add*: *nat-mult-distrib*)
done

lemma *power-nat-number-of*:
  (*number-of v* :: *nat*) ^ *n* =
    (*if neg* (*number-of v* :: *int*) *then 0*^*n else nat* ((*number-of v* :: *int*) ^ *n*))
by (*simp only*: *simp-thms neg-nat not-neg-eq-ge-0 nat-number-of-def nat-power-eq*
     split add: *split-if cong*: *imp-cong*)

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

For arbitrary rings

lemma *power-number-of-even*:
  fixes *z* :: *'a*::{*number-ring*,*recpower*}
  shows *z* ^ *number-of* (*Int.Bit0 w*) = (*let w* = *z* ^ (*number-of w*) *in w* * *w*)
unfolding *Let-def nat-number-of-def number-of-Bit0*
apply (*rule-tac x* = *number-of w* **in** *spec*, *clarify*)
apply (*case-tac* (*0*::*int*) <= *x*)
apply (*auto simp add*: *nat-mult-distrib power-even-eq power2-eq-square*)
done

lemma *power-number-of-odd*:
  fixes *z* :: *'a*::{*number-ring*,*recpower*}
  shows *z* ^ *number-of* (*Int.Bit1 w*) = (*if* (*0*::*int*) <= *number-of w*
    then (*let w* = *z* ^ (*number-of w*) *in z* * *w* * *w*) *else 1*)
unfolding *Let-def nat-number-of-def number-of-Bit1*
apply (*rule-tac x* = *number-of w* **in** *spec*, *auto*)
apply (*simp only*: *nat-add-distrib nat-mult-distrib*)
apply *simp*
apply (*auto simp add*: *nat-add-distrib nat-mult-distrib power-even-eq power2-eq-square*

neg-nat power-Suc)
done

lemmas *zpower-number-of-even* = *power-number-of-even* [**where** *'a=int*]
lemmas *zpower-number-of-odd* = *power-number-of-odd* [**where** *'a=int*]

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

lemmas *power-number-of-odd-number-of* [*simp*] =
power-number-of-odd [*of number-of v*, *standard*]

ML
⟪
val numeral-ss = @{*simpset*} *addsimps* @{*thms numerals*};

val nat-bin-arith-setup =
Lin-Arith.map-data
(*fn* {*add-mono-thms*, *mult-mono-thms*, *inj-thms*, *lessD*, *neqE*, *simpset*} =>
{*add-mono-thms* = *add-mono-thms*, *mult-mono-thms* = *mult-mono-thms*,
inj-thms = *inj-thms*,
lessD = *lessD*, *neqE* = *neqE*,
simpset = *simpset addsimps* @{*thms neg-simps*} @
[@{*thm Suc-nat-number-of*}, @{*thm int-nat-number-of*}]})
⟫

declaration ⟪ *K nat-bin-arith-setup* ⟫

declare *split-div*[*of - - number-of k*, *standard*, *arith-split*]
declare *split-mod*[*of - - number-of k*, *standard*, *arith-split*]

lemma *nat-number-of-Pls*: *Numeral0* = (*0*::*nat*)
by (*simp add*: *number-of-Pls nat-number-of-def*)

lemma *nat-number-of-Min*: *number-of Int.Min* = (*0*::*nat*)
apply (*simp only*: *number-of-Min nat-number-of-def nat-zminus-int*)
done

lemma *nat-number-of-Bit0*:
number-of (*Int.Bit0 w*) = (*let n*::*nat* = *number-of w in n* + *n*)
unfolding *nat-number-of-def number-of-is-id numeral-simps Let-def*
by *auto*

lemma *nat-number-of-Bit1*:
number-of (*Int.Bit1 w*) =
(*if neg* (*number-of w* :: *int*) *then 0*

else let n = number-of w in Suc (n + n))
unfolding *nat-number-of-def number-of-is-id numeral-simps neg-def Let-def*
by *auto*

lemmas *nat-number =*
nat-number-of-Pls nat-number-of-Min
nat-number-of-Bit0 nat-number-of-Bit1

lemma *Let-Suc* [*simp*]: *Let (Suc n) f == f (Suc n)*
by (*simp add*: *Let-def*)

lemma *power-m1-even*: *(−1) ^ (2∗n) = (1::'a::{number-ring,recpower})*
by (*simp add*: *power-mult power-Suc*)

lemma *power-m1-odd*: *(−1) ^ Suc(2∗n) = (−1::'a::{number-ring,recpower})*
by (*simp add*: *power-mult power-Suc*)

32.10 Literal arithmetic and *of-nat*

lemma *of-nat-double*:
0 ≤ x ==> of-nat (nat (2 ∗ x)) = of-nat (nat x) + of-nat (nat x)
by (*simp only*: *mult-2 nat-add-distrib of-nat-add*)

lemma *nat-numeral-m1-eq-0*: *−1 = (0::nat)*
by (*simp only*: *nat-number-of-def*)

lemma *of-nat-number-of-lemma*:
of-nat (number-of v :: nat) =
(if 0 ≤ (number-of v :: int)
then (number-of v :: 'a :: number-ring)
else 0)
by (*simp add*: *int-number-of-def nat-number-of-def number-of-eq of-nat-nat*)

lemma *of-nat-number-of-eq* [*simp*]:
of-nat (number-of v :: nat) =
(if neg (number-of v :: int) then 0
else (number-of v :: 'a :: number-ring))
by (*simp only*: *of-nat-number-of-lemma neg-def*, *simp*)

32.11 Lemmas for the Combination and Cancellation Simprocs

lemma *nat-number-of-add-left*:
number-of v + (number-of v' + (k::nat)) =
(if neg (number-of v :: int) then number-of v' + k
else if neg (number-of v' :: int) then number-of v + k
else number-of (v + v') + k)
unfolding *nat-number-of-def number-of-is-id neg-def*
by *auto*

lemma *nat-number-of-mult-left*:
*number-of v * (number-of v' * (k::nat)) =*
(if v < Int.Pls then 0
*else number-of (v * v') * k)*
by *simp*

32.11.1 For *combine-numerals*

lemma *left-add-mult-distrib*: *i*u + (j*u + k) = (i+j)*u + (k::nat)*
by (*simp add*: *add-mult-distrib*)

32.11.2 For *cancel-numerals*

lemma *nat-diff-add-eq1*:
*j <= (i::nat) ==> ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)*
by (*simp split add*: *nat-diff-split add*: *add-mult-distrib*)

lemma *nat-diff-add-eq2*:
*i <= (j::nat) ==> ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))*
by (*simp split add*: *nat-diff-split add*: *add-mult-distrib*)

lemma *nat-eq-add-iff1*:
*j <= (i::nat) ==> (i*u + m = j*u + n) = ((i-j)*u + m = n)*
by (*auto split add*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-eq-add-iff2*:
*i <= (j::nat) ==> (i*u + m = j*u + n) = (m = (j-i)*u + n)*
by (*auto split add*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-less-add-iff1*:
*j <= (i::nat) ==> (i*u + m < j*u + n) = ((i-j)*u + m < n)*
by (*auto split add*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-less-add-iff2*:
*i <= (j::nat) ==> (i*u + m < j*u + n) = (m < (j-i)*u + n)*
by (*auto split add*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-le-add-iff1*:
*j <= (i::nat) ==> (i*u + m <= j*u + n) = ((i-j)*u + m <= n)*
by (*auto split add*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-le-add-iff2*:
*i <= (j::nat) ==> (i*u + m <= j*u + n) = (m <= (j-i)*u + n)*
by (*auto split add*: *nat-diff-split simp add*: *add-mult-distrib*)

32.11.3 For *cancel-numeral-factors*

lemma *nat-mult-le-cancel1*: *(0::nat) < k ==> (k*m <= k*n) = (m<=n)*
by *auto*

lemma *nat-mult-less-cancel1*: $(0::nat) < k ==> (k*m < k*n) = (m<n)$
by *auto*

lemma *nat-mult-eq-cancel1*: $(0::nat) < k ==> (k*m = k*n) = (m=n)$
by *auto*

lemma *nat-mult-div-cancel1*: $(0::nat) < k ==> (k*m)\ div\ (k*n) = (m\ div\ n)$
by *auto*

lemma *nat-mult-dvd-cancel-disj*[*simp*]:
$(k*m)\ dvd\ (k*n) = (k=0 \mid m\ dvd\ (n::nat))$
by(*auto simp*: *dvd-eq-mod-eq-0 mod-mult-distrib2*[*symmetric*])

lemma *nat-mult-dvd-cancel1*: $0 < k \Longrightarrow (k*m)\ dvd\ (k*n::nat) = (m\ dvd\ n)$
by(*auto*)

32.11.4 For *cancel-factor*

lemma *nat-mult-le-cancel-disj*: $(k*m <= k*n) = ((0::nat) < k --> m<=n)$
by *auto*

lemma *nat-mult-less-cancel-disj*: $(k*m < k*n) = ((0::nat) < k \ \&\ m<n)$
by *auto*

lemma *nat-mult-eq-cancel-disj*: $(k*m = k*n) = (k = (0::nat) \mid m=n)$
by *auto*

lemma *nat-mult-div-cancel-disj*[*simp*]:
$(k*m)\ div\ (k*n) = (if\ k = (0::nat)\ then\ 0\ else\ m\ div\ n)$
by (*simp add*: *nat-mult-div-cancel1*)

32.12 Simprocs for the Naturals

use *Tools/nat-simprocs.ML*
declaration ⟪ *K nat-simprocs-setup* ⟫

32.12.1 For simplifying *Suc m − K* and *K − Suc m*

Where K above is a literal

lemma *Suc-diff-eq-diff-pred*: $Numeral0 < n ==> Suc\ m - n = m - (n - Numeral1)$
by (*simp add*: *numeral-0-eq-0 numeral-1-eq-1 split add*: *nat-diff-split*)

Now just instantiating *n* to *number-of v* does the right simplification, but with some redundant inequality tests.

lemma *neg-number-of-pred-iff-0*:
$neg\ (number\text{-}of\ (Int.pred\ v)::int) = (number\text{-}of\ v = (0::nat))$
apply (*subgoal-tac neg* $(number\text{-}of\ (Int.pred\ v)) = (number\text{-}of\ v < Suc\ 0)$)

apply (*simp only*: *less-Suc-eq-le le-0-eq*)
apply (*subst less-number-of-Suc*, *simp*)
done

No longer required as a simprule because of the *inverse-fold* simproc

lemma *Suc-diff-number-of*:
    Int.Pls < *v* ==>
     Suc m − (*number-of v*) = *m* − (*number-of* (*Int.pred v*))
apply (*subst Suc-diff-eq-diff-pred*)
apply *simp*
apply (*simp del*: *nat-numeral-1-eq-1*)
apply (*auto simp only*: *diff-nat-number-of less-0-number-of* [*symmetric*]
                  neg-number-of-pred-iff-0)
done

lemma *diff-Suc-eq-diff-pred*: *m* − *Suc n* = (*m* − *1*) − *n*
by (*simp add*: *numerals split add*: *nat-diff-split*)

32.12.2 For *nat-case* and *nat-rec*

lemma *nat-case-number-of* [*simp*]:
    nat-case a f (*number-of v*) =
      (*let pv* = *number-of* (*Int.pred v*) *in*
       if neg pv then a else f (*nat pv*))
by (*simp split add*: *nat.split add*: *Let-def neg-number-of-pred-iff-0*)

lemma *nat-case-add-eq-if* [*simp*]:
    nat-case a f ((*number-of v*) + *n*) =
      (*let pv* = *number-of* (*Int.pred v*) *in*
       if neg pv then nat-case a f n else f (*nat pv* + *n*))
apply (*subst add-eq-if*)
apply (*simp split add*: *nat.split*
          del: *nat-numeral-1-eq-1*
          add: *nat-numeral-1-eq-1* [*symmetric*]
              numeral-1-eq-Suc-0 [*symmetric*]
              neg-number-of-pred-iff-0)
done

lemma *nat-rec-number-of* [*simp*]:
    nat-rec a f (*number-of v*) =
      (*let pv* = *number-of* (*Int.pred v*) *in*
       if neg pv then a else f (*nat pv*) (*nat-rec a f* (*nat pv*)))
apply (*case-tac* (*number-of v*) ::*nat*)
apply (*simp-all* (*no-asm-simp*) *add*: *Let-def neg-number-of-pred-iff-0*)
apply (*simp split add*: *split-if-asm*)
done

lemma *nat-rec-add-eq-if* [*simp*]:
    nat-rec a f (*number-of v* + *n*) =

```
      (let pv = number-of (Int.pred v) in
       if neg pv then nat-rec a f n
                 else f (nat pv + n) (nat-rec a f (nat pv + n)))
apply (subst add-eq-if)
apply (simp split add: nat.split
          del: nat-numeral-1-eq-1
          add: nat-numeral-1-eq-1 [symmetric]
               numeral-1-eq-Suc-0 [symmetric]
               neg-number-of-pred-iff-0)
done
```

32.12.3 Various Other Lemmas

Evens and Odds, for Mutilated Chess Board

Lemmas for specialist use, NOT as default simprules

```
lemma nat-mult-2: 2 * z = (z+z::nat)
proof -
  have 2*z = (1 + 1)*z by simp
  also have ... = z+z by (simp add: left-distrib)
  finally show ?thesis .
qed

lemma nat-mult-2-right: z * 2 = (z+z::nat)
by (subst mult-commute, rule nat-mult-2)
```

Case analysis on $n < (2::'a)$

```
lemma less-2-cases: (n::nat) < 2 ==> n = 0 | n = Suc 0
by arith

lemma div2-Suc-Suc [simp]: Suc(Suc m) div 2 = Suc (m div 2)
by arith

lemma add-self-div-2 [simp]: (m + m) div 2 = (m::nat)
by (simp add: nat-mult-2 [symmetric])

lemma mod2-Suc-Suc [simp]: Suc(Suc(m)) mod 2 = m mod 2
apply (subgoal-tac m mod 2 < 2)
apply (erule less-2-cases [THEN disjE])
apply (simp-all (no-asm-simp) add: Let-def mod-Suc nat-1)
done

lemma mod2-gr-0 [simp]: !!m::nat. (0 < m mod 2) = (m mod 2 = 1)
apply (subgoal-tac m mod 2 < 2)
apply (force simp del: mod-less-divisor, simp)
done
```

Removal of Small Numerals: 0, 1 and (in additive positions) 2

```
lemma add-2-eq-Suc [simp]: 2 + n = Suc (Suc n)
```

by *simp*

lemma *add-2-eq-Suc'* [*simp*]: *n + 2 = Suc (Suc n)*
by *simp*

Can be used to eliminate long strings of Sucs, but not by default

lemma *Suc3-eq-add-3*: *Suc (Suc (Suc n)) = 3 + n*
by *simp*

These lemmas collapse some needless occurrences of Suc: at least three Sucs, since two and fewer are rewritten back to Suc again! We already have some rules to simplify operands smaller than 3.

lemma *div-Suc-eq-div-add3* [*simp*]: *m div (Suc (Suc (Suc n))) = m div (3+n)*
by (*simp add*: *Suc3-eq-add-3*)

lemma *mod-Suc-eq-mod-add3* [*simp*]: *m mod (Suc (Suc (Suc n))) = m mod (3+n)*
by (*simp add*: *Suc3-eq-add-3*)

lemma *Suc-div-eq-add3-div*: *(Suc (Suc (Suc m))) div n = (3+m) div n*
by (*simp add*: *Suc3-eq-add-3*)

lemma *Suc-mod-eq-add3-mod*: *(Suc (Suc (Suc m))) mod n = (3+m) mod n*
by (*simp add*: *Suc3-eq-add-3*)

lemmas *Suc-div-eq-add3-div-number-of =*
Suc-div-eq-add3-div [*of - number-of v*, *standard*]
declare *Suc-div-eq-add3-div-number-of* [*simp*]

lemmas *Suc-mod-eq-add3-mod-number-of =*
Suc-mod-eq-add3-mod [*of - number-of v*, *standard*]
declare *Suc-mod-eq-add3-mod-number-of* [*simp*]

end

33 Groebner-Basis: Semiring normalization and Groebner Bases

theory *Groebner-Basis*
imports *NatBin*
uses
Tools/Groebner-Basis/misc.ML
Tools/Groebner-Basis/normalizer-data.ML
(*Tools/Groebner-Basis/normalizer.ML*)
(*Tools/Groebner-Basis/groebner.ML*)
begin

33.1 Semiring normalization

setup *NormalizerData.setup*

locale *gb-semiring* =
  fixes *add mul pwr r0 r1*
  assumes *add-a*:(*add x* (*add y z*) = *add* (*add x y*) *z*)
    and *add-c*: *add x y* = *add y x* **and** *add-0*:*add r0 x* = *x*
    and *mul-a*:*mul x* (*mul y z*) = *mul* (*mul x y*) *z* **and** *mul-c*:*mul x y* = *mul y x*
    and *mul-1*:*mul r1 x* = *x* **and** *mul-0*:*mul r0 x* = *r0*
    and *mul-d*:*mul x* (*add y z*) = *add* (*mul x y*) (*mul x z*)
    and *pwr-0*:*pwr x 0* = *r1* **and** *pwr-Suc*:*pwr x* (*Suc n*) = *mul x* (*pwr x n*)
begin

lemma *mul-pwr*:*mul* (*pwr x p*) (*pwr x q*) = *pwr x* (*p* + *q*)
proof (*induct p*)
  case *0*
  then show *?case* **by** (*auto simp add*: *pwr-0 mul-1*)
next
  case *Suc*
  from *this* [*symmetric*] **show** *?case*
    by (*auto simp add*: *pwr-Suc mul-1 mul-a*)
qed

lemma *pwr-mul*: *pwr* (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*)
proof (*induct q arbitrary*: *x y*, *auto simp add*:*pwr-0 pwr-Suc mul-1*)
  fix *q x y*
  assume ⋀*x y*. *pwr* (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*)
  have *mul* (*mul x y*) (*mul* (*pwr x q*) (*pwr y q*)) = *mul x* (*mul y* (*mul* (*pwr x q*) (*pwr y q*)))
    by (*simp add*: *mul-a*)
  also have ... = (*mul* (*mul y* (*mul* (*pwr y q*) (*pwr x q*))) *x*) **by** (*simp add*: *mul-c*)
  also have ... = (*mul* (*mul y* (*pwr y q*)) (*mul* (*pwr x q*) *x*)) **by** (*simp add*: *mul-a*)
  finally show *mul* (*mul x y*) (*mul* (*pwr x q*) (*pwr y q*)) =
    mul (*mul x* (*pwr x q*)) (*mul y* (*pwr y q*)) **by** (*simp add*: *mul-c*)
qed

lemma *pwr-pwr*: *pwr* (*pwr x p*) *q* = *pwr x* (*p* ∗ *q*)
proof (*induct p arbitrary*: *q*)
  case *0*
  show *?case* **using** *pwr-Suc mul-1 pwr-0* **by** (*induct q*) *auto*
next
  case *Suc*
  thus *?case* **by** (*auto simp add*: *mul-pwr* [*symmetric*] *pwr-mul pwr-Suc*)
qed

33.1.1 Declaring the abstract theory

lemma *semiring-ops*:

shows *TERM* (*add x y*) **and** *TERM* (*mul x y*) **and** *TERM* (*pwr x n*)
and *TERM r0* **and** *TERM r1* **.**

lemma *semiring-rules*:
add (*mul a m*) (*mul b m*) = *mul* (*add a b*) *m*
add (*mul a m*) *m* = *mul* (*add a r1*) *m*
add m (*mul a m*) = *mul* (*add a r1*) *m*
add m m = *mul* (*add r1 r1*) *m*
add r0 a = *a*
add a r0 = *a*
mul a b = *mul b a*
mul (*add a b*) *c* = *add* (*mul a c*) (*mul b c*)
mul r0 a = *r0*
mul a r0 = *r0*
mul r1 a = *a*
mul a r1 = *a*
mul (*mul lx ly*) (*mul rx ry*) = *mul* (*mul lx rx*) (*mul ly ry*)
mul (*mul lx ly*) (*mul rx ry*) = *mul lx* (*mul ly* (*mul rx ry*))
mul (*mul lx ly*) (*mul rx ry*) = *mul rx* (*mul* (*mul lx ly*) *ry*)
mul (*mul lx ly*) *rx* = *mul* (*mul lx rx*) *ly*
mul (*mul lx ly*) *rx* = *mul lx* (*mul ly rx*)
mul lx (*mul rx ry*) = *mul* (*mul lx rx*) *ry*
mul lx (*mul rx ry*) = *mul rx* (*mul lx ry*)
add (*add a b*) (*add c d*) = *add* (*add a c*) (*add b d*)
add (*add a b*) *c* = *add a* (*add b c*)
add a (*add c d*) = *add c* (*add a d*)
add (*add a b*) *c* = *add* (*add a c*) *b*
add a c = *add c a*
add a (*add c d*) = *add* (*add a c*) *d*
mul (*pwr x p*) (*pwr x q*) = *pwr x* (*p* + *q*)
mul x (*pwr x q*) = *pwr x* (*Suc q*)
mul (*pwr x q*) *x* = *pwr x* (*Suc q*)
mul x x = *pwr x 2*
pwr (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*)
pwr (*pwr x p*) *q* = *pwr x* (*p* ∗ *q*)
pwr x 0 = *r1*
pwr x 1 = *x*
mul x (*add y z*) = *add* (*mul x y*) (*mul x z*)
pwr x (*Suc q*) = *mul x* (*pwr x q*)
pwr x (*2*∗*n*) = *mul* (*pwr x n*) (*pwr x n*)
pwr x (*Suc* (*2*∗*n*)) = *mul x* (*mul* (*pwr x n*) (*pwr x n*))
proof −
show *add* (*mul a m*) (*mul b m*) = *mul* (*add a b*) *m* **using** *mul-d mul-c* **by** *simp*
next show*add* (*mul a m*) *m* = *mul* (*add a r1*) *m* **using** *mul-d mul-c mul-1* **by** *simp*
next show *add m* (*mul a m*) = *mul* (*add a r1*) *m* **using** *mul-c mul-d mul-1 add-c* **by** *simp*
next show *add m m* = *mul* (*add r1 r1*) *m* **using** *mul-c mul-d mul-1* **by** *simp*
next show *add r0 a* = *a* **using** *add-0* **by** *simp*

next show *add a r0 = a* **using** *add-0 add-c* **by** *simp*
next show *mul a b = mul b a* **using** *mul-c* **by** *simp*
next show *mul (add a b) c = add (mul a c) (mul b c)* **using** *mul-c mul-d* **by** *simp*
next show *mul r0 a = r0* **using** *mul-0* **by** *simp*
next show *mul a r0 = r0* **using** *mul-0 mul-c* **by** *simp*
next show *mul r1 a = a* **using** *mul-1* **by** *simp*
next show *mul a r1 = a* **using** *mul-1 mul-c* **by** *simp*
next show *mul (mul lx ly) (mul rx ry) = mul (mul lx rx) (mul ly ry)*
  using *mul-c mul-a* **by** *simp*
next show *mul (mul lx ly) (mul rx ry) = mul lx (mul ly (mul rx ry))*
  using *mul-a* **by** *simp*
next
  have *mul (mul lx ly) (mul rx ry) = mul (mul rx ry) (mul lx ly)* **by** (*rule mul-c*)
  also have *... = mul rx (mul ry (mul lx ly))* **using** *mul-a* **by** *simp*
  finally
  show *mul (mul lx ly) (mul rx ry) = mul rx (mul (mul lx ly) ry)*
    using *mul-c* **by** *simp*
next show *mul (mul lx ly) rx = mul (mul lx rx) ly* **using** *mul-c mul-a* **by** *simp*
next
  show *mul (mul lx ly) rx = mul lx (mul ly rx)* **by** (*simp add*: *mul-a*)
next show *mul lx (mul rx ry) = mul (mul lx rx) ry* **by** (*simp add*: *mul-a*)
next show *mul lx (mul rx ry) = mul rx (mul lx ry)* **by** (*simp add*: *mul-a*,*simp add*: *mul-c*)
next show *add (add a b) (add c d) = add (add a c) (add b d)*
  using *add-c add-a* **by** *simp*
next show *add (add a b) c = add a (add b c)* **using** *add-a* **by** *simp*
next show *add a (add c d) = add c (add a d)*
  apply (*simp add*: *add-a*) **by** (*simp only*: *add-c*)
next show *add (add a b) c = add (add a c) b* **using** *add-a add-c* **by** *simp*
next show *add a c = add c a* **by** (*rule add-c*)
next show *add a (add c d) = add (add a c) d* **using** *add-a* **by** *simp*
next show *mul (pwr x p) (pwr x q) = pwr x (p + q)* **by** (*rule mul-pwr*)
next show *mul x (pwr x q) = pwr x (Suc q)* **using** *pwr-Suc* **by** *simp*
next show *mul (pwr x q) x = pwr x (Suc q)* **using** *pwr-Suc mul-c* **by** *simp*
next show *mul x x = pwr x 2* **by** (*simp add*: *nat-number pwr-Suc pwr-0 mul-1 mul-c*)
next show *pwr (mul x y) q = mul (pwr x q) (pwr y q)* **by** (*rule pwr-mul*)
next show *pwr (pwr x p) q = pwr x (p * q)* **by** (*rule pwr-pwr*)
next show *pwr x 0 = r1* **using** *pwr-0* **.**
next show *pwr x 1 = x* **unfolding** *One-nat-def* **by** (*simp add*: *nat-number pwr-Suc pwr-0 mul-1 mul-c*)
next show *mul x (add y z) = add (mul x y) (mul x z)* **using** *mul-d* **by** *simp*
next show *pwr x (Suc q) = mul x (pwr x q)* **using** *pwr-Suc* **by** *simp*
next show *pwr x (2 * n) = mul (pwr x n) (pwr x n)* **by** (*simp add*: *nat-number mul-pwr*)
next show *pwr x (Suc (2 * n)) = mul x (mul (pwr x n) (pwr x n))*
  by (*simp add*: *nat-number pwr-Suc mul-pwr*)
qed

lemmas *gb-semiring-axioms'* =
gb-semiring-axioms [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*]

end

interpretation *class-semiring*: *gb-semiring*
op + *op* ∗ *op* ^ *0*::'*a*::{*comm-semiring-1*, *recpower*} *1*
proof qed (*auto simp add*: *algebra-simps power-Suc*)

lemmas *nat-arith* =
add-nat-number-of
diff-nat-number-of
mult-nat-number-of
eq-nat-number-of
less-nat-number-of

lemma *not-iszero-Numeral1*: ¬ *iszero* (*Numeral1*::'*a*::*number-ring*)
by (*simp add*: *numeral-1-eq-1*)

lemmas *comp-arith* =
Let-def arith-simps nat-arith rel-simps neg-simps if-False
if-True add-0 add-Suc add-number-of-left mult-number-of-left
numeral-1-eq-1[*symmetric*] *Suc-eq-add-numeral-1*
numeral-0-eq-0[*symmetric*] *numerals*[*symmetric*]
iszero-simps not-iszero-Numeral1

lemmas *semiring-norm* = *comp-arith*

ML ⟪
local

open Conv;

fun numeral-is-const ct = *can HOLogic.dest-number* (*Thm.term-of ct*);

fun int-of-rat x =
(*case Rat.quotient-of-rat x of* (*i*, *1*) => *i*
| - => *error int-of-rat*: *bad int*);

val numeral-conv =
Simplifier.rewrite (*HOL-basic-ss addsimps* @{*thms semiring-norm*}) *then-conv*
Simplifier.rewrite (*HOL-basic-ss addsimps*
(@{*thms numeral-1-eq-1*} @ @{*thms numeral-0-eq-0*} @ @{*thms numerals*(*1*−*2*)}));

in

```
fun normalizer-funs key =
  NormalizerData.funs key
   {is-const = fn phi => numeral-is-const,
    dest-const = fn phi => fn ct =>
      Rat.rat-of-int (snd
        (HOLogic.dest-number (Thm.term-of ct)
          handle TERM - => error ring-dest-const)),
    mk-const = fn phi => fn cT => fn x => Numeral.mk-cnumber cT (int-of-rat
x),
    conv = fn phi => K numeral-conv}

end
⟫
```

declaration ⟪ *normalizer-funs* @{*thm class-semiring.gb-semiring-axioms'*} ⟫

locale *gb-ring* = *gb-semiring* +
  fixes *sub* :: *'a* ⇒ *'a* ⇒ *'a*
    and *neg* :: *'a* ⇒ *'a*
  assumes *neg-mul*: *neg x* = *mul* (*neg r1*) *x*
    and *sub-add*: *sub x y* = *add x* (*neg y*)
begin

lemma *ring-ops*: **shows** *TERM* (*sub x y*) **and** *TERM* (*neg x*) **.**

lemmas *ring-rules* = *neg-mul sub-add*

lemmas *gb-ring-axioms'* =
  gb-ring-axioms [*normalizer*
    semiring ops: *semiring-ops*
    semiring rules: *semiring-rules*
    ring ops: *ring-ops*
    ring rules: *ring-rules*]

end

interpretation *class-ring*: *gb-ring op* + *op* ∗ *op* ^
    0::*'a*::{*comm-semiring-1*,*recpower*,*number-ring*} *1 op* − *uminus*
  proof qed *simp-all*

declaration ⟪ *normalizer-funs* @{*thm class-ring.gb-ring-axioms'*} ⟫

use *Tools/Groebner-Basis/normalizer.ML*

method-setup *sring-norm* = ⟪
  Scan.succeed (*SIMPLE-METHOD'* o *Normalizer.semiring-normalize-tac*)
⟫ *semiring normalizer*

locale *gb-field* = *gb-ring* +
  fixes *divide* :: *'a* ⇒ *'a* ⇒ *'a*
    and *inverse*:: *'a* ⇒ *'a*
  assumes *divide-inverse*: *divide x y* = *mul x* (*inverse y*)
     and *inverse-divide*: *inverse x* = *divide r1 x*
begin

lemma *field-ops*: **shows** *TERM* (*divide x y*) **and** *TERM* (*inverse x*) **.**

lemmas *field-rules* = *divide-inverse inverse-divide*

lemmas *gb-field-axioms'* =
  gb-field-axioms [*normalizer*
    semiring ops: *semiring-ops*
    semiring rules: *semiring-rules*
    ring ops: *ring-ops*
    ring rules: *ring-rules*
    field ops: *field-ops*
    field rules: *field-rules*]

end

33.2 Groebner Bases

locale *semiringb* = *gb-semiring* +
  assumes *add-cancel*: *add* (*x*::*'a*) *y* = *add x z* ⟷ *y* = *z*
  and *add-mul-solve*: *add* (*mul w y*) (*mul x z*) =
    add (*mul w z*) (*mul x y*) ⟷ *w* = *x* ∨ *y* = *z*
begin

lemma *noteq-reduce*: *a* ≠ *b* ∧ *c* ≠ *d* ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
proof−
  have *a* ≠ *b* ∧ *c* ≠ *d* ⟷ ¬ (*a* = *b* ∨ *c* = *d*) **by** *simp*
  also have ... ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
    using *add-mul-solve* **by** *blast*
  finally show *a* ≠ *b* ∧ *c* ≠ *d* ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
    by *simp*
qed

lemma *add-scale-eq-noteq*: ⟦*r* ≠ *r0* ; (*a* = *b*) ∧ ~(*c* = *d*)⟧
  ⟹ *add a* (*mul r c*) ≠ *add b* (*mul r d*)
proof(*clarify*)

```
  assume nz: r≠ r0 and cnd: c≠d
    and eq: add b (mul r c) = add b (mul r d)
  hence mul r c = mul r d using cnd add-cancel by simp
  hence add (mul r0 d) (mul r c) = add (mul r0 c) (mul r d)
    using mul-0 add-cancel by simp
  thus False using add-mul-solve nz cnd by simp
qed

lemma add-r0-iff:  x = add x a ⟷ a = r0
proof−
  have a = r0 ⟷ add x a = add x r0 by (simp add: add-cancel)
  thus x = add x a ⟷ a = r0 by (auto simp add: add-c add-0)
qed

declare gb-semiring-axioms' [normalizer del]

lemmas semiringb-axioms' = semiringb-axioms [normalizer
  semiring ops: semiring-ops
  semiring rules: semiring-rules
  idom rules: noteq-reduce add-scale-eq-noteq]

end

locale ringb = semiringb + gb-ring +
  assumes subr0-iff: sub x y = r0 ⟷ x = y
begin

declare gb-ring-axioms' [normalizer del]

lemmas ringb-axioms' = ringb-axioms [normalizer
  semiring ops: semiring-ops
  semiring rules: semiring-rules
  ring ops: ring-ops
  ring rules: ring-rules
  idom rules: noteq-reduce add-scale-eq-noteq
  ideal rules: subr0-iff add-r0-iff]

end


lemma no-zero-divirors-neq0:
  assumes az: (a::'a::no-zero-divisors) ≠ 0
    and ab: a*b = 0 shows b = 0
proof −
  { assume bz: b ≠ 0
    from no-zero-divisors [OF az bz] ab have False by blast }
  thus b = 0 by blast
qed
```

interpretation *class-ringb*: *ringb*
*op + op * op ^ 0::'a::{idom,recpower,number-ring} 1 op − uminus*
proof(*unfold-locales*, *simp add*: *algebra-simps power-Suc*, *auto*)
fix *w x y z ::'a::{idom,recpower,number-ring}*
assume *p*: $w * y + x * z = w * z + x * y$ **and** *ynz*: $y \neq z$
hence *ynz'*: $y - z \neq 0$ **by** *simp*
from *p* **have** $w * y + x* z - w*z - x*y = 0$ **by** *simp*
hence $w* (y - z) - x * (y - z) = 0$ **by** (*simp add*: *algebra-simps*)
hence $(y - z) * (w - x) = 0$ **by** (*simp add*: *algebra-simps*)
with *no-zero-divirors-neq0* [*OF ynz'*]
have $w - x = 0$ **by** *blast*
thus $w = x$ **by** *simp*
qed

declaration ⟪ *normalizer-funs* @{*thm class-ringb.ringb-axioms'*} ⟫

interpretation *natgb*: *semiringb*
*op + op * op ^ 0::nat 1*
proof (*unfold-locales*, *simp add*: *algebra-simps power-Suc*)
fix *w x y z ::nat*
{ assume *p*: $w * y + x * z = w * z + x * y$ **and** *ynz*: $y \neq z$
hence $y < z \vee y > z$ **by** *arith*
moreover {
assume *lt*:$y < z$ **hence** $\exists k.\ z = y + k \wedge k > 0$ **by** (*rule-tac x*=$z - y$ **in** *exI*, *auto*)
then obtain *k* **where** *kp*: $k>0$ **and** *yz*:$z = y + k$ **by** *blast*
from *p* **have** $(w * y + x *y) + x*k = (w * y + x*y) + w*k$ **by** (*simp add*: *yz algebra-simps*)
hence $x*k = w*k$ **by** *simp*
hence $w = x$ **using** *kp* **by** (*simp add*: *mult-cancel2*) **}**
moreover {
assume *lt*: $y > z$ **hence** $\exists k.\ y = z + k \wedge k>0$ **by** (*rule-tac x*=$y - z$ **in** *exI*, *auto*)
then obtain *k* **where** *kp*: $k>0$ **and** *yz*:$y = z + k$ **by** *blast*
from *p* **have** $(w * z + x *z) + w*k = (w * z + x*z) + x*k$ **by** (*simp add*: *yz algebra-simps*)
hence $w*k = x*k$ **by** *simp*
hence $w = x$ **using** *kp* **by** (*simp add*: *mult-cancel2*)**}**
ultimately have $w=x$ **by** *blast* **}**
thus $(w * y + x * z = w * z + x * y) = (w = x \vee y = z)$ **by** *auto*
qed

declaration ⟪ *normalizer-funs* @{*thm natgb.semiringb-axioms'*} ⟫

locale *fieldgb* = *ringb* + *gb-field*
begin

declare *gb-field-axioms'* [*normalizer del*]

lemmas *fieldgb-axioms'* = *fieldgb-axioms* [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*
field ops: *field-ops*
field rules: *field-rules*
idom rules: *noteq-reduce add-scale-eq-noteq*
ideal rules: *subr0-iff add-r0-iff*]

end

lemmas *bool-simps* = *simp-thms*(*1*−*34*)
lemma *dnf*:
$(P \,\&\, (Q \mid R)) = ((P\&Q) \mid (P\&R))$ $((Q \mid R) \,\&\, P) = ((Q\&P) \mid (R\&P))$
$(P \wedge Q) = (Q \wedge P)$ $(P \vee Q) = (Q \vee P)$
by *blast*+

lemmas *weak-dnf-simps* = *dnf bool-simps*

lemma *nnf-simps*:
$(\neg(P \wedge Q)) = (\neg P \vee \neg Q)$ $(\neg(P \vee Q)) = (\neg P \wedge \neg Q)$ $(P \longrightarrow Q) = (\neg P \vee Q)$
$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg\, Q))$ $(\neg\, \neg(P)) = P$
by *blast*+

lemma *PFalse*:
$P \equiv False \Longrightarrow \neg\, P$
$\neg\, P \Longrightarrow (P \equiv False)$
by *auto*
use *Tools/Groebner-Basis/groebner.ML*

method-setup *algebra* =
⟪
let
fun keyword k = *Scan.lift* (*Args.*$$$ *k* −− *Args.colon*) >> *K* ()
val addN = *add*
val delN = *del*
val any-keyword = *keyword addN* || *keyword delN*
val thms = *Scan.repeat* (*Scan.unless any-keyword Attrib.multi-thm*) >> *flat*;
in
((*Scan.optional* (*keyword addN* |−− *thms*) []) −−
(*Scan.optional* (*keyword delN* |−− *thms*) [])) >>
(*fn* (*add-ths*, *del-ths*) => *fn ctxt* =>
SIMPLE-METHOD' (*Groebner.algebra-tac add-ths del-ths ctxt*))
end
⟫ *solve polynomial equations over* (*semi*)*rings and ideal membership problems using Groebner bases*
declare *dvd-def*[*algebra*]

declare *dvd-eq-mod-eq-0*[*symmetric*, *algebra*]
declare *mod-div-trivial*[*algebra*]
declare *mod-mod-trivial*[*algebra*]
declare *conjunct1*[*OF DIVISION-BY-ZERO*, *algebra*]
declare *conjunct2*[*OF DIVISION-BY-ZERO*, *algebra*]
declare *zmod-zdiv-equality*[*symmetric*,*algebra*]
declare *zdiv-zmod-equality*[*symmetric*, *algebra*]
declare *zdiv-zminus-zminus*[*algebra*]
declare *zmod-zminus-zminus*[*algebra*]
declare *zdiv-zminus2*[*algebra*]
declare *zmod-zminus2*[*algebra*]
declare *zdiv-zero*[*algebra*]
declare *zmod-zero*[*algebra*]
declare *mod-by-1*[*algebra*]
declare *div-by-1*[*algebra*]
declare *zmod-minus1-right*[*algebra*]
declare *zdiv-minus1-right*[*algebra*]
declare *mod-div-trivial*[*algebra*]
declare *mod-mod-trivial*[*algebra*]
declare *mod-mult-self2-is-0*[*algebra*]
declare *mod-mult-self1-is-0*[*algebra*]
declare *zmod-eq-0-iff*[*algebra*]
declare *dvd-0-left-iff*[*algebra*]
declare *zdvd1-eq*[*algebra*]
declare *zmod-eq-dvd-iff*[*algebra*]
declare *nat-mod-eq-iff*[*algebra*]

33.3 Groebner Bases for fields

interpretation *class-fieldgb*:
*fieldgb op + op * op ^ 0*::'*a*::{*field*,*recpower*,*number-ring*} *1 op − uminus op / inverse* **apply** (*unfold-locales*) **by** (*simp-all add*: *divide-inverse*)

lemma *divide-Numeral1*: (*x*::'*a*::{*field*,*number-ring*}) / *Numeral1* = *x* **by** *simp*
lemma *divide-Numeral0*: (*x*::'*a*::{*field*,*number-ring*, *division-by-zero*}) / *Numeral0* = *0*
by *simp*
lemma *mult-frac-frac*: ((*x*::'*a*::{*field*,*division-by-zero*}) / *y*) * (*z* / *w*) = (*x***z*) / (*y***w*)
by *simp*
lemma *mult-frac-num*: ((*x*::'*a*::{*field*, *division-by-zero*}) / *y*) * *z* = (*x***z*) / *y*
by *simp*
lemma *mult-num-frac*: ((*x*::'*a*::{*field*, *division-by-zero*}) / *y*) * *z* = (*x***z*) / *y*
by *simp*

lemma *Numeral1-eq1-nat*: (*1*::*nat*) = *Numeral1* **by** *simp*

lemma *add-frac-num*: *y*≠ *0* ⟹ (*x*::'*a*::{*field*, *division-by-zero*}) / *y* + *z* = (*x* + *z***y*) / *y*

```
  by (simp add: add-divide-distrib)
lemma add-num-frac: y≠ 0 ⟹ z + (x::'a::{field, division-by-zero}) / y = (x +
z*y) / y
  by (simp add: add-divide-distrib)
ML⟪ let open Conv in fconv-rule (arg-conv (arg1-conv (rewr-conv (mk-meta-eq
@{thm mult-commute})))) (@{thm divide-inverse} RS sym)end⟫
ML⟪
local
 val zr = @{cpat 0}
 val zT = ctyp-of-term zr
 val geq = @{cpat op =}
 val eqT = Thm.dest-ctyp (ctyp-of-term geq) |> hd
 val add-frac-eq = mk-meta-eq @{thm add-frac-eq}
 val add-frac-num = mk-meta-eq @{thm add-frac-num}
 val add-num-frac = mk-meta-eq @{thm add-num-frac}

 fun prove-nz ss T t =
    let
      val z = instantiate-cterm ([(zT,T)],[]) zr
      val eq = instantiate-cterm ([(eqT,T)],[]) geq
      val th = Simplifier.rewrite (ss addsimps simp-thms)
          (Thm.capply @{cterm Trueprop} (Thm.capply @{cterm Not}
                (Thm.capply (Thm.capply eq t) z)))
    in equal-elim (symmetric th) TrueI
    end

 fun proc phi ss ct =
  let
    val ((x,y),(w,z)) =
       (Thm.dest-binop #> (fn (a,b) => (Thm.dest-binop a, Thm.dest-binop b)))
ct
    val - = map (HOLogic.dest-number o term-of) [x,y,z,w]
    val T = ctyp-of-term x
    val [y-nz, z-nz] = map (prove-nz ss T) [y, z]
    val th = instantiate' [SOME T] (map SOME [y,z,x,w]) add-frac-eq
  in SOME (implies-elim (implies-elim th y-nz) z-nz)
  end
  handle CTERM - => NONE | TERM - => NONE | THM - => NONE

 fun proc2 phi ss ct =
  let
    val (l,r) = Thm.dest-binop ct
    val T = ctyp-of-term l
  in (case (term-of l, term-of r) of
      (Const(@{const-name HOL.divide},-)$-$-, -) =>
        let val (x,y) = Thm.dest-binop l val z = r
            val - = map (HOLogic.dest-number o term-of) [x,y,z]
            val ynz = prove-nz ss T y
           in SOME (implies-elim (instantiate' [SOME T] (map SOME [y,x,z])
```

```
add-frac-num) ynz)
        end
    | (-, Const (@{const-name HOL.divide},-)$-$-) =>
        let val (x,y) = Thm.dest-binop r val z = l
            val - = map (HOLogic.dest-number o term-of) [x,y,z]
            val ynz = prove-nz ss T y
           in SOME (implies-elim (instantiate' [SOME T] (map SOME [y,z,x])
add-num-frac) ynz)
        end
    | - => NONE)
  end
  handle CTERM - => NONE | TERM - => NONE | THM - => NONE

 fun is-number (Const(@{const-name HOL.divide},-)$a$b) = is-number a andalso
is-number b
   | is-number t = can HOLogic.dest-number t

 val is-number = is-number o term-of

 fun proc3 phi ss ct =
  (case term-of ct of
   Const(@{const-name HOL.less},-)$(Const(@{const-name HOL.divide},-)$-$-)$-
=>
      let
        val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
        val - = map is-number [a,b,c]
        val T = ctyp-of-term c
        val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-less-eq}
      in SOME (mk-meta-eq th) end
 | Const(@{const-name HOL.less-eq},-)$(Const(@{const-name HOL.divide},-)$-$-)$-
=>
      let
        val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
        val - = map is-number [a,b,c]
        val T = ctyp-of-term c
        val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-le-eq}
      in SOME (mk-meta-eq th) end
  | Const(op =,-)$(Const(@{const-name HOL.divide},-)$-$-)$- =>
      let
        val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
        val - = map is-number [a,b,c]
        val T = ctyp-of-term c
        val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-eq-eq}
      in SOME (mk-meta-eq th) end
 | Const(@{const-name HOL.less},-)$-$(Const(@{const-name HOL.divide},-)$-$-)
=>
    let
      val (a,(b,c)) = Thm.dest-binop ct ||> Thm.dest-binop
        val - = map is-number [a,b,c]
```

```
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm less-divide-eq}
    in SOME (mk-meta-eq th) end
 | Const(@{const-name HOL.less-eq},-)$-$(Const(@{const-name HOL.divide},-)$-$-)
=>
    let
      val (a,(b,c)) = Thm.dest-binop ct |> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm le-divide-eq}
    in SOME (mk-meta-eq th) end
 | Const(op =,-)$-$(Const(@{const-name HOL.divide},-)$-$-) =>
    let
      val (a,(b,c)) = Thm.dest-binop ct |> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm eq-divide-eq}
    in SOME (mk-meta-eq th) end
 | - => NONE)
 handle TERM - => NONE | CTERM - => NONE | THM - => NONE

val add-frac-frac-simproc =
      make-simproc {lhss = [@{cpat (?x::?'a::field)/?y + (?w::?'a::field)/?z}],
                    name = add-frac-frac-simproc,
                    proc = proc, identifier = []}

val add-frac-num-simproc =
        make-simproc {lhss = [@{cpat (?x::?'a::field)/?y + ?z}, @{cpat ?z +
(?x::?'a::field)/?y}],
                    name = add-frac-num-simproc,
                    proc = proc2, identifier = []}

val ord-frac-simproc =
 make-simproc
   {lhss = [@{cpat (?a::(?'a::{field, ord}))/?b < ?c},
            @{cpat (?a::(?'a::{field, ord}))/?b ≤ ?c},
            @{cpat ?c < (?a::(?'a::{field, ord}))/?b},
            @{cpat ?c ≤ (?a::(?'a::{field, ord}))/?b},
            @{cpat ?c = ((?a::(?'a::{field, ord}))/?b)},
            @{cpat ((?a::(?'a::{field, ord}))/ ?b) = ?c}],
            name = ord-frac-simproc, proc = proc3, identifier = []}

local
open Conv
in

val ths = [@{thm mult-numeral-1}, @{thm mult-numeral-1-right},
           @{thm divide-Numeral1},
           @{thm Ring-and-Field.divide-zero}, @{thm divide-Numeral0},
```

```
          @{thm divide-divide-eq-left}, @{thm mult-frac-frac},
          @{thm mult-num-frac}, @{thm mult-frac-num},
          @{thm mult-frac-frac}, @{thm times-divide-eq-right},
          @{thm times-divide-eq-left}, @{thm divide-divide-eq-right},
          @{thm diff-def}, @{thm minus-divide-left},
          @{thm Numeral1-eq1-nat}, @{thm add-divide-distrib} RS sym,
          @{thm divide-inverse} RS sym, @{thm inverse-divide},
        fconv-rule (arg-conv (arg1-conv (rewr-conv (mk-meta-eq @{thm mult-commute}))))

          (@{thm divide-inverse} RS sym)]

val comp-conv = (Simplifier.rewrite
(HOL-basic-ss addsimps @{thms Groebner-Basis.comp-arith}
              addsimps ths addsimps simp-thms
              addsimprocs field-cancel-numeral-factors
               addsimprocs [add-frac-frac-simproc, add-frac-num-simproc,
                            ord-frac-simproc]
                addcongs [@{thm if-weak-cong}]))
then-conv (Simplifier.rewrite (HOL-basic-ss addsimps
  [@{thm numeral-1-eq-1},@{thm numeral-0-eq-0}] @ @{thms numerals(1-2)}))
end

fun numeral-is-const ct =
  case term-of ct of
   Const (@{const-name HOL.divide},-) $ a $ b =>
     can HOLogic.dest-number a andalso can HOLogic.dest-number b
 | Const (@{const-name HOL.inverse},-)$t => can HOLogic.dest-number t
 | t => can HOLogic.dest-number t

fun dest-const 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 HOL.inverse},-)$t =>
             Rat.inv (Rat.rat-of-int (snd (HOLogic.dest-number t)))
 | t => Rat.rat-of-int (snd (HOLogic.dest-number t)))
   handle TERM - => error ring-dest-const)

fun mk-const 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

in
 val field-comp-conv = comp-conv;
```

```
 val fieldgb-declaration =
  NormalizerData.funs @{thm class-fieldgb.fieldgb-axioms'}
   {is-const = K numeral-is-const,
    dest-const = K dest-const,
    mk-const = mk-const,
    conv = K (K comp-conv)}
end;
⟫
```

declaration *fieldgb-declaration*

end

34 SetInterval: Set intervals

theory *SetInterval*
imports *Int*
begin

context *ord*
begin
definition
  lessThan :: *'a => 'a set* ((*1*{..<-})) **where**
  {..<*u*} == {*x*. *x* < *u*}

definition
  atMost :: *'a => 'a set* ((*1*{..-})) **where**
  {..*u*} == {*x*. *x* ≤ *u*}

definition
  greaterThan :: *'a => 'a set* ((*1*{-<..})) **where**
  {*l*<..} == {*x*. *l*<*x*}

definition
  atLeast :: *'a => 'a set* ((*1*{-..})) **where**
  {*l*..} == {*x*. *l*≤*x*}

definition
  greaterThanLessThan :: *'a => 'a => 'a set* ((*1*{-<..<-})) **where**
  {*l*<..<*u*} == {*l*<..} *Int* {..<*u*}

definition
  atLeastLessThan :: *'a => 'a => 'a set* ((*1*{-..<-})) **where**
  {*l*..<*u*} == {*l*..} *Int* {..<*u*}

definition
  greaterThanAtMost :: *'a => 'a => 'a set* ((*1*{-<..-})) **where**
  {*l*<..*u*} == {*l*<..} *Int* {..*u*}

definition
atLeastAtMost :: *'a => 'a => 'a set* ((*1*{-..-})) **where**
{*l..u*} == {*l..*} *Int* {*..u*}

end

A note of warning when using {*..<n*} on type *nat*: it is equivalent to {*0..<n*} but some lemmas involving {*m..<n*} may not exist in {*..<n*}-form as well.

syntax
@*UNION-le* :: *'a => 'a => 'b set => 'b set* ((*3UN -<=-./ -*) *10*)
@*UNION-less* :: *'a => 'a => 'b set => 'b set* ((*3UN -<-./ -*) *10*)
@*INTER-le* :: *'a => 'a => 'b set => 'b set* ((*3INT -<=-./ -*) *10*)
@*INTER-less* :: *'a => 'a => 'b set => 'b set* ((*3INT -<-./ -*) *10*)

syntax (*xsymbols*)
@*UNION-le* :: *'a => 'a => 'b set => 'b set* ((*3*⋃ *-≤-./ -*) *10*)
@*UNION-less* :: *'a => 'a => 'b set => 'b set* ((*3*⋃ *-<-./ -*) *10*)
@*INTER-le* :: *'a => 'a => 'b set => 'b set* ((*3*⋂ *-≤-./ -*) *10*)
@*INTER-less* :: *'a => 'a => 'b set => 'b set* ((*3*⋂ *-<-./ -*) *10*)

syntax (*latex* **output**)
@*UNION-le* :: *'a ⇒ 'a => 'b set => 'b set* ((*3*⋃(*00- ≤ -*)*/ -*) *10*)
@*UNION-less* :: *'a ⇒ 'a => 'b set => 'b set* ((*3*⋃(*00- < -*)*/ -*) *10*)
@*INTER-le* :: *'a ⇒ 'a => 'b set => 'b set* ((*3*⋂(*00- ≤ -*)*/ -*) *10*)
@*INTER-less* :: *'a ⇒ 'a => 'b set => 'b set* ((*3*⋂(*00- < -*)*/ -*) *10*)

translations
UN i<=n. A == *UN i:*{*..n*}*. A*
UN i<n. A == *UN i:*{*..<n*}*. A*
INT i<=n. A == *INT i:*{*..n*}*. A*
INT i<n. A == *INT i:*{*..<n*}*. A*

34.1 Various equivalences

lemma (**in** *ord*) *lessThan-iff* [*iff*]: (*i*: *lessThan k*) = (*i<k*)
by (*simp add*: *lessThan-def*)

lemma *Compl-lessThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*lessThan k* = *atLeast k*
apply (*auto simp add*: *lessThan-def atLeast-def*)
done

lemma *single-Diff-lessThan* [*simp*]: !!*k*:: *'a*::*order*. {*k*} − *lessThan k* = {*k*}
by *auto*

lemma (**in** *ord*) *greaterThan-iff* [*iff*]: (*i*: *greaterThan k*) = (*k<i*)
by (*simp add*: *greaterThan-def*)

lemma *Compl-greaterThan* [*simp*]:
  !!*k*:: *'a*::*linorder*. −*greaterThan k* = *atMost k*
 by (*auto simp add*: *greaterThan-def atMost-def*)

lemma *Compl-atMost* [*simp*]: !!*k*:: *'a*::*linorder*. −*atMost k* = *greaterThan k*
apply (*subst Compl-greaterThan* [*symmetric*])
apply (*rule double-complement*)
done

lemma (**in** *ord*) *atLeast-iff* [*iff*]: (*i*: *atLeast k*) = (*k*<=*i*)
by (*simp add*: *atLeast-def*)

lemma *Compl-atLeast* [*simp*]:
  !!*k*:: *'a*::*linorder*. −*atLeast k* = *lessThan k*
 by (*auto simp add*: *lessThan-def atLeast-def*)

lemma (**in** *ord*) *atMost-iff* [*iff*]: (*i*: *atMost k*) = (*i*<=*k*)
by (*simp add*: *atMost-def*)

lemma *atMost-Int-atLeast*: !!*n*:: *'a*::*order*. *atMost n Int atLeast n* = {*n*}
by (*blast intro*: *order-antisym*)

34.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-subset-iff* [*iff*]:
  (*atLeast x* ⊆ *atLeast y*) = (*y* ≤ (*x*::*'a*::*order*))
by (*blast intro*: *order-trans*)

lemma *atLeast-eq-iff* [*iff*]:
  (*atLeast x* = *atLeast y*) = (*x* = (*y*::*'a*::*linorder*))
by (*blast intro*: *order-antisym order-trans*)

lemma *greaterThan-subset-iff* [*iff*]:
  (*greaterThan x* ⊆ *greaterThan y*) = (*y* ≤ (*x*::*'a*::*linorder*))
apply (*auto simp add*: *greaterThan-def*)
 apply (*subst linorder-not-less* [*symmetric*], *blast*)
done

lemma *greaterThan-eq-iff* [*iff*]:
  (*greaterThan x* = *greaterThan y*) = (*x* = (*y*::*'a*::*linorder*))
apply (*rule iffI*)
 apply (*erule equalityE*)
 apply *simp-all*
done

lemma *atMost-subset-iff* [*iff*]: (*atMost x* ⊆ *atMost y*) = (*x* ≤ (*y*::*'a*::*order*))
by (*blast intro*: *order-trans*)

lemma *atMost-eq-iff* [*iff*]: (*atMost x* = *atMost y*) = (*x* = (*y*::*'a*::*linorder*))

by (*blast intro*: *order-antisym order-trans*)

lemma *lessThan-subset-iff* [*iff*]:
(*lessThan x* ⊆ *lessThan y*) = (*x* ≤ (*y*::'*a*::*linorder*))
apply (*auto simp add*: *lessThan-def*)
apply (*subst linorder-not-less* [*symmetric*], *blast*)
done

lemma *lessThan-eq-iff* [*iff*]:
(*lessThan x* = *lessThan y*) = (*x* = (*y*::'*a*::*linorder*))
apply (*rule iffI*)
apply (*erule equalityE*)
apply *simp-all*
done

34.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [*simp*,*noatp*]:
(*i* : {*l*<..<*u*}) = (*l* < *i* & *i* < *u*)
by (*simp add*: *greaterThanLessThan-def*)

lemma *atLeastLessThan-iff* [*simp*,*noatp*]:
(*i* : {*l*..<*u*}) = (*l* <= *i* & *i* < *u*)
by (*simp add*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-iff* [*simp*,*noatp*]:
(*i* : {*l*<..*u*}) = (*l* < *i* & *i* <= *u*)
by (*simp add*: *greaterThanAtMost-def*)

lemma *atLeastAtMost-iff* [*simp*,*noatp*]:
(*i* : {*l*..*u*}) = (*l* <= *i* & *i* <= *u*)
by (*simp add*: *atLeastAtMost-def*)

The above four lemmas could be declared as iffs. If we do so, a call to blast in Hyperreal/Star.ML, lemma *STAR-Int* seems to take forever (more than one hour).

end

34.3.1 Emptyness and singletons

context *order*
begin

lemma *atLeastAtMost-empty* [*simp*]: *n* < *m* ==> {*m*..*n*} = {}
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

lemma *atLeastLessThan-empty*[*simp*]: $n \leq m \Longrightarrow \{m..<n\} = \{\}$
by (*auto simp add*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-empty*[*simp*]:$l \leq k \Longrightarrow \{k<..l\} = \{\}$
by(*auto simp*:*greaterThanAtMost-def greaterThan-def atMost-def*)

lemma *greaterThanLessThan-empty*[*simp*]:$l \leq k \Longrightarrow \{k<..<l\} = \{\}$
by(*auto simp*:*greaterThanLessThan-def greaterThan-def lessThan-def*)

lemma *atLeastAtMost-singleton* [*simp*]: $\{a..a\} = \{a\}$
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

end

34.4 Intervals of natural numbers

34.4.1 The Constant *lessThan*

lemma *lessThan-0* [*simp*]: *lessThan* (*0*::*nat*) = {}
by (*simp add*: *lessThan-def*)

lemma *lessThan-Suc*: *lessThan* (*Suc k*) = *insert k* (*lessThan k*)
by (*simp add*: *lessThan-def less-Suc-eq*, *blast*)

lemma *lessThan-Suc-atMost*: *lessThan* (*Suc k*) = *atMost k*
by (*simp add*: *lessThan-def atMost-def less-Suc-eq-le*)

lemma *UN-lessThan-UNIV*: (*UN m*::*nat*. *lessThan m*) = *UNIV*
by *blast*

34.4.2 The Constant *greaterThan*

lemma *greaterThan-0* [*simp*]: *greaterThan 0* = *range Suc*
apply (*simp add*: *greaterThan-def*)
apply (*blast dest*: *gr0-conv-Suc* [*THEN iffD1*])
done

lemma *greaterThan-Suc*: *greaterThan* (*Suc k*) = *greaterThan k* − {*Suc k*}
apply (*simp add*: *greaterThan-def*)
apply (*auto elim*: *linorder-neqE*)
done

lemma *INT-greaterThan-UNIV*: (*INT m*::*nat*. *greaterThan m*) = {}
by *blast*

34.4.3 The Constant *atLeast*

lemma *atLeast-0* [*simp*]: *atLeast* (*0*::*nat*) = *UNIV*
by (*unfold atLeast-def UNIV-def*, *simp*)

lemma *atLeast-Suc*: *atLeast* (*Suc k*) = *atLeast k* − {*k*}
apply (*simp add*: *atLeast-def*)
apply (*simp add*: *Suc-le-eq*)
apply (*simp add*: *order-le-less*, *blast*)
done

lemma *atLeast-Suc-greaterThan*: *atLeast* (*Suc k*) = *greaterThan k*
by (*auto simp add*: *greaterThan-def atLeast-def less-Suc-eq-le*)

lemma *UN-atLeast-UNIV*: (*UN m*::*nat*. *atLeast m*) = *UNIV*
by *blast*

34.4.4 The Constant *atMost*

lemma *atMost-0* [*simp*]: *atMost* (*0*::*nat*) = {*0*}
by (*simp add*: *atMost-def*)

lemma *atMost-Suc*: *atMost* (*Suc k*) = *insert* (*Suc k*) (*atMost k*)
apply (*simp add*: *atMost-def*)
apply (*simp add*: *less-Suc-eq order-le-less*, *blast*)
done

lemma *UN-atMost-UNIV*: (*UN m*::*nat*. *atMost m*) = *UNIV*
by *blast*

34.4.5 The Constant *atLeastLessThan*

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

lemma *atLeast0LessThan*: {*0*::*nat*..<*n*} = {..<*n*}
by(*simp add*:*lessThan-def atLeastLessThan-def*)

lemma *atLeast0AtMost*: {*0*..*n*::*nat*} = {..*n*}
by(*simp add*:*atMost-def atLeastAtMost-def*)

declare *atLeast0LessThan*[*symmetric*, *code unfold*]
atLeast0AtMost[*symmetric*, *code unfold*]

lemma *atLeastLessThan0*: {*m*..<*0*::*nat*} = {}
by (*simp add*: *atLeastLessThan-def*)

34.4.6 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:

$\{m..<Suc\ n\} = (if\ m \leq n\ then\ insert\ n\ \{m..<n\}\ else\ \{\})$
by (*auto simp add*: *atLeastLessThan-def*)

lemma *atLeastLessThan-singleton* [*simp*]: $\{m..<Suc\ m\} = \{m\}$
by (*auto simp add*: *atLeastLessThan-def*)

lemma *atLeastLessThanSuc-atLeastAtMost*: $\{l..<Suc\ u\} = \{l..u\}$
by (*simp add*: *lessThan-Suc-atMost atLeastAtMost-def atLeastLessThan-def*)

lemma *atLeastSucAtMost-greaterThanAtMost*: $\{Suc\ l..u\} = \{l<..u\}$
by (*simp add*: *atLeast-Suc-greaterThan atLeastAtMost-def*
greaterThanAtMost-def)

lemma *atLeastSucLessThan-greaterThanLessThan*: $\{Suc\ l..<u\} = \{l<..<u\}$
by (*simp add*: *atLeast-Suc-greaterThan atLeastLessThan-def*
greaterThanLessThan-def)

lemma *atLeastAtMostSuc-conv*: $m \leq Suc\ n \Longrightarrow \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$
by (*auto simp add*: *atLeastAtMost-def*)

34.4.7 Image

lemma *image-add-atLeastAtMost*:
$(\%n{::}nat.\ n+k)\ `\ \{i..j\} = \{i+k..j+k\}$ (**is** *?A* = *?B*)
proof
show *?A* $\subseteq$ *?B* **by** *auto*
next
show *?B* $\subseteq$ *?A*
proof
fix *n* **assume** *a*: *n* : *?B*
hence $n - k : \{i..j\}$ **by** *auto*
moreover have $n = (n - k) + k$ **using** *a* **by** *auto*
ultimately show *n* : *?A* **by** *blast*
qed
qed

lemma *image-add-atLeastLessThan*:
$(\%n{::}nat.\ n+k)\ `\ \{i..<j\} = \{i+k..<j+k\}$ (**is** *?A* = *?B*)
proof
show *?A* $\subseteq$ *?B* **by** *auto*
next
show *?B* $\subseteq$ *?A*
proof
fix *n* **assume** *a*: *n* : *?B*
hence $n - k : \{i..<j\}$ **by** *auto*
moreover have $n = (n - k) + k$ **using** *a* **by** *auto*
ultimately show *n* : *?A* **by** *blast*
qed

qed

corollary *image-Suc-atLeastAtMost*[*simp*]:
Suc ' {i..j} = {Suc i..Suc j}
using *image-add-atLeastAtMost*[**where** *k=Suc 0*] **by** *simp*

corollary *image-Suc-atLeastLessThan*[*simp*]:
Suc ' {i..<j} = {Suc i..<Suc j}
using *image-add-atLeastLessThan*[**where** *k=Suc 0*] **by** *simp*

lemma *image-add-int-atLeastLessThan*:
(%x. x + (l::int)) ' {0..<u-l} = {l..<u}
apply (*auto simp add*: *image-def*)
apply (*rule-tac x = x - l* **in** *bexI*)
apply *auto*
done

34.4.8 Finiteness

lemma *finite-lessThan* [*iff*]: **fixes** *k* :: *nat* **shows** *finite* {..<*k*}
by (*induct k*) (*simp-all add*: *lessThan-Suc*)

lemma *finite-atMost* [*iff*]: **fixes** *k* :: *nat* **shows** *finite* {..*k*}
by (*induct k*) (*simp-all add*: *atMost-Suc*)

lemma *finite-greaterThanLessThan* [*iff*]:
fixes *l* :: *nat* **shows** *finite* {*l*<..<*u*}
by (*simp add*: *greaterThanLessThan-def*)

lemma *finite-atLeastLessThan* [*iff*]:
fixes *l* :: *nat* **shows** *finite* {*l*..<*u*}
by (*simp add*: *atLeastLessThan-def*)

lemma *finite-greaterThanAtMost* [*iff*]:
fixes *l* :: *nat* **shows** *finite* {*l*<..*u*}
by (*simp add*: *greaterThanAtMost-def*)

lemma *finite-atLeastAtMost* [*iff*]:
fixes *l* :: *nat* **shows** *finite* {*l*..*u*}
by (*simp add*: *atLeastAtMost-def*)

A bounded set of natural numbers is finite.

lemma *bounded-nat-set-is-finite*:
(ALL i:N. i < (n::nat)) ==> finite N
apply (*rule finite-subset*)
apply (*rule-tac* [*2*] *finite-lessThan*, *auto*)
done

lemma *finite-less-ub*:

!!f::nat=>nat. (!!n. $n \leq f\ n$) ==> *finite* {n. $f\ n \leq u$}
by (*rule-tac* B={..u} **in** *finite-subset*, *auto intro*: *order-trans*)

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

lemma *subset-card-intvl-is-intvl*:
$A <= \{k..<k+card\ A\} \Longrightarrow A = \{k..<k+card\ A\}$ (**is** *PROP ?P*)
proof *cases*
assume *finite A*
thus *PROP ?P*
proof(*induct A rule*:*finite-linorder-induct*)
case *empty* **thus** *?case* **by** *auto*
next
case (*insert A b*)
moreover hence b ~: A **by** *auto*
moreover have $A <= \{k..<k+card\ A\}$ **and** $b = k+card\ A$
using ‹b ~: A› *insert* **by** *fastsimp+*
ultimately show *?case* **by** *auto*
qed
next
assume ~*finite A* **thus** *PROP ?P* **by** *simp*
qed

34.4.9 Cardinality

lemma *card-lessThan* [*simp*]: *card* {..<u} = u
by (*induct u*, *simp-all add*: *lessThan-Suc*)

lemma *card-atMost* [*simp*]: *card* {..u} = *Suc u*
by (*simp add*: *lessThan-Suc-atMost* [*THEN sym*])

lemma *card-atLeastLessThan* [*simp*]: *card* {l..<u} = $u - l$
apply (*subgoal-tac card* {l..<u} = *card* {..<$u-l$})
apply (*erule ssubst*, *rule card-lessThan*)
apply (*subgoal-tac* (%x. $x + l$) ‘ {..<$u-l$} = {l..<u})
apply (*erule subst*)
apply (*rule card-image*)
apply (*simp add*: *inj-on-def*)
apply (*auto simp add*: *image-def atLeastLessThan-def lessThan-def*)
apply (*rule-tac* $x = x - l$ **in** *exI*)
apply *arith*
done

lemma *card-atLeastAtMost* [*simp*]: *card* {l..u} = *Suc* $u - l$
by (*subst atLeastLessThanSuc-atLeastAtMost* [*THEN sym*], *simp*)

lemma *card-greaterThanAtMost* [*simp*]: *card* {l<..u} = $u - l$
by (*subst atLeastSucAtMost-greaterThanAtMost* [*THEN sym*], *simp*)

lemma *card-greaterThanLessThan* [*simp*]: *card* {*l*<..<*u*} = *u* − *Suc l*
by (*subst atLeastSucLessThan-greaterThanLessThan* [*THEN sym*], *simp*)

lemma *ex-bij-betw-nat-finite*:
finite M ⟹ ∃ *h*. *bij-betw h* {*0*..<*card M*} *M*
apply(*drule finite-imp-nat-seg-image-inj-on*)
apply(*auto simp*:*atLeast0LessThan*[*symmetric*] *lessThan-def*[*symmetric*] *card-image bij-betw-def*)
done

lemma *ex-bij-betw-finite-nat*:
finite M ⟹ ∃ *h*. *bij-betw h M* {*0*..<*card M*}
by (*blast dest*: *ex-bij-betw-nat-finite bij-betw-inv*)

34.5 Intervals of integers

lemma *atLeastLessThanPlusOne-atLeastAtMost-int*: {*l*..<*u*+*1*} = {*l*..(*u*::*int*)}
by (*auto simp add*: *atLeastAtMost-def atLeastLessThan-def*)

lemma *atLeastPlusOneAtMost-greaterThanAtMost-int*: {*l*+*1*..*u*} = {*l*<..(*u*::*int*)}
by (*auto simp add*: *atLeastAtMost-def greaterThanAtMost-def*)

lemma *atLeastPlusOneLessThan-greaterThanLessThan-int*:
{*l*+*1*..<*u*} = {*l*<..<*u*::*int*}
by (*auto simp add*: *atLeastLessThan-def greaterThanLessThan-def*)

34.5.1 Finiteness

lemma *image-atLeastZeroLessThan-int*: *0* ≤ *u* ==>
{(*0*::*int*)..<*u*} = *int* ‘ {..<*nat u*}
apply (*unfold image-def lessThan-def*)
apply *auto*
apply (*rule-tac x* = *nat x* **in** *exI*)
apply (*auto simp add*: *zless-nat-conj zless-nat-eq-int-zless* [*THEN sym*])
done

lemma *finite-atLeastZeroLessThan-int*: *finite* {(*0*::*int*)..<*u*}
apply (*case-tac* *0* ≤ *u*)
apply (*subst image-atLeastZeroLessThan-int*, *assumption*)
apply (*rule finite-imageI*)
apply *auto*
done

lemma *finite-atLeastLessThan-int* [*iff*]: *finite* {*l*..<*u*::*int*}
apply (*subgoal-tac* (%*x*. *x* + *l*) ‘ {*0*..<*u*−*l*} = {*l*..<*u*})
apply (*erule subst*)
apply (*rule finite-imageI*)
apply (*rule finite-atLeastZeroLessThan-int*)
apply (*rule image-add-int-atLeastLessThan*)

done

lemma *finite-atLeastAtMost-int* [*iff*]: *finite* {*l*..(*u*::*int*)}
by (*subst atLeastLessThanPlusOne-atLeastAtMost-int* [*THEN sym*], *simp*)

lemma *finite-greaterThanAtMost-int* [*iff*]: *finite* {*l*<..(*u*::*int*)}
by (*subst atLeastPlusOneAtMost-greaterThanAtMost-int* [*THEN sym*], *simp*)

lemma *finite-greaterThanLessThan-int* [*iff*]: *finite* {*l*<..<*u*::*int*}
by (*subst atLeastPlusOneLessThan-greaterThanLessThan-int* [*THEN sym*], *simp*)

34.5.2 Cardinality

lemma *card-atLeastZeroLessThan-int*: *card* {(*0*::*int*)..<*u*} = *nat u*
apply (*case-tac* 0 ≤ *u*)
apply (*subst image-atLeastZeroLessThan-int*, *assumption*)
apply (*subst card-image*)
apply (*auto simp add*: *inj-on-def*)
done

lemma *card-atLeastLessThan-int* [*simp*]: *card* {*l*..<*u*} = *nat* (*u* − *l*)
apply (*subgoal-tac card* {*l*..<*u*} = *card* {*0*..<*u*−*l*})
apply (*erule ssubst*, *rule card-atLeastZeroLessThan-int*)
apply (*subgoal-tac* (%*x*. *x* + *l*) ' {*0*..<*u*−*l*} = {*l*..<*u*})
apply (*erule subst*)
apply (*rule card-image*)
apply (*simp add*: *inj-on-def*)
apply (*rule image-add-int-atLeastLessThan*)
done

lemma *card-atLeastAtMost-int* [*simp*]: *card* {*l*..*u*} = *nat* (*u* − *l* + *1*)
apply (*subst atLeastLessThanPlusOne-atLeastAtMost-int* [*THEN sym*])
apply (*auto simp add*: *algebra-simps*)
done

lemma *card-greaterThanAtMost-int* [*simp*]: *card* {*l*<..*u*} = *nat* (*u* − *l*)
by (*subst atLeastPlusOneAtMost-greaterThanAtMost-int* [*THEN sym*], *simp*)

lemma *card-greaterThanLessThan-int* [*simp*]: *card* {*l*<..<*u*} = *nat* (*u* − (*l* + *1*))
by (*subst atLeastPlusOneLessThan-greaterThanLessThan-int* [*THEN sym*], *simp*)

lemma *finite-M-bounded-by-nat*: *finite* {*k*. *P k* ∧ *k* < (*i*::*nat*)}
proof −
have {*k*. *P k* ∧ *k* < *i*} ⊆ {..<*i*} **by** *auto*
with *finite-lessThan*[*of i*] **show** *?thesis* **by** (*simp add*: *finite-subset*)
qed

lemma *card-less*:
assumes *zero-in-M*: *0* ∈ *M*

shows *card* $\{k \in M.\ k < Suc\ i\} \neq 0$
proof −
from *zero-in-M* **have** $\{k \in M.\ k < Suc\ i\} \neq \{\}$ **by** *auto*
with *finite-M-bounded-by-nat* **show** *?thesis* **by** (*auto simp add*: *card-eq-0-iff*)
qed

lemma *card-less-Suc2*: $0 \notin M \Longrightarrow card\ \{k.\ Suc\ k \in M \wedge k < i\} = card\ \{k \in M.\ k < Suc\ i\}$
apply (*rule card-bij-eq* [*of Suc* - - $\lambda x.\ x - Suc\ 0$])
apply *simp*
apply *fastsimp*
apply *auto*
apply (*rule inj-on-diff-nat*)
apply *auto*
apply (*case-tac x*)
apply *auto*
apply (*case-tac xa*)
apply *auto*
apply (*case-tac xa*)
apply *auto*
done

lemma *card-less-Suc*:
assumes *zero-in-M*: $0 \in M$
shows $Suc\ (card\ \{k.\ Suc\ k \in M \wedge k < i\}) = card\ \{k \in M.\ k < Suc\ i\}$
proof −
from *assms* **have** *a*: $0 \in \{k \in M.\ k < Suc\ i\}$ **by** *simp*
hence *c*: $\{k \in M.\ k < Suc\ i\} = insert\ 0\ (\{k \in M.\ k < Suc\ i\} - \{0\})$
by (*auto simp only*: *insert-Diff*)
have *b*: $\{k \in M.\ k < Suc\ i\} - \{0\} = \{k \in M - \{0\}.\ k < Suc\ i\}$ **by** *auto*
from *finite-M-bounded-by-nat*[*of* $\lambda x.\ x \in M\ Suc\ i$] **have** $Suc\ (card\ \{k.\ Suc\ k \in M \wedge k < i\}) = card\ (insert\ 0\ (\{k \in M.\ k < Suc\ i\} - \{0\}))$
apply (*subst card-insert*)
apply *simp-all*
apply (*subst b*)
apply (*subst card-less-Suc2*[*symmetric*])
apply *simp-all*
done
with *c* **show** *?thesis* **by** *simp*
qed

34.6 Lemmas useful with the summation operator setsum

For examples, see Algebra/poly/UnivPoly2.thy

34.6.1 Disjoint Unions

Singletons and open intervals

lemma *ivl-disj-un-singleton*:
$\{l::'a::linorder\}\ Un\ \{l<..\} = \{l..\}$
$\{..<u\}\ Un\ \{u::'a::linorder\} = \{..u\}$
$(l::'a::linorder) < u ==> \{l\}\ Un\ \{l<..<u\} = \{l..<u\}$
$(l::'a::linorder) < u ==> \{l<..<u\}\ Un\ \{u\} = \{l<..u\}$
$(l::'a::linorder) <= u ==> \{l\}\ Un\ \{l<..u\} = \{l..u\}$
$(l::'a::linorder) <= u ==> \{l..<u\}\ Un\ \{u\} = \{l..u\}$
by *auto*

One- and two-sided intervals

lemma *ivl-disj-un-one*:
$(l::'a::linorder) < u ==> \{..l\}\ Un\ \{l<..<u\} = \{..<u\}$
$(l::'a::linorder) <= u ==> \{..<l\}\ Un\ \{l..<u\} = \{..<u\}$
$(l::'a::linorder) <= u ==> \{..l\}\ Un\ \{l<..u\} = \{..u\}$
$(l::'a::linorder) <= u ==> \{..<l\}\ Un\ \{l..u\} = \{..u\}$
$(l::'a::linorder) <= u ==> \{l<..u\}\ Un\ \{u<..\} = \{l<..\}$
$(l::'a::linorder) < u ==> \{l<..<u\}\ Un\ \{u..\} = \{l<..\}$
$(l::'a::linorder) <= u ==> \{l..u\}\ Un\ \{u<..\} = \{l..\}$
$(l::'a::linorder) <= u ==> \{l..<u\}\ Un\ \{u..\} = \{l..\}$
by *auto*

Two- and two-sided intervals

lemma *ivl-disj-un-two*:
$[|\ (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..<u\} = \{l<..<u\}$
$[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l<..m\}\ Un\ \{m<..<u\} = \{l<..<u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..<u\} = \{l..<u\}$
$[|\ (l::'a::linorder) <= m;\ m < u\ |] ==> \{l..m\}\ Un\ \{m<..<u\} = \{l..<u\}$
$[|\ (l::'a::linorder) < m;\ m <= u\ |] ==> \{l<..<m\}\ Un\ \{m..u\} = \{l<..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l<..m\}\ Un\ \{m<..u\} = \{l<..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..<m\}\ Un\ \{m..u\} = \{l..u\}$
$[|\ (l::'a::linorder) <= m;\ m <= u\ |] ==> \{l..m\}\ Un\ \{m<..u\} = \{l..u\}$
by *auto*

lemmas *ivl-disj-un = ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two*

34.6.2 Disjoint Intersections

Singletons and open intervals

lemma *ivl-disj-int-singleton*:
$\{l::'a::order\}\ Int\ \{l<..\} = \{\}$
$\{..<u\}\ Int\ \{u\} = \{\}$
$\{l\}\ Int\ \{l<..<u\} = \{\}$
$\{l<..<u\}\ Int\ \{u\} = \{\}$
$\{l\}\ Int\ \{l<..u\} = \{\}$
$\{l..<u\}\ Int\ \{u\} = \{\}$
by *simp+*

One- and two-sided intervals

lemma *ivl-disj-int-one*:
$\{..l::'a::order\}$ *Int* $\{l<..<u\} = \{\}$
$\{..<l\}$ *Int* $\{l..<u\} = \{\}$
$\{..l\}$ *Int* $\{l<..u\} = \{\}$
$\{..<l\}$ *Int* $\{l..u\} = \{\}$
$\{l<..u\}$ *Int* $\{u<..\} = \{\}$
$\{l<..<u\}$ *Int* $\{u..\} = \{\}$
$\{l..u\}$ *Int* $\{u<..\} = \{\}$
$\{l..<u\}$ *Int* $\{u..\} = \{\}$
by *auto*

Two- and two-sided intervals

lemma *ivl-disj-int-two*:
$\{l::'a::order<..<m\}$ *Int* $\{m..<u\} = \{\}$
$\{l<..m\}$ *Int* $\{m<..<u\} = \{\}$
$\{l..<m\}$ *Int* $\{m..<u\} = \{\}$
$\{l..m\}$ *Int* $\{m<..<u\} = \{\}$
$\{l<..<m\}$ *Int* $\{m..u\} = \{\}$
$\{l<..m\}$ *Int* $\{m<..u\} = \{\}$
$\{l..<m\}$ *Int* $\{m..u\} = \{\}$
$\{l..m\}$ *Int* $\{m<..u\} = \{\}$
by *auto*

lemmas *ivl-disj-int* = *ivl-disj-int-singleton ivl-disj-int-one ivl-disj-int-two*

34.6.3 Some Differences

lemma *ivl-diff*[*simp*]:
$i \le n \Longrightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$
by(*auto*)

34.6.4 Some Subset Conditions

lemma *ivl-subset* [*simp*,*noatp*]:
$(\{i..<j\} \subseteq \{m..<n\}) = (j \le i \mid m \le i \ \&\ j \le (n::'a::linorder))$
apply(*auto simp*:*linorder-not-le*)
apply(*rule ccontr*)
apply(*insert linorder-le-less-linear*[*of i n*])
apply(*clarsimp simp*:*linorder-not-le*)
apply(*fastsimp*)
done

34.7 Summation indexed over intervals

syntax
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*SUM - = -..-./ -*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*SUM - = -..<-./ -*) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*SUM -<-./ -*) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*SUM -<=-./ -*) [*0*,*0*,*10*] *10*)

syntax (*xsymbols*)
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ - = -..<-./ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ -<-./ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ -≤-./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ - = -..-./ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ - = -..<-./ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ -<-./ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∑ -≤-./ -) [*0*,*0*,*10*] *10*)
syntax (*latex-sum* **output**)
-from-to-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∑_- = _ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∑_-^<- = _ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∑_- < _ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∑_- ≤ _ -) [*0*,*0*,*10*] *10*)

translations
∑ *x*=*a*..*b*. *t* == *CONST setsum* (%*x*. *t*) {*a*..*b*}
∑ *x*=*a*..<*b*. *t* == *CONST setsum* (%*x*. *t*) {*a*..<*b*}
∑ *i*≤*n*. *t* == *CONST setsum* (λ*i*. *t*) {..*n*}
∑ *i*<*n*. *t* == *CONST setsum* (λ*i*. *t*) {..<*n*}

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	LaTeX
∑ *x*∈{*a*..*b*}. *e*	∑ *x* = *a*..*b*. *e*	$\sum_{x = a}^{b} e$
∑ *x*∈{*a*..<*b*}. *e*	∑ *x* = *a*..<*b*. *e*	$\sum_{x = a}^{<b} e$
∑ *x*∈{..*b*}. *e*	∑ *x*≤*b*. *e*	$\sum_{x \le b} e$
∑ *x*∈{..<*b*}. *e*	∑ *x*<*b*. *e*	$\sum_{x < b} e$

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to *latex-sum* (e.g. via *mode* = *latex-sum* in antiquotations). It is not the default LaTeX output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on *nat* it is better to use ∑ *x* = *0*..<*n*. *e* rather than ∑ *x*<*n*. *e*: *setsum* may not provide all lemmas available for {*m*..<*n*} also in the special form for {..<*n*}.

This congruence rule should be used for sums over intervals as the standard

theorem *setsum-cong* does not work well with the simplifier who adds the unsimplified premise $x \in B$ to the context.

lemma *setsum-ivl-cong*:
$[\![a = c;\ b = d;\ !!x.\ [\![\ c \le x;\ x < d\ ]\!] \Longrightarrow f\ x = g\ x\ ]\!] \Longrightarrow$
setsum f {a..<b} = setsum g {c..<d}
by(*rule setsum-cong*, *simp-all*)

lemma *setsum-atMost-Suc*[*simp*]: $(\sum i \le Suc\ n.\ f\ i) = (\sum i \le n.\ f\ i) + f(Suc\ n)$
by (*simp add*:*atMost-Suc add-ac*)

lemma *setsum-lessThan-Suc*[*simp*]: $(\sum i < Suc\ n.\ f\ i) = (\sum i < n.\ f\ i) + f\ n$
by (*simp add*:*lessThan-Suc add-ac*)

lemma *setsum-cl-ivl-Suc*[*simp*]:
setsum f {m..Suc n} = (if Suc n < m then 0 else setsum f {m..n} + f(Suc n))
by (*auto simp*:*add-ac atLeastAtMostSuc-conv*)

lemma *setsum-op-ivl-Suc*[*simp*]:
setsum f {m..<Suc n} = (if n < m then 0 else setsum f {m..<n} + f(n))
by (*auto simp*:*add-ac atLeastLessThanSuc*)

lemma *setsum-head*:
fixes *n* :: *nat*
assumes *mn*: *m <= n*
shows $(\sum x{\in}\{m..n\}.\ P\ x) = P\ m + (\sum x{\in}\{m<..n\}.\ P\ x)$ (**is** *?lhs = ?rhs*)
proof −
from *mn*
have *{m..n} = {m} ∪ {m<..n}*
by (*auto intro*: *ivl-disj-un-singleton*)
hence *?lhs* $= (\sum x{\in}\{m\} \cup \{m<..n\}.\ P\ x)$
by (*simp add*: *atLeast0LessThan*)
also have *... = ?rhs* **by** *simp*
finally show *?thesis* **.**
qed

lemma *setsum-head-Suc*:
m ≤ n ⟹ setsum f {m..n} = f m + setsum f {Suc m..n}
by (*simp add*: *setsum-head atLeastSucAtMost-greaterThanAtMost*)

lemma *setsum-head-upt-Suc*:
m < n ⟹ setsum f {m..<n} = f m + setsum f {Suc m..<n}
apply(*insert setsum-head-Suc*[*of m n − Suc 0 f*])
apply (*simp add*: *atLeastLessThanSuc-atLeastAtMost*[*symmetric*] *algebra-simps*)
done

lemma *setsum-add-nat-ivl*: $[\![m \leq n; n \leq p]\!] \Longrightarrow$
setsum f {m..<n} + setsum f {n..<p} = setsum f {m..<p::nat}
by (*simp add:setsum-Un-disjoint*[*symmetric*] *ivl-disj-int ivl-disj-un*)

lemma *setsum-diff-nat-ivl*:
fixes *f* :: *nat* ⇒ *'a::ab-group-add*
shows $[\![m \leq n; n \leq p]\!] \Longrightarrow$
setsum f {m..<p} − setsum f {m..<n} = setsum f {n..<p}
using *setsum-add-nat-ivl* [*of m n p f*,*symmetric*]
apply (*simp add*: *add-ac*)
done

34.8 Shifting bounds

lemma *setsum-shift-bounds-nat-ivl*:
setsum f {m+k..<n+k} = setsum (%i. f(i + k)){m..<n::nat}
by (*induct n*, *auto simp:atLeastLessThanSuc*)

lemma *setsum-shift-bounds-cl-nat-ivl*:
setsum f {m+k..n+k} = setsum (%i. f(i + k)){m..n::nat}
apply (*insert setsum-reindex*[*OF inj-on-add-nat*, **where** *h=f* **and** *B* = {*m..n*}])
apply (*simp add:image-add-atLeastAtMost o-def*)
done

corollary *setsum-shift-bounds-cl-Suc-ivl*:
setsum f {Suc m..Suc n} = setsum (%i. f(Suc i)){m..n}
by (*simp add:setsum-shift-bounds-cl-nat-ivl*[**where** *k=Suc 0*, *simplified*])

corollary *setsum-shift-bounds-Suc-ivl*:
setsum f {Suc m..<Suc n} = setsum (%i. f(Suc i)){m..<n}
by (*simp add:setsum-shift-bounds-nat-ivl*[**where** *k=Suc 0*, *simplified*])

lemma *setsum-shift-lb-Suc0-0*:
f(0::nat) = (0::nat) ⟹ setsum f {Suc 0..k} = setsum f {0..k}
by(*simp add:setsum-head-Suc*)

lemma *setsum-shift-lb-Suc0-0-upt*:
f(0::nat) = 0 ⟹ setsum f {Suc 0..<k} = setsum f {0..<k}
apply(*cases k*)**apply** *simp*
apply(*simp add:setsum-head-upt-Suc*)
done

34.9 The formula for geometric sums

lemma *geometric-sum*:
x ~= 1 ==> (∑ i=0..<n. x ^ i) =
(x ^ n − 1) / (x − 1::'a::{field, recpower})
by (*induct n*) (*simp-all add:field-simps power-Suc*)

34.10 The formula for arithmetic sums

lemma *gauss-sum*:
$((1::'a::comm\text{-}semiring\text{-}1) + 1)*(\sum i{\in}\{1..n\}.\ of\text{-}nat\ i) =$
$of\text{-}nat\ n*((of\text{-}nat\ n)+1)$
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case* **by** (*simp add*: *algebra-simps*)
qed

theorem *arith-series-general*:
$((1::'a::comm\text{-}semiring\text{-}1) + 1) * (\sum i{\in}\{..<n\}.\ a + of\text{-}nat\ i * d) =$
$of\text{-}nat\ n * (a + (a + of\text{-}nat(n - 1)*d))$
proof *cases*
assume *ngt1*: $n > 1$
let $?I = \lambda i.\ of\text{-}nat\ i$ **and** $?n = of\text{-}nat\ n$
have
$(\sum i{\in}\{..<n\}.\ a+?I\ i*d) =$
$((\sum i{\in}\{..<n\}.\ a) + (\sum i{\in}\{..<n\}.\ ?I\ i*d))$
by (*rule setsum-addf*)
also from *ngt1* **have** $\ldots = ?n*a + (\sum i{\in}\{..<n\}.\ ?I\ i*d)$ **by** *simp*
also from *ngt1* **have** $\ldots = (?n*a + d*(\sum i{\in}\{1..<n\}.\ ?I\ i))$
unfolding *One-nat-def*
by (*simp add*: *setsum-right-distrib atLeast0LessThan*[*symmetric*] *setsum-shift-lb-Suc0-0-upt mult-ac*)
also have $(1+1)*\ldots = (1+1)*?n*a + d*(1+1)*(\sum i{\in}\{1..<n\}.\ ?I\ i)$
by (*simp add*: *left-distrib right-distrib*)
also from *ngt1* **have** $\{1..<n\} = \{1..n - 1\}$
by (*cases n*) (*auto simp*: *atLeastLessThanSuc-atLeastAtMost*)
also from *ngt1*
have $(1+1)*?n*a + d*(1+1)*(\sum i{\in}\{1..n - 1\}.\ ?I\ i) = ((1+1)*?n*a + d*?I\ (n - 1)*?I\ n)$
by (*simp only*: *mult-ac gauss-sum* [*of n − 1*], *unfold One-nat-def*)
(*simp add*: *mult-ac trans* [*OF add-commute of-nat-Suc* [*symmetric*]])
finally show *?thesis* **by** (*simp add*: *algebra-simps*)
next
assume $\neg(n > 1)$
hence $n = 1 \vee n = 0$ **by** *auto*
thus *?thesis* **by** (*auto simp*: *algebra-simps*)
qed

lemma *arith-series-nat*:
$Suc\ (Suc\ 0) * (\sum i{\in}\{..<n\}.\ a+i*d) = n * (a + (a+(n - 1)*d))$
proof −
have
$((1::nat) + 1) * (\sum i{\in}\{..<n::nat\}.\ a + of\text{-}nat(i)*d) =$
$of\text{-}nat(n) * (a + (a + of\text{-}nat(n - 1)*d))$

by (*rule arith-series-general*)
thus *?thesis*
unfolding *One-nat-def* **by** (*auto simp add*: *of-nat-id*)
qed

lemma *arith-series-int*:
(*2*::*int*) * (∑ *i*∈{..<*n*}. *a* + *of-nat i* * *d*) =
of-nat n * (*a* + (*a* + *of-nat*(*n* − *1*)*d))
proof −
have
((*1*::*int*) + *1*) * (∑ *i*∈{..<*n*}. *a* + *of-nat i* * *d*) =
of-nat(*n*) * (*a* + (*a* + *of-nat*(*n* − *1*)*d))
by (*rule arith-series-general*)
thus *?thesis* **by** *simp*
qed

lemma *sum-diff-distrib*:
fixes *P*::*nat*⇒*nat*
shows
∀ *x*. *Q x* ≤ *P x* ⟹
(∑ *x*<*n*. *P x*) − (∑ *x*<*n*. *Q x*) = (∑ *x*<*n*. *P x* − *Q x*)
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*)

let *?lhs* = (∑ *x*<*n*. *P x*) − (∑ *x*<*n*. *Q x*)
let *?rhs* = ∑ *x*<*n*. *P x* − *Q x*

from *Suc* **have** *?lhs* = *?rhs* **by** *simp*
moreover
from *Suc* **have** *?lhs* + *P n* − *Q n* = *?rhs* + (*P n* − *Q n*) **by** *simp*
moreover
from *Suc* **have**
(∑ *x*<*n*. *P x*) + *P n* − ((∑ *x*<*n*. *Q x*) + *Q n*) = *?rhs* + (*P n* − *Q n*)
by (*subst diff-diff-left*[*symmetric*],
subst diff-add-assoc2)
(*auto simp*: *diff-add-assoc2 intro*: *setsum-mono*)
ultimately
show *?case* **by** *simp*
qed

34.11 Products indexed over intervals

syntax
-from-to-setprod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*PROD - = -..-./ -*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setprod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*PROD - = -..<-./ -*) [*0*,*0*,*0*,*10*] *10*)
-upt-setprod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*PROD -<-./ -*) [*0*,*0*,*10*] *10*)

-upto-setprod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*PROD -<=-./ -*) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-from-to-setprod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∏ *- = -..-./ -*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setprod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∏ *- = -..<-./ -*) [*0*,*0*,*0*,*10*] *10*)
-upt-setprod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∏ *-<-./ -*) [*0*,*0*,*10*] *10*)
-upto-setprod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∏ *-≤-./ -*) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-from-to-setprod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∏ *- = -..-./ -*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setprod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∏ *- = -..<-./ -*) [*0*,*0*,*0*,*10*] *10*)
-upt-setprod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∏ *-<-./ -*) [*0*,*0*,*10*] *10*)
-upto-setprod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* ((*3*∏ *-≤-./ -*) [*0*,*0*,*10*] *10*)
syntax (*latex-prod* **output**)
-from-to-setprod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∏$_{-=\_}$ *-*) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setprod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∏$_{-=\_}^{<-}$ *-*) [*0*,*0*,*0*,*10*] *10*)
-upt-setprod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∏$_{-<\_}$ *-*) [*0*,*0*,*10*] *10*)
-upto-setprod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
((*3*∏$_{-\leq\_}$ *-*) [*0*,*0*,*10*] *10*)

translations
∏ *x*=*a*..*b*. *t* == *CONST setprod* (%*x*. *t*) {*a*..*b*}
∏ *x*=*a*..<*b*. *t* == *CONST setprod* (%*x*. *t*) {*a*..<*b*}
∏ *i*≤*n*. *t* == *CONST setprod* (λ*i*. *t*) {..*n*}
∏ *i*<*n*. *t* == *CONST setprod* (λ*i*. *t*) {..<*n*}

end

35 Presburger: Decision Procedure for Presburger Arithmetic

theory *Presburger*
imports *Groebner-Basis SetInterval*
uses
Tools/Qelim/qelim.ML
Tools/Qelim/cooper-data.ML
Tools/Qelim/generated-cooper.ML
(*Tools/Qelim/cooper.ML*)
(*Tools/Qelim/presburger.ML*)
begin

setup *CooperData.setup*

35.1 The $-\infty$ and $+\infty$ Properties

lemma *minf*:
$[\![\exists (z ::'a::linorder).\forall x<z.\ P\ x = P'\ x;\ \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z ::'a::linorder).\forall x<z.\ P\ x = P'\ x;\ \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x = t) = False$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \neq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x < t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \leq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x > t) = False$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \geq t) = False$
$\exists z.\forall (x::'a::\{linorder,plus,Ring\text{-}and\text{-}Field.dvd\})<z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'a::\{linorder,plus,Ring\text{-}and\text{-}Field.dvd\})<z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x<z.\ F = F$
by ((*erule exE*, *erule exE*,*rule-tac x=min z za* **in** *exI*,*simp*)+, (*rule-tac x=t* **in** *exI*,*fastsimp*)+) *simp-all*

lemma *pinf*:
$[\![\exists (z ::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z ::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x = t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \neq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x < t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \leq t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x > t) = True$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \geq t) = True$
$\exists z.\forall (x::'a::\{linorder,plus,Ring\text{-}and\text{-}Field.dvd\})>z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'a::\{linorder,plus,Ring\text{-}and\text{-}Field.dvd\})>z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x>z.\ F = F$
by ((*erule exE*, *erule exE*,*rule-tac x=max z za* **in** *exI*,*simp*)+,(*rule-tac x=t* **in** *exI*,*fastsimp*)+) *simp-all*

lemma *inf-period*:
$[\![\forall x\ k.\ P\ x = P\ (x - k*D);\ \forall x\ k.\ Q\ x = Q\ (x - k*D)]\!]$
$\Longrightarrow \forall x\ k.\ (P\ x \wedge Q\ x) = (P\ (x - k*D) \wedge Q\ (x - k*D))$
$[\![\forall x\ k.\ P\ x = P\ (x - k*D);\ \forall x\ k.\ Q\ x = Q\ (x - k*D)]\!]$
$\Longrightarrow \forall x\ k.\ (P\ x \vee Q\ x) = (P\ (x - k*D) \vee Q\ (x - k*D))$
$(d::'a::\{comm\text{-}ring,Ring\text{-}and\text{-}Field.dvd\})\ dvd\ D \Longrightarrow \forall x\ k.\ (d\ dvd\ x + t) = (d\ dvd\ (x - k*D) + t)$
$(d::'a::\{comm\text{-}ring,Ring\text{-}and\text{-}Field.dvd\})\ dvd\ D \Longrightarrow \forall x\ k.\ (\neg d\ dvd\ x + t) = (\neg d\ dvd\ (x - k*D) + t)$
$\forall x\ k.\ F = F$

apply (*auto elim*!: *dvdE simp add*: *algebra-simps*)
unfolding *mult-assoc* [*symmetric*] *left-distrib* [*symmetric*] *left-diff-distrib* [*symmetric*]
unfolding *dvd-def mult-commute* [*of d*]
by *auto*

35.2 The A and B sets

lemma *bset*:
$[\![\forall x.(\forall j \in \{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)$;
$\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x - D) \wedge Q\ (x - D))$
$[\![\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)$;
$\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \vee Q\ x) \longrightarrow (P(x - D) \vee Q\ (x - D))$
$[\![D{>}0; t - 1{\in} B]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t))$
$[\![D{>}0\ ;\ t - 1 \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x - D) + t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (\neg d\ dvd\ x{+}t) \longrightarrow (\neg\ d\ dvd\ (x - D) + t))$
$\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow F \longrightarrow F$
proof (*blast*, *blast*)
assume *dp*: $D > 0$ **and** *tB*: $t - 1{\in} B$
show $(\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** *x*=*1*],*erule ballE*[**where** *x*=*t* − *1*])
apply *algebra* **using** *dp tB* **by** *simp-all*
next
assume *dp*: $D > 0$ **and** *tB*: $t \in B$
show $(\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** *x*=*D*],*erule ballE*[**where** *x*=*t*])
apply *algebra*
using *dp tB* **by** *simp-all*
next
assume *dp*: $D > 0$ **thus** $(\forall x.(\forall j{\in}\{1 .. D\}. \forall b{\in}B.\ x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$ **by** *arith*
next

assume *dp*: $D > 0$ **thus** $\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (x \leq t) \longrightarrow (x - D \leq t)$ **by** *arith*
next
assume *dp*: $D > 0$ **and** *tB*:$t \in B$
{fix x **assume** *nob*: $\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j$ **and** *g*: $x > t$ **and** *ng*: $\neg\ (x - D) > t$
hence $x - t \leq D$ **and** $1 \leq x - t$ **by** *simp+*
hence $\exists j \in \{1 \,..\, D\}.\ x - t = j$ **by** *auto*
hence $\exists j \in \{1 \,..\, D\}.\ x = t + j$ **by** (*simp add*: *algebra-simps*)
with *nob tB* **have** *False* **by** *simp***}**
thus $\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (x > t) \longrightarrow (x - D > t)$ **by** *blast*
next
assume *dp*: $D > 0$ **and** *tB*:$t - 1{\in}\ B$
{fix x **assume** *nob*: $\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j$ **and** *g*: $x \geq t$ **and** *ng*: $\neg\ (x - D) \geq t$
hence $x - (t - 1) \leq D$ **and** $1 \leq x - (t - 1)$ **by** *simp+*
hence $\exists j \in \{1 \,..\, D\}.\ x - (t - 1) = j$ **by** *auto*
hence $\exists j \in \{1 \,..\, D\}.\ x = (t - 1) + j$ **by** (*simp add*: *algebra-simps*)
with *nob tB* **have** *False* **by** *simp***}**
thus $\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (x \geq t) \longrightarrow (x - D \geq t)$ **by** *blast*
next
assume *d*: *d dvd D*
{fix x **assume** *H*: *d dvd* $x + t$ **with** *d* **have** *d dvd* $(x - D) + t$ **by** *algebra***}**
thus $\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x - D) + t)$ **by** *simp*
next
assume *d*: *d dvd D*
{fix x **assume** *H*: $\neg(d\ dvd\ x + t)$ **with** *d* **have** $\neg\ d\ dvd\ (x - D) + t$
by (*clarsimp simp add*: *dvd-def*,*erule-tac* $x{=}\ ka + k$ **in** *allE*,*simp add*: *algebra-simps*)**}**
thus $\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (\neg d\ dvd\ x{+}t) \longrightarrow (\neg d\ dvd\ (x - D) + t)$ **by** *auto*
qed *blast*

lemma *aset*:
$[\![\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ P\ x \longrightarrow P(x + D)\ ;$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x + D) \wedge Q\ (x + D))$
$[\![\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ P\ x \longrightarrow P(x + D)\ ;$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (P\ x \vee Q\ x) \longrightarrow (P(x + D) \vee Q\ (x + D))$
$[\![D{>}0;\ t + 1{\in}\ A]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x = t) \longrightarrow (x + D = t))$
$[\![D{>}0\ ;\ t \in A]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \neq t) \longrightarrow (x + D \neq t))$
$[\![D{>}0;\ t{\in}\ A]\!] \Longrightarrow (\forall (x{::}int).\ (\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x < t) \longrightarrow (x + D < t))$

$[\![D{>}0;\ t + 1 \in A]\!] \Longrightarrow (\forall\,(x{::}int).(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \leq t) \longrightarrow (x + D \leq t))$
$D{>}0 \Longrightarrow (\forall\,(x{::}int).(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x > t) \longrightarrow (x + D > t))$
$D{>}0 \Longrightarrow (\forall\,(x{::}int).(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \geq t) \longrightarrow (x + D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall\,(x{::}int).(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x + D) + t))$
$d\ dvd\ D \Longrightarrow (\forall\,(x{::}int).(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (\neg d\ dvd\ x{+}t) \longrightarrow (\neg\ d\ dvd\ (x + D) + t))$
$\forall\, x.(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j) \longrightarrow F \longrightarrow F$

proof (*blast*, *blast*)
assume *dp*: $D > 0$ **and** *tA*: $t + 1 \in A$
show $(\forall\, x.(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x = t) \longrightarrow (x + D = t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** $x{=}1$],*erule ballE*[**where** $x{=}t + 1$])
using *dp tA* **by** *simp-all*
next
assume *dp*: $D > 0$ **and** *tA*: $t \in A$
show $(\forall\, x.(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \neq t) \longrightarrow (x + D \neq t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** $x{=}D$],*erule ballE*[**where** $x{=}t$])
using *dp tA* **by** *simp-all*
next
assume *dp*: $D > 0$ **thus** $(\forall\, x.(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x > t) \longrightarrow (x + D > t))$ **by** *arith*
next
assume *dp*: $D > 0$ **thus** $\forall\, x.(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \geq t) \longrightarrow (x + D \geq t)$ **by** *arith*
next
assume *dp*: $D > 0$ **and** *tA*:$t \in A$
{fix x **assume** *nob*: $\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j$ **and** *g*: $x < t$ **and** *ng*: $\neg\ (x + D) < t$
hence $t - x \leq D$ **and** $1 \leq t - x$ **by** *simp+*
hence $\exists j \in \{1\ ..\ D\}.\ t - x = j$ **by** *auto*
hence $\exists j \in \{1\ ..\ D\}.\ x = t - j$ **by** (*auto simp add*: *algebra-simps*)
with *nob tA* **have** *False* **by** *simp***}**
thus $\forall\, x.(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x < t) \longrightarrow (x + D < t)$ **by** *blast*
next
assume *dp*: $D > 0$ **and** *tA*:$t + 1 \in A$
{fix x **assume** *nob*: $\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j$ **and** *g*: $x \leq t$ **and** *ng*: $\neg\ (x + D) \leq t$
hence $(t + 1) - x \leq D$ **and** $1 \leq (t + 1) - x$ **by** (*simp-all add*: *algebra-simps*)
hence $\exists j \in \{1\ ..\ D\}.\ (t + 1) - x = j$ **by** *auto*
hence $\exists j \in \{1\ ..\ D\}.\ x = (t + 1) - j$ **by** (*auto simp add*: *algebra-simps*)
with *nob tA* **have** *False* **by** *simp***}**
thus $\forall\, x.(\forall\, j{\in}\{1\ ..\ D\}.\ \forall\, b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \leq t) \longrightarrow (x + D \leq t)$ **by** *blast*
next
assume *d*: *d dvd D*
{fix x **assume** *H*: *d dvd* $x + t$ **with** *d* **have** *d dvd* $(x + D) + t$

by (*clarsimp simp add*: *dvd-def*,*rule-tac x*= *ka* + *k* **in** *exI*,*simp add*: *algebra-simps*)**}**
thus ∀(*x*::*int*).(∀*j*∈{*1* .. *D*}. ∀*b*∈*A*. *x* ≠ *b* − *j*)⟶ (*d dvd x*+*t*) ⟶ (*d dvd* (*x* + *D*) + *t*) **by** *simp*
next
assume *d*: *d dvd D*
{fix *x* **assume** *H*: ¬(*d dvd x* + *t*) **with** *d* **have** ¬*d dvd* (*x* + *D*) + *t*
by (*clarsimp simp add*: *dvd-def*,*erule-tac x*= *ka* − *k* **in** *allE*,*simp add*: *algebra-simps*)**}**
thus ∀(*x*::*int*).(∀*j*∈{*1* .. *D*}. ∀*b*∈*A*. *x* ≠ *b* − *j*)⟶ (¬*d dvd x*+*t*) ⟶ (¬*d dvd* (*x* + *D*) + *t*) **by** *auto*
qed *blast*

35.3 Cooper's Theorem −∞ and +∞ Version

35.3.1 First some trivial facts about periodic sets or predicates

lemma *periodic-finite-ex*:
assumes *dpos*: (*0*::*int*) < *d* **and** *modd*: *ALL x k*. *P x* = *P*(*x* − *k*∗*d*)
shows (*EX x*. *P x*) = (*EX j* : {*1*..*d*}. *P j*)
(**is** *?LHS* = *?RHS*)
proof
assume *?LHS*
then obtain *x* **where** *P*: *P x* **..**
have *x mod d* = *x* − (*x div d*)∗*d* **by**(*simp add*:*zmod-zdiv-equality mult-ac eq-diff-eq*)
hence *Pmod*: *P x* = *P*(*x mod d*) **using** *modd* **by** *simp*
show *?RHS*
proof (*cases*)
assume *x mod d* = *0*
hence *P 0* **using** *P Pmod* **by** *simp*
moreover have *P 0* = *P*(*0* − (−*1*)∗*d*) **using** *modd* **by** *blast*
ultimately have *P d* **by** *simp*
moreover have *d* : {*1*..*d*} **using** *dpos* **by**(*simp add*:*atLeastAtMost-iff*)
ultimately show *?RHS* **..**
next
assume *not0*: *x mod d* ≠ *0*
have *P*(*x mod d*) **using** *dpos P Pmod* **by**(*simp add*:*pos-mod-sign pos-mod-bound*)
moreover have *x mod d* : {*1*..*d*}
proof −
from *dpos* **have** *0* ≤ *x mod d* **by**(*rule pos-mod-sign*)
moreover from *dpos* **have** *x mod d* < *d* **by**(*rule pos-mod-bound*)
ultimately show *?thesis* **using** *not0* **by**(*simp add*:*atLeastAtMost-iff*)
qed
ultimately show *?RHS* **..**
qed
qed *auto*

35.3.2 The −∞ Version

lemma *decr-lemma*: *0* < (*d*::*int*) ⟹ *x* − (*abs*(*x*−*z*)+*1*) ∗ *d* < *z*

by(*induct rule*: *int-gr-induct*,*simp-all add*:*int-distrib*)

lemma *incr-lemma*: $0 < (d::int) \Longrightarrow z < x + (abs(x-z)+1) * d$
by(*induct rule*: *int-gr-induct*, *simp-all add*:*int-distrib*)

theorem *int-induct*[*case-names base step1 step2*]:
assumes
base: $P(k::int)$ **and** *step1*: $\bigwedge i.\ [\![k \leq i;\ P\ i]\!] \Longrightarrow P(i+1)$ **and**
step2: $\bigwedge i.\ [\![k \geq i;\ P\ i]\!] \Longrightarrow P(i-1)$
shows $P\ i$
proof −
have $i \leq k \vee i \geq k$ **by** *arith*
thus *?thesis* **using** *prems int-ge-induct*[**where** $P=P$ **and** $k=k$ **and** $i=i$] *int-le-induct*[**where** $P=P$ **and** $k=k$ **and** $i=i$] **by** *blast*
qed

lemma *decr-mult-lemma*:
assumes *dpos*: $(0::int) < d$ **and** *minus*: $\forall x.\ P\ x \longrightarrow P(x-d)$ **and** *knneg*: $0 <= k$
shows $ALL\ x.\ P\ x \longrightarrow P(x - k*d)$
using *knneg*
proof (*induct rule*:*int-ge-induct*)
case *base* **thus** *?case* **by** *simp*
next
case (*step i*)
{fix x
have $P\ x \longrightarrow P\ (x - i * d)$ **using** *step.hyps* **by** *blast*
also have $\ldots \longrightarrow P(x - (i + 1) * d)$ **using** *minus*[*THEN spec*, *of* $x - i * d$]
by (*simp add*:*int-distrib OrderedGroup.diff-diff-eq*[*symmetric*])
ultimately have $P\ x \longrightarrow P(x - (i + 1) * d)$ **by** *blast***}**
thus *?case* **..**
qed

lemma *minusinfinity*:
assumes *dpos*: $0 < d$ **and**
P1eqP1: $ALL\ x\ k.\ P1\ x = P1(x - k*d)$ **and** *ePeqP1*: $EX\ z::int.\ ALL\ x.\ x < z \longrightarrow (P\ x = P1\ x)$
shows $(EX\ x.\ P1\ x) \longrightarrow (EX\ x.\ P\ x)$
proof
assume *eP1*: $EX\ x.\ P1\ x$
then obtain x **where** *P1*: $P1\ x$ **..**
from *ePeqP1* **obtain** z **where** *P1eqP*: $ALL\ x.\ x < z \longrightarrow (P\ x = P1\ x)$ **..**
let $?w = x - (abs(x-z)+1) * d$
from *dpos* **have** *w*: $?w < z$ **by**(*rule decr-lemma*)
have $P1\ x = P1\ ?w$ **using** *P1eqP1* **by** *blast*
also have $\ldots = P(?w)$ **using** *w P1eqP* **by** *blast*
finally have $P\ ?w$ **using** *P1* **by** *blast*
thus $EX\ x.\ P\ x$ **..**
qed

lemma *cpmi*:
assumes *dp*: $0 < D$ **and** *p1*:$\exists\, z.\ \forall\ x{<}\ z.\ P\ x = P'\ x$
and *nb*:$\forall\, x.(\forall\ j{\in}\ \{1..D\}.\ \forall\, (b{::}int) \in B.\ x \neq b{+}j) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$
and *pd*: $\forall\ x\ k.\ P'\ x = P'\ (x{-}k{*}D)$
shows $(\exists\, x.\ P\ x) = ((\exists\ j{\in}\ \{1..D\}\ .\ P'\ j)\ |\ (\exists\ j\ \in \{1..D\}.\exists\ b{\in}\ B.\ P\ (b{+}j)))$
(**is** *?L* $= (?R1 \vee ?R2)$)
proof−
{assume *?R2* **hence** *?L* **by** *blast***}**
moreover
{assume *H*:*?R1* **hence** *?L* **using** *minusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] **by** *simp***}**
moreover
{ fix *x*
assume *P*: *P x* **and** *H*: ¬ *?R2*
{fix *y* **assume** ¬ $(\exists j{\in}\{1..D\}.\ \exists\, b{\in}B.\ P\ (b + j))$ **and** *P*: *P y*
hence ~$(EX\ (j{::}int) : \{1..D\}.\ EX\ (b{::}int) : B.\ y = b{+}j)$ **by** *auto*
with *nb P* **have** $P\ (y - D)$ **by** *auto* **}**
hence $ALL\ x.{\sim}(EX\ (j{::}int) : \{1..D\}.\ EX\ (b{::}int) : B.\ P(b{+}j)) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$ **by** *blast*
with *H P* **have** *th*: $\forall\, x.\ P\ x \longrightarrow P\ (x - D)$ **by** *auto*
from *p1* **obtain** *z* **where** *z*: $ALL\ x.\ x < z \longrightarrow (P\ x = P'\ x)$ **by** *blast*
let *?y* $= x - (|x - z| + 1){*}D$
have *zp*: $0 <= (|x - z| + 1)$ **by** *arith*
from *dp* **have** *yz*: *?y* $< z$ **using** *decr-lemma*[*OF dp*] **by** *simp*
from *z*[*rule-format*, *OF yz*] *decr-mult-lemma*[*OF dp th zp*, *rule-format*, *OF P*]
have *th2*: *P' ?y* **by** *auto*
with *periodic-finite-ex*[*OF dp pd*]
have *?R1* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

35.3.3 The $+\infty$ Version

lemma *plusinfinity*:
assumes *dpos*: $(0{::}int) < d$ **and**
P1eqP1: $\forall\, x\ k.\ P'\ x = P'(x - k{*}d)$ **and** *ePeqP1*: $\exists\ z.\ \forall\ x{>}z.\ P\ x = P'\ x$
shows $(\exists\ x.\ P'\ x) \longrightarrow (\exists\ x.\ P\ x)$
proof
assume *eP1*: $EX\ x.\ P'\ x$
then obtain *x* **where** *P1*: *P' x* **..**
from *ePeqP1* **obtain** *z* **where** *P1eqP*: $\forall\, x{>}z.\ P\ x = P'\ x$ **..**
let *?w'* $= x + (abs(x{-}z){+}1) * d$
let *?w* $= x - (-(abs(x{-}z) + 1)){*}d$
have *ww'*[*simp*]: *?w* $=$ *?w'* **by** (*simp add*: *algebra-simps*)
from *dpos* **have** *w*: *?w* $> z$ **by**(*simp only*: *ww' incr-lemma*)
hence *P' x* $=$ *P' ?w* **using** *P1eqP1* **by** *blast*
also have $\ldots = P(?w)$ **using** *w P1eqP* **by** *blast*

finally have *P ?w* **using** *P1* **by** *blast*
thus *EX x. P x* **..**
qed

lemma *incr-mult-lemma*:
assumes *dpos*: *(0::int) < d* **and** *plus*: *ALL x::int. P x ⟶ P(x + d)* **and** *knneg*: *0 <= k*
shows *ALL x. P x ⟶ P(x + k*d)*
using *knneg*
proof (*induct rule:int-ge-induct*)
case *base* **thus** *?case* **by** *simp*
next
case (*step i*)
{fix *x*
have *P x ⟶ P (x + i * d)* **using** *step.hyps* **by** *blast*
also have *... ⟶ P(x + (i + 1) * d)* **using** *plus[THEN spec, of x + i * d]*
by (*simp add:int-distrib zadd-ac*)
ultimately have *P x ⟶ P(x + (i + 1) * d)* **by** *blast***}**
thus *?case* **..**
qed

lemma *cppi*:
assumes *dp*: *0 < D* **and** *p1*:*∃ z. ∀ x> z. P x = P' x*
and *nb*:*∀ x.(∀ j∈ {1..D}. ∀(b::int) ∈ A. x ≠ b − j) −−> P (x) −−> P (x + D)*
and *pd*: *∀ x k. P' x= P' (x−k*D)*
shows *(∃ x. P x) = ((∃ j∈ {1..D} . P' j) | (∃ j ∈ {1..D}.∃ b∈ A. P (b − j)))*
(**is** *?L = (?R1 ∨ ?R2)*)
proof−
{assume *?R2* **hence** *?L* **by** *blast***}**
moreover
{assume *H*:*?R1* **hence** *?L* **using** *plusinfinity[OF dp pd p1] periodic-finite-ex[OF dp pd]* **by** *simp***}**
moreover
{ fix *x*
assume *P*: *P x* **and** *H*: *¬ ?R2*
{fix *y* **assume** *¬ (∃j∈{1..D}. ∃ b∈A. P (b − j))* **and** *P*: *P y*
hence *~(EX (j::int) : {1..D}. EX (b::int) : A. y = b − j)* **by** *auto*
with *nb P* **have** *P (y + D)* **by** *auto* **}**
hence *ALL x.~(EX (j::int) : {1..D}. EX (b::int) : A. P(b−j)) −−> P (x) −−> P (x + D)* **by** *blast*
with *H P* **have** *th*: *∀ x. P x ⟶ P (x + D)* **by** *auto*
from *p1* **obtain** *z* **where** *z*: *ALL x. x > z −−> (P x = P' x)* **by** *blast*
let *?y = x + (|x − z| + 1)*D*
have *zp*: *0 <= (|x − z| + 1)* **by** *arith*
from *dp* **have** *yz*: *?y > z* **using** *incr-lemma[OF dp]* **by** *simp*
from *z[rule-format, OF yz] incr-mult-lemma[OF dp th zp, rule-format, OF P]*
have *th2*: *P' ?y* **by** *auto*
with *periodic-finite-ex[OF dp pd]*

have *?R1* **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

lemma *simp-from-to*: $\{i..j::int\} = (if\ j < i\ then\ \{\}\ else\ insert\ i\ \{i+1..j\})$
apply(*simp add*:*atLeastAtMost-def atLeast-def atMost-def*)
apply(*fastsimp*)
done

theorem *unity-coeff-ex*: $(\exists (x::'a::\{semiring\text{-}0,Ring\text{-}and\text{-}Field.dvd\}).\ P\ (l * x)) \equiv (\exists x.\ l\ dvd\ (x + 0) \wedge P\ x)$
apply (*rule eq-reflection* [*symmetric*])
apply (*rule iffI*)
defer
apply (*erule exE*)
apply (*rule-tac* $x = l * x$ **in** *exI*)
apply (*simp add*: *dvd-def*)
apply (*rule-tac* $x = x$ **in** *exI*, *simp*)
apply (*erule exE*)
apply (*erule conjE*)
apply *simp*
apply (*erule dvdE*)
apply (*rule-tac* $x = k$ **in** *exI*)
apply *simp*
done

lemma *zdvd-mono*: **assumes** *not0*: $(k::int) \neq 0$
shows $((m::int)\ dvd\ t) \equiv (k*m\ dvd\ k*t)$
using *not0* **by** (*simp add*: *dvd-def*)

lemma *uminus-dvd-conv*: $(d\ dvd\ (t::int)) \equiv (-d\ dvd\ t)\ (d\ dvd\ (t::int)) \equiv (d\ dvd\ -t)$
by *simp-all*

Theorems for transforming predicates on nat to predicates on *int*

lemma *all-nat*: $(\forall x::nat.\ P\ x) = (\forall x::int.\ 0 <= x \longrightarrow P\ (nat\ x))$
by (*simp split add*: *split-nat*)

lemma *ex-nat*: $(\exists x::nat.\ P\ x) = (\exists x::int.\ 0 <= x \wedge P\ (nat\ x))$
apply (*auto split add*: *split-nat*)
apply (*rule-tac x=int x* **in** *exI*, *simp*)
apply (*rule-tac* $x = nat\ x$ **in** *exI*,*erule-tac* $x = nat\ x$ **in** *allE*, *simp*)
done

lemma *zdiff-int-split*: $P\ (int\ (x - y)) =$
$((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
by (*case-tac* $y \leq x$, *simp-all add*: *zdiff-int*)

lemma *number-of1*: $(0::int) <= number\text{-}of\ n \Longrightarrow (0::int) <= number\text{-}of\ (Int.Bit0\ n) \wedge (0::int) <= number\text{-}of\ (Int.Bit1\ n)$
by *simp*
lemma *number-of2*: $(0::int) <= Numeral0$ **by** *simp*
lemma *Suc-plus1*: $Suc\ n = n + 1$ **by** *simp*

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*: $(0 <= x \Longrightarrow P = P') \Longrightarrow (0 <= (x::int) \longrightarrow P) = (0 <= x \longrightarrow P')$ **by** *simp*

theorem *conj-le-cong*: $(0 <= x \Longrightarrow P = P') \Longrightarrow (0 <= (x::int) \wedge P) = (0 <= x \wedge P')$
by (*simp cong*: *conj-cong*)
lemma *int-eq-number-of-eq*:
$(((number\text{-}of\ v)::int) = (number\text{-}of\ w)) = iszero\ ((number\text{-}of\ (v + (uminus\ w)))::int)$
by (*rule eq-number-of-eq*)

declare *dvd-eq-mod-eq-0*[*symmetric*, *presburger*]
declare *mod-1*[*presburger*]
declare *mod-0*[*presburger*]
declare *mod-by-1*[*presburger*]
declare *zmod-zero*[*presburger*]
declare *zmod-self*[*presburger*]
declare *mod-self*[*presburger*]
declare *mod-by-0*[*presburger*]
declare *mod-div-trivial*[*presburger*]
declare *div-mod-equality2*[*presburger*]
declare *div-mod-equality*[*presburger*]
declare *mod-div-equality2*[*presburger*]
declare *mod-div-equality*[*presburger*]
declare *mod-mult-self1*[*presburger*]
declare *mod-mult-self2*[*presburger*]
declare *zdiv-zmod-equality2*[*presburger*]
declare *zdiv-zmod-equality*[*presburger*]
declare *mod2-Suc-Suc*[*presburger*]
lemma [*presburger*]: $(a::int)\ div\ 0 = 0$ **and** [*presburger*]: $a\ mod\ 0 = a$
by *simp-all*

use *Tools/Qelim/cooper.ML*
oracle *linzqe-oracle* = *Coopereif.cooper-oracle*

use *Tools/Qelim/presburger.ML*

setup ⟪ *Arith-Data.add-tactic Presburger arithmetic* (*K* (*Presburger.cooper-tac true* [] [])) ⟫

method-setup *presburger* = ⟪
let

```
 fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ()
 fun simple-keyword k = Scan.lift (Args.$$$ k) >> K ()
 val addN = add
 val delN = del
 val elimN = elim
 val any-keyword = keyword addN || keyword delN || simple-keyword elimN
 val thms = Scan.repeat (Scan.unless any-keyword Attrib.multi-thm) >> flat;
in
  Scan.optional (simple-keyword elimN >> K false) true --
  Scan.optional (keyword addN |-- thms) [] --
  Scan.optional (keyword delN |-- thms) [] >>
  (fn ((elim, add-ths), del-ths) => fn ctxt =>
    SIMPLE-METHOD' (Presburger.cooper-tac elim add-ths del-ths ctxt))
end
⟩⟩ Cooper's algorithm for Presburger arithmetic

lemma [presburger, algebra]: m mod 2 = (1::nat) ⟷ ¬ 2 dvd m  by presburger
lemma [presburger, algebra]: m mod 2 = Suc 0 ⟷ ¬ 2 dvd m  by presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = (1::nat) ⟷ ¬ 2 dvd m  by
presburger
lemma [presburger, algebra]: m mod (Suc (Suc 0)) = Suc 0 ⟷ ¬ 2 dvd m  by
presburger
lemma [presburger, algebra]: m mod 2 = (1::int) ⟷ ¬ 2 dvd m  by presburger


lemma zdvd-period:
  fixes a d :: int
  assumes advdd: a dvd d
  shows a dvd (x + t) ⟷ a dvd ((x + c * d) + t)
  using advdd
  apply -
  apply (rule iffI)
  by algebra+

end
```

36 Recdef: TFL: recursive function definitions

```
theory Recdef
imports FunDef Plain
uses
  (Tools/TFL/casesplit.ML)
  (Tools/TFL/utils.ML)
  (Tools/TFL/usyntax.ML)
  (Tools/TFL/dcterm.ML)
  (Tools/TFL/thms.ML)
  (Tools/TFL/rules.ML)
  (Tools/TFL/thry.ML)
```

(*Tools/TFL/tfl.ML*)
(*Tools/TFL/post.ML*)
(*Tools/recdef-package.ML*)
begin

* This form avoids giant explosions in proofs. NOTE USE OF ==

lemma *def-wfrec*: [| f==*wfrec* r H; *wf*(r) |] ==> $f(a) = H$ (*cut* f r a) a
apply *auto*
apply (*blast intro*: *wfrec*)
done

lemma *tfl-wf-induct*: *ALL R*. *wf R* −−>
(*ALL P*. (*ALL x*. (*ALL y*. (*y*,*x*):*R* −−> *P y*) −−> *P x*) −−> (*ALL x*. *P x*))
apply *clarify*
apply (*rule-tac r* = *R* **and** *P* = *P* **and** *a* = *x* **in** *wf-induct*, *assumption*, *blast*)
done

lemma *tfl-cut-apply*: *ALL f R*. (*x*,*a*):*R* −−> (*cut f R a*)(*x*) = *f*(*x*)
apply *clarify*
apply (*rule cut-apply*, *assumption*)
done

lemma *tfl-wfrec*:
ALL M R f. (*f*=*wfrec R M*) −−> *wf R* −−> (*ALL x*. *f x* = *M* (*cut f R x*) *x*)
apply *clarify*
apply (*erule wfrec*)
done

lemma *tfl-eq-True*: (*x* = *True*) −−> *x*
by *blast*

lemma *tfl-rev-eq-mp*: (*x* = *y*) −−> *y* −−> *x*
by *blast*

lemma *tfl-simp-thm*: (*x* −−> *y*) −−> (*x* = *x'*) −−> (*x'* −−> *y*)
by *blast*

lemma *tfl-P-imp-P-iff-True*: *P* ==> *P* = *True*
by *blast*

lemma *tfl-imp-trans*: (*A* −−> *B*) ==> (*B* −−> *C*) ==> (*A* −−> *C*)
by *blast*

lemma *tfl-disj-assoc*: (*a* ∨ *b*) ∨ *c* == *a* ∨ (*b* ∨ *c*)
by *simp*

lemma *tfl-disjE*: *P* ∨ *Q* ==> *P* −−> *R* ==> *Q* −−> *R* ==> *R*

by *blast*

lemma *tfl-exE*: $\exists x.\ P\ x \Longrightarrow \forall x.\ P\ x \longrightarrow Q \Longrightarrow Q$
by *blast*

use *Tools/TFL/casesplit.ML*
use *Tools/TFL/utils.ML*
use *Tools/TFL/usyntax.ML*
use *Tools/TFL/dcterm.ML*
use *Tools/TFL/thms.ML*
use *Tools/TFL/rules.ML*
use *Tools/TFL/thry.ML*
use *Tools/TFL/tfl.ML*
use *Tools/TFL/post.ML*
use *Tools/recdef-package.ML*
setup *RecdefPackage.setup*

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def
less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong let-cong image-cong INT-cong UN-cong bex-cong ball-cong imp-cong

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-pred-nat
wf-same-fst
wf-empty

end

37 Hilbert-Choice: Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*
imports *Nat Wellfounded Plain*
uses (*Tools/meson.ML*) (*Tools/specification-package.ML*)
begin

37.1 Hilbert's epsilon

axiomatization
  Eps :: (*'a* => *bool*) => *'a*
where
  someI: *P x* ==> *P* (*Eps P*)

syntax (*epsilon*)
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3ϵ -./ -*) [*0*, *10*] *10*)
syntax (*HOL*)
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3@ -./ -*) [*0*, *10*] *10*)
syntax
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3SOME -./ -*) [*0*, *10*] *10*)
translations
  SOME x. P == *CONST Eps* (%*x. P*)

print-translation ⟪
(∗ *to avoid eta−contraction of body* ∗)
[(@{*const-syntax Eps*}, *fn* [*Abs abs*] =>
    let val (*x*,*t*) = *atomic-abs-tr' abs*
    in Syntax.const -Eps $ *x* $ *t end*)]
⟫

constdefs
  inv :: (*'a* => *'b*) => (*'b* => *'a*)
  inv(*f* :: *'a* => *'b*) == %*y. SOME x. f x* = *y*

  Inv :: *'a set* => (*'a* => *'b*) => (*'b* => *'a*)
  Inv A f == %*x. SOME y. y* ∈ *A* & *f y* = *x*

37.2 Hilbert's Epsilon-operator

Easier to apply than *someI* if the witness comes from an existential formula

lemma *someI-ex* [*elim?*]: ∃ *x. P x* ==> *P* (*SOME x. P x*)
apply (*erule exE*)
apply (*erule someI*)
done

Easier to apply than *someI* because the conclusion has only one occurrence of *P*.

lemma *someI2*: [| *P a*; !!*x. P x* ==> *Q x* |] ==> *Q* (*SOME x. P x*)
by (*blast intro*: *someI*)

Easier to apply than *someI2* if the witness comes from an existential formula

lemma *someI2-ex*: [| ∃ *a. P a*; !!*x. P x* ==> *Q x* |] ==> *Q* (*SOME x. P x*)
by (*blast intro*: *someI2*)

lemma *some-equality* [*intro*]:
    [| *P a*; !!*x. P x* ==> *x*=*a* |] ==> (*SOME x. P x*) = *a*

by (*blast intro*: *someI2*)

lemma *some1-equality*: [| *EX*!*x*. *P x*; *P a* |] ==> (*SOME x*. *P x*) = *a*
by (*blast intro*: *some-equality*)

lemma *some-eq-ex*: *P* (*SOME x*. *P x*) = (∃ *x*. *P x*)
by (*blast intro*: *someI*)

lemma *some-eq-trivial* [*simp*]: (*SOME y*. *y*=*x*) = *x*
apply (*rule some-equality*)
apply (*rule refl*, *assumption*)
done

lemma *some-sym-eq-trivial* [*simp*]: (*SOME y*. *x*=*y*) = *x*
apply (*rule some-equality*)
apply (*rule refl*)
apply (*erule sym*)
done

37.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

lemma *choice*: ∀ *x*. ∃ *y*. *Q x y* ==> ∃ *f*. ∀ *x*. *Q x* (*f x*)
by (*fast elim*: *someI*)

lemma *bchoice*: ∀ *x*∈*S*. ∃ *y*. *Q x y* ==> ∃ *f*. ∀ *x*∈*S*. *Q x* (*f x*)
by (*fast elim*: *someI*)

37.4 Function Inverse

lemma *inv-id* [*simp*]: *inv id* = *id*
by (*simp add*: *inv-def id-def*)

A one-to-one function has an inverse.

lemma *inv-f-f* [*simp*]: *inj f* ==> *inv f* (*f x*) = *x*
by (*simp add*: *inv-def inj-eq*)

lemma *inv-f-eq*: [| *inj f*; *f x* = *y* |] ==> *inv f y* = *x*
apply (*erule subst*)
apply (*erule inv-f-f*)
done

lemma *inj-imp-inv-eq*: [| *inj f*; ∀ *x*. *f*(*g x*) = *x* |] ==> *inv f* = *g*
by (*blast intro*: *ext inv-f-eq*)

But is it useful?

lemma *inj-transfer*:

assumes *injf*: *inj f* **and** *minor*: !!*y*. *y* ∈ *range*(*f*) ==> *P*(*inv f y*)
shows *P x*
proof −
have *f x* ∈ *range f* **by** *auto*
hence *P*(*inv f* (*f x*)) **by** (*rule minor*)
thus *P x* **by** (*simp add*: *inv-f-f* [*OF injf*])
qed

lemma *inj-iff*: (*inj f*) = (*inv f o f* = *id*)
apply (*simp add*: *o-def expand-fun-eq*)
apply (*blast intro*: *inj-on-inverseI inv-f-f*)
done

lemma *inv-o-cancel*[*simp*]: *inj f* ==> *inv f o f* = *id*
by (*simp add*: *inj-iff*)

lemma *o-inv-o-cancel*[*simp*]: *inj f* ==> *g o inv f o f* = *g*
by (*simp add*: *o-assoc*[*symmetric*])

lemma *inv-image-cancel*[*simp*]:
inj f ==> *inv f* ‘ *f* ‘ *S* = *S*
by (*simp add*: *image-compose*[*symmetric*])

lemma *inj-imp-surj-inv*: *inj f* ==> *surj* (*inv f*)
by (*blast intro*: *surjI inv-f-f*)

lemma *f-inv-f*: *y* ∈ *range*(*f*) ==> *f*(*inv f y*) = *y*
apply (*simp add*: *inv-def*)
apply (*fast intro*: *someI*)
done

lemma *surj-f-inv-f*: *surj f* ==> *f*(*inv f y*) = *y*
by (*simp add*: *f-inv-f surj-range*)

lemma *inv-injective*:
assumes *eq*: *inv f x* = *inv f y*
and *x*: *x*: *range f*
and *y*: *y*: *range f*
shows *x*=*y*
proof −
have *f* (*inv f x*) = *f* (*inv f y*) **using** *eq* **by** *simp*
thus *?thesis* **by** (*simp add*: *f-inv-f x y*)
qed

lemma *inj-on-inv*: *A* <= *range*(*f*) ==> *inj-on* (*inv f*) *A*
by (*fast intro*: *inj-onI elim*: *inv-injective injD*)

lemma *surj-imp-inj-inv*: *surj f* ==> *inj* (*inv f*)

by (*simp add*: *inj-on-inv surj-range*)

lemma *surj-iff*: (*surj f*) = (*f o inv f* = *id*)
apply (*simp add*: *o-def expand-fun-eq*)
apply (*blast intro*: *surjI surj-f-inv-f*)
done

lemma *surj-imp-inv-eq*: [| *surj f*; ∀ *x*. *g*(*f x*) = *x* |] ==> *inv f* = *g*
apply (*rule ext*)
apply (*drule-tac x* = *inv f x* **in** *spec*)
apply (*simp add*: *surj-f-inv-f*)
done

lemma *bij-imp-bij-inv*: *bij f* ==> *bij* (*inv f*)
by (*simp add*: *bij-def inj-imp-surj-inv surj-imp-inj-inv*)

lemma *inv-equality*: [| !!*x*. *g* (*f x*) = *x*; !!*y*. *f* (*g y*) = *y* |] ==> *inv f* = *g*
apply (*rule ext*)
apply (*auto simp add*: *inv-def*)
done

lemma *inv-inv-eq*: *bij f* ==> *inv* (*inv f*) = *f*
apply (*rule inv-equality*)
apply (*auto simp add*: *bij-def surj-f-inv-f*)
done

lemma *o-inv-distrib*: [| *bij f*; *bij g* |] ==> *inv* (*f o g*) = *inv g o inv f*
apply (*rule inv-equality*)
apply (*auto simp add*: *bij-def surj-f-inv-f*)
done

lemma *image-surj-f-inv-f*: *surj f* ==> *f* ‘ (*inv f* ‘ *A*) = *A*
by (*simp add*: *image-eq-UN surj-f-inv-f*)

lemma *image-inv-f-f*: *inj f* ==> (*inv f*) ‘ (*f* ‘ *A*) = *A*
by (*simp add*: *image-eq-UN*)

lemma *inv-image-comp*: *inj f* ==> *inv f* ‘ (*f*‘*X*) = *X*
by (*auto simp add*: *image-def*)

lemma *bij-image-Collect-eq*: *bij f* ==> *f* ‘ *Collect P* = {*y*. *P* (*inv f y*)}
apply *auto*
apply (*force simp add*: *bij-is-inj*)
apply (*blast intro*: *bij-is-surj* [*THEN surj-f-inv-f*, *symmetric*])
done

lemma *bij-vimage-eq-inv-image*: *bij f ==> f −‘ A = inv f ‘ A*
apply (*auto simp add*: *bij-is-surj* [*THEN surj-f-inv-f*])
apply (*blast intro*: *bij-is-inj* [*THEN inv-f-f*, *symmetric*])
done

37.5 Inverse of a PI-function (restricted domain)

lemma *Inv-f-f*: [| *inj-on f A*; *x ∈ A* |] ==> *Inv A f* (*f x*) = *x*
apply (*simp add*: *Inv-def inj-on-def*)
apply (*blast intro*: *someI2*)
done

lemma *f-Inv-f*: *y ∈ f‘A* ==> *f* (*Inv A f y*) = *y*
apply (*simp add*: *Inv-def*)
apply (*fast intro*: *someI2*)
done

lemma *Inv-injective*:
assumes *eq*: *Inv A f x* = *Inv A f y*
and *x*: *x*: *f‘A*
and *y*: *y*: *f‘A*
shows *x*=*y*
proof −
have *f* (*Inv A f x*) = *f* (*Inv A f y*) **using** *eq* **by** *simp*
thus *?thesis* **by** (*simp add*: *f-Inv-f x y*)
qed

lemma *inj-on-Inv*: *B* <= *f‘A* ==> *inj-on* (*Inv A f*) *B*
apply (*rule inj-onI*)
apply (*blast intro*: *inj-onI dest*: *Inv-injective injD*)
done

lemma *Inv-mem*: [| *f ‘ A* = *B*; *x ∈ B* |] ==> *Inv A f x ∈ A*
apply (*simp add*: *Inv-def*)
apply (*fast intro*: *someI2*)
done

lemma *Inv-f-eq*: [| *inj-on f A*; *f x* = *y*; *x ∈ A* |] ==> *Inv A f y* = *x*
apply (*erule subst*)
apply (*erule Inv-f-f*, *assumption*)
done

lemma *Inv-comp*:
[| *inj-on f* (*g ‘ A*); *inj-on g A*; *x ∈ f ‘ g ‘ A* |] ==>
Inv A (*f o g*) *x* = (*Inv A g o Inv* (*g ‘ A*) *f*) *x*
apply *simp*
apply (*rule Inv-f-eq*)
apply (*fast intro*: *comp-inj-on*)
apply (*simp add*: *f-Inv-f Inv-mem*)

apply (*simp add*: *Inv-mem*)
done

lemma *bij-betw-Inv*: *bij-betw f A B* ⟹ *bij-betw* (*Inv A f*) *B A*
apply (*auto simp add*: *bij-betw-def inj-on-Inv Inv-mem*)
apply (*simp add*: *image-compose* [*symmetric*] *o-def*)
apply (*simp add*: *image-def Inv-f-f*)
done

37.6 Other Consequences of Hilbert's Epsilon

Hilbert's Epsilon and the *split* Operator

Looping simprule

lemma *split-paired-Eps*: (*SOME x*. *P x*) = (*SOME* (*a*,*b*). *P*(*a*,*b*))
by *simp*

lemma *Eps-split*: *Eps* (*split P*) = (*SOME xy*. *P* (*fst xy*) (*snd xy*))
by (*simp add*: *split-def*)

lemma *Eps-split-eq* [*simp*]: (@(*x'*,*y'*). *x* = *x'* & *y* = *y'*) = (*x*,*y*)
by *blast*

A relation is wellfounded iff it has no infinite descending chain

lemma *wf-iff-no-infinite-down-chain*:
wf r = (~(∃*f*. ∀*i*. (*f*(*Suc i*),*f i*) ∈ *r*))
apply (*simp only*: *wf-eq-minimal*)
apply (*rule iffI*)
apply (*rule notI*)
apply (*erule exE*)
apply (*erule-tac x* = {*w*. ∃*i*. *w*=*f i*} **in** *allE*, *blast*)
apply (*erule contrapos-np*, *simp*, *clarify*)
apply (*subgoal-tac* ∀*n*. *nat-rec x* (%*i y*. @*z*. *z*:*Q* & (*z*,*y*) :*r*) *n* ∈ *Q*)
apply (*rule-tac x* = *nat-rec x* (%*i y*. @*z*. *z*:*Q* & (*z*,*y*) :*r*) **in** *exI*)
apply (*rule allI*, *simp*)
apply (*rule someI2-ex*, *blast*, *blast*)
apply (*rule allI*)
apply (*induct-tac n*, *simp-all*)
apply (*rule someI2-ex*, *blast*+)
done

lemma *wf-no-infinite-down-chainE*:
assumes *wf r* **obtains** *k* **where** (*f* (*Suc k*), *f k*) ∉ *r*
using ‹*wf r*› *wf-iff-no-infinite-down-chain*[*of r*] **by** *blast*

A dynamically-scoped fact for TFL

lemma *tfl-some*: ∀*P x*. *P x* −−> *P* (*Eps P*)
by (*blast intro*: *someI*)

37.7 Least value operator

constdefs
LeastM :: [*'a* => *'b*::*ord*, *'a* => *bool*] => *'a*
LeastM m P == *SOME x*. *P x* & (∀ *y*. *P y* −−> *m x* <= *m y*)

syntax
-*LeastM* :: [*pttrn*, *'a* => *'b*::*ord*, *bool*] => *'a* (*LEAST* - *WRT* -. - [*0*, *4*, *10*] *10*)
translations
LEAST x WRT m. *P* == *LeastM m* (%*x*. *P*)

lemma *LeastMI2*:
P x ==> (!!*y*. *P y* ==> *m x* <= *m y*)
==> (!!*x*. *P x* ==> ∀ *y*. *P y* −−> *m x* ≤ *m y* ==> *Q x*)
==> *Q* (*LeastM m P*)
apply (*simp add*: *LeastM-def*)
apply (*rule someI2-ex*, *blast*, *blast*)
done

lemma *LeastM-equality*:
P k ==> (!!*x*. *P x* ==> *m k* <= *m x*)
==> *m* (*LEAST x WRT m*. *P x*) = (*m k*::*'a*::*order*)
apply (*rule LeastMI2*, *assumption*, *blast*)
apply (*blast intro*!: *order-antisym*)
done

lemma *wf-linord-ex-has-least*:
wf r ==> ∀ *x y*. ((*x*,*y*):*r*^+) = ((*y*,*x*)~:*r*^*) ==> *P k*
==> ∃ *x*. *P x* & (!*y*. *P y* −−> (*m x*,*m y*):*r*^*)
apply (*drule wf-trancl* [*THEN wf-eq-minimal* [*THEN iffD1*]])
apply (*drule-tac x* = *m*'*Collect P* **in** *spec*, *force*)
done

lemma *ex-has-least-nat*:
P k ==> ∃ *x*. *P x* & (∀ *y*. *P y* −−> *m x* <= (*m y*::*nat*))
apply (*simp only*: *pred-nat-trancl-eq-le* [*symmetric*])
apply (*rule wf-pred-nat* [*THEN wf-linord-ex-has-least*])
apply (*simp add*: *less-eq linorder-not-le pred-nat-trancl-eq-le*, *assumption*)
done

lemma *LeastM-nat-lemma*:
P k ==> *P* (*LeastM m P*) & (∀ *y*. *P y* −−> *m* (*LeastM m P*) <= (*m y*::*nat*))
apply (*simp add*: *LeastM-def*)
apply (*rule someI-ex*)
apply (*erule ex-has-least-nat*)
done

lemmas *LeastM-natI* = *LeastM-nat-lemma* [*THEN conjunct1*, *standard*]

lemma *LeastM-nat-le*: *P x ==> m (LeastM m P) <= (m x::nat)*
by (*rule LeastM-nat-lemma* [*THEN conjunct2*, *THEN spec*, *THEN mp*], *assumption*, *assumption*)

37.8 Greatest value operator

constdefs
GreatestM :: ['a => 'b::ord, 'a => bool] => 'a
GreatestM m P == SOME x. P x & (∀ y. P y --> m y <= m x)

Greatest :: ('a::ord => bool) => 'a (**binder** *GREATEST 10*)
Greatest == GreatestM (%x. x)

syntax
-GreatestM :: [pttrn, 'a=>'b::ord, bool] => 'a
(*GREATEST - WRT -. - [0, 4, 10] 10*)

translations
GREATEST x WRT m. P == GreatestM m (%x. P)

lemma *GreatestMI2*:
P x ==> (!!y. P y ==> m y <= m x)
==> (!!x. P x ==> ∀ y. P y --> m y ≤ m x ==> Q x)
==> Q (GreatestM m P)
apply (*simp add*: *GreatestM-def*)
apply (*rule someI2-ex*, *blast*, *blast*)
done

lemma *GreatestM-equality*:
P k ==> (!!x. P x ==> m x <= m k)
==> m (GREATEST x WRT m. P x) = (m k::'a::order)
apply (*rule-tac m = m* **in** *GreatestMI2*, *assumption*, *blast*)
apply (*blast intro!*: *order-antisym*)
done

lemma *Greatest-equality*:
P (k::'a::order) ==> (!!x. P x ==> x <= k) ==> (GREATEST x. P x) = k
apply (*simp add*: *Greatest-def*)
apply (*erule GreatestM-equality*, *blast*)
done

lemma *ex-has-greatest-nat-lemma*:
P k ==> ∀ x. P x --> (∃ y. P y & ~ ((m y::nat) <= m x))
==> ∃ y. P y & ~ (m y < m k + n)
apply (*induct n*, *force*)
apply (*force simp add*: *le-Suc-eq*)
done

lemma *ex-has-greatest-nat*:

```
  P k ==> ∀ y. P y --> m y < b
    ==> ∃ x. P x & (∀ y. P y --> (m y::nat) <= m x)
  apply (rule ccontr)
  apply (cut-tac P = P and n = b - m k in ex-has-greatest-nat-lemma)
    apply (subgoal-tac [3] m k <= b, auto)
  done

lemma GreatestM-nat-lemma:
  P k ==> ∀ y. P y --> m y < b
    ==> P (GreatestM m P) & (∀ y. P y --> (m y::nat) <= m (GreatestM m
P))
  apply (simp add: GreatestM-def)
  apply (rule someI-ex)
  apply (erule ex-has-greatest-nat, assumption)
  done

lemmas GreatestM-natI = GreatestM-nat-lemma [THEN conjunct1, standard]

lemma GreatestM-nat-le:
  P x ==> ∀ y. P y --> m y < b
    ==> (m x::nat) <= m (GreatestM m P)
  apply (blast dest: GreatestM-nat-lemma [THEN conjunct2, THEN spec, of P])
  done
```

Specialization to *GREATEST*.

```
lemma GreatestI: P (k::nat) ==> ∀ y. P y --> y < b ==> P (GREATEST
x. P x)
  apply (simp add: Greatest-def)
  apply (rule GreatestM-natI, auto)
  done

lemma Greatest-le:
    P x ==> ∀ y. P y --> y < b ==> (x::nat) <= (GREATEST x. P x)
  apply (simp add: Greatest-def)
  apply (rule GreatestM-nat-le, auto)
  done
```

37.9 The Meson proof procedure

37.9.1 Negation Normal Form

de Morgan laws

```
lemma meson-not-conjD: ~(P&Q) ==> ~P | ~Q
  and meson-not-disjD: ~(P|Q) ==> ~P & ~Q
  and meson-not-notD: ~~P ==> P
  and meson-not-allD: !!P. ~(∀ x. P(x)) ==> ∃ x. ~P(x)
  and meson-not-exD: !!P. ~(∃ x. P(x)) ==> ∀ x. ~P(x)
  by fast+
```

Removal of −−> and <−> (positive and negative occurrences)

lemma *meson-imp-to-disjD*: $P-->Q ==> ~P \mid Q$
and *meson-not-impD*: $~(P-->Q) ==> P \& ~Q$
and *meson-iff-to-disjD*: $P=Q ==> (~P \mid Q) \& (~Q \mid P)$
and *meson-not-iffD*: $~(P=Q) ==> (P \mid Q) \& (~P \mid ~Q)$
— Much more efficient than $P \wedge \neg Q \vee Q \wedge \neg P$ for computing CNF
and *meson-not-refl-disj-D*: $x ~= x \mid P ==> P$
by *fast+*

37.9.2 Pulling out the existential quantifiers

Conjunction

lemma *meson-conj-exD1*: $!!P\ Q.\ (\exists x.\ P(x)) \& Q ==> \exists x.\ P(x) \& Q$
and *meson-conj-exD2*: $!!P\ Q.\ P \& (\exists x.\ Q(x)) ==> \exists x.\ P \& Q(x)$
by *fast+*

Disjunction

lemma *meson-disj-exD*: $!!P\ Q.\ (\exists x.\ P(x)) \mid (\exists x.\ Q(x)) ==> \exists x.\ P(x) \mid Q(x)$
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants
and *meson-disj-exD1*: $!!P\ Q.\ (\exists x.\ P(x)) \mid Q ==> \exists x.\ P(x) \mid Q$
and *meson-disj-exD2*: $!!P\ Q.\ P \mid (\exists x.\ Q(x)) ==> \exists x.\ P \mid Q(x)$
by *fast+*

37.9.3 Generating clauses for the Meson Proof Procedure

Disjunctions

lemma *meson-disj-assoc*: $(P|Q)|R ==> P|(Q|R)$
and *meson-disj-comm*: $P|Q ==> Q|P$
and *meson-disj-FalseD1*: $False|P ==> P$
and *meson-disj-FalseD2*: $P|False ==> P$
by *fast+*

37.10 Lemmas for Meson, the Model Elimination Procedure

Generation of contrapositives

Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

lemma *make-neg-rule*: $~P|Q ==> ((~P==>P) ==> Q)$
by *blast*

Version for Plaisted's "Postive refinement" of the Meson procedure

lemma *make-refined-neg-rule*: $~P|Q ==> (P ==> Q)$
by *blast*

P should be a literal

lemma *make-pos-rule*: $P|Q \Longrightarrow ((P \Longrightarrow \sim P) \Longrightarrow Q)$
by *blast*

Versions of *make-neg-rule* and *make-pos-rule* that don't insert new assumptions, for ordinary resolution.

lemmas *make-neg-rule'* = *make-refined-neg-rule*

lemma *make-pos-rule'*: $[|P|Q; \sim P|] \Longrightarrow Q$
by *blast*

Generation of a goal clause – put away the final literal

lemma *make-neg-goal*: $\sim P \Longrightarrow ((\sim P \Longrightarrow P) \Longrightarrow False)$
by *blast*

lemma *make-pos-goal*: $P \Longrightarrow ((P \Longrightarrow \sim P) \Longrightarrow False)$
by *blast*

37.10.1 Lemmas for Forward Proof

There is a similarity to congruence rules

lemma *conj-forward*: $[|\ P' \& Q';\ P' \Longrightarrow P;\ Q' \Longrightarrow Q\ |] \Longrightarrow P \& Q$
by *blast*

lemma *disj-forward*: $[|\ P'|Q';\ P' \Longrightarrow P;\ Q' \Longrightarrow Q\ |] \Longrightarrow P|Q$
by *blast*

lemma *disj-forward2*:
$[|\ P'|Q';\ P' \Longrightarrow P;\ [|\ Q';\ P \Longrightarrow False\ |] \Longrightarrow Q\ |] \Longrightarrow P|Q$
apply *blast*
done

lemma *all-forward*: $[|\ \forall x.\ P'(x);\ !!x.\ P'(x) \Longrightarrow P(x)\ |] \Longrightarrow \forall x.\ P(x)$
by *blast*

lemma *ex-forward*: $[|\ \exists x.\ P'(x);\ !!x.\ P'(x) \Longrightarrow P(x)\ |] \Longrightarrow \exists x.\ P(x)$
by *blast*

Many of these bindings are used by the ATP linkup, and not just by legacy proof scripts.

ML
⟪
val inv-def = thm inv-def;
val Inv-def = thm Inv-def;

val someI = thm someI;

val someI-ex = thm someI-ex;
val someI2 = thm someI2;
val someI2-ex = thm someI2-ex;
val some-equality = thm some-equality;
val some1-equality = thm some1-equality;
val some-eq-ex = thm some-eq-ex;
val some-eq-trivial = thm some-eq-trivial;
val some-sym-eq-trivial = thm some-sym-eq-trivial;
val choice = thm choice;
val bchoice = thm bchoice;
val inv-id = thm inv-id;
val inv-f-f = thm inv-f-f;
val inv-f-eq = thm inv-f-eq;
val inj-imp-inv-eq = thm inj-imp-inv-eq;
val inj-transfer = thm inj-transfer;
val inj-iff = thm inj-iff;
val inj-imp-surj-inv = thm inj-imp-surj-inv;
val f-inv-f = thm f-inv-f;
val surj-f-inv-f = thm surj-f-inv-f;
val inv-injective = thm inv-injective;
val inj-on-inv = thm inj-on-inv;
val surj-imp-inj-inv = thm surj-imp-inj-inv;
val surj-iff = thm surj-iff;
val surj-imp-inv-eq = thm surj-imp-inv-eq;
val bij-imp-bij-inv = thm bij-imp-bij-inv;
val inv-equality = thm inv-equality;
val inv-inv-eq = thm inv-inv-eq;
val o-inv-distrib = thm o-inv-distrib;
val image-surj-f-inv-f = thm image-surj-f-inv-f;
val image-inv-f-f = thm image-inv-f-f;
val inv-image-comp = thm inv-image-comp;
val bij-image-Collect-eq = thm bij-image-Collect-eq;
val bij-vimage-eq-inv-image = thm bij-vimage-eq-inv-image;
val Inv-f-f = thm Inv-f-f;
val f-Inv-f = thm f-Inv-f;
val Inv-injective = thm Inv-injective;
val inj-on-Inv = thm inj-on-Inv;
val split-paired-Eps = thm split-paired-Eps;
val Eps-split = thm Eps-split;
val Eps-split-eq = thm Eps-split-eq;
val wf-iff-no-infinite-down-chain = thm wf-iff-no-infinite-down-chain;
val Inv-mem = thm Inv-mem;
val Inv-f-eq = thm Inv-f-eq;
val Inv-comp = thm Inv-comp;
val tfl-some = thm tfl-some;
val make-neg-rule = thm make-neg-rule;
val make-refined-neg-rule = thm make-refined-neg-rule;
val make-pos-rule = thm make-pos-rule;
val make-neg-rule' = thm make-neg-rule';

val make-pos-rule′ = thm make-pos-rule′;
val make-neg-goal = thm make-neg-goal;
val make-pos-goal = thm make-pos-goal;
val conj-forward = thm conj-forward;
val disj-forward = thm disj-forward;
val disj-forward2 = thm disj-forward2;
val all-forward = thm all-forward;
val ex-forward = thm ex-forward;
⟩⟩

37.11 Meson package

use *Tools/meson.ML*

setup *Meson.setup*

37.12 Specification package – Hilbertized version

lemma *exE-some*: [| *Ex P* ; *c* == *Eps P* |] ==> *P c*
by (*simp only*: *someI-ex*)

use *Tools/specification-package.ML*

end

38 ATP-Linkup: The Isabelle-ATP Linkup

theory *ATP-Linkup*
imports *Divides Record Hilbert-Choice Plain*
uses
Tools/polyhash.ML
Tools/res-clause.ML
(*Tools/res-axioms.ML*)
(*Tools/res-hol-clause.ML*)
(*Tools/res-reconstruct.ML*)
(*Tools/res-atp.ML*)
(*Tools/atp-manager.ML*)
(*Tools/atp-wrapper.ML*)
~~/*src/Tools/Metis/metis.ML*
(*Tools/metis-tools.ML*)
begin

definition *COMBI* :: *'a* => *'a*
where *COMBI P* == *P*

definition *COMBK* :: *'a* => *'b* => *'a*
where *COMBK P Q* == *P*

definition *COMBB* :: ('b => 'c) => ('a => 'b) => 'a => 'c
where *COMBB P Q R* == *P* (*Q R*)

definition *COMBC* :: ('a => 'b => 'c) => 'b => 'a => 'c
where *COMBC P Q R* == *P R Q*

definition *COMBS* :: ('a => 'b => 'c) => ('a => 'b) => 'a => 'c
where *COMBS P Q R* == *P R* (*Q R*)

definition *fequal* :: 'a => 'a => *bool*
where *fequal X Y* == (*X*=*Y*)

lemma *fequal-imp-equal*: *fequal X Y* ==> *X*=*Y*
by (*simp add*: *fequal-def*)

lemma *equal-imp-fequal*: *X*=*Y* ==> *fequal X Y*
by (*simp add*: *fequal-def*)

These two represent the equivalence between Boolean equality and iff. They can't be converted to clauses automatically, as the iff would be expanded...

lemma *iff-positive*: *P* | *Q* | *P*=*Q*
by *blast*

lemma *iff-negative*: ~*P* | ~*Q* | *P*=*Q*
by *blast*

Theorems for translation to combinators

lemma *abs-S*: (%*x*. (*f x*) (*g x*)) == *COMBS f g*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBS-def*)
done

lemma *abs-I*: (%*x*. *x*) == *COMBI*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBI-def*)
done

lemma *abs-K*: (%*x*. *y*) == *COMBK y*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBK-def*)
done

lemma *abs-B*: (%*x*. *a* (*g x*)) == *COMBB a g*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBB-def*)

done

lemma *abs-C*: (%*x*. (*f x*) *b*) == *COMBC f b*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBC-def*)
done

38.1 Setup of external ATPs

use *Tools/res-axioms.ML* **setup** *ResAxioms.setup*
use *Tools/res-hol-clause.ML*
use *Tools/res-reconstruct.ML* **setup** *ResReconstruct.setup*
use *Tools/res-atp.ML*

use *Tools/atp-manager.ML*
use *Tools/atp-wrapper.ML*

basic provers

setup ⟪ *AtpManager.add-prover spass AtpWrapper.spass* ⟫
setup ⟪ *AtpManager.add-prover vampire AtpWrapper.vampire* ⟫
setup ⟪ *AtpManager.add-prover e AtpWrapper.eprover* ⟫

provers with stuctured output

setup ⟪ *AtpManager.add-prover vampire-full AtpWrapper.vampire-full* ⟫
setup ⟪ *AtpManager.add-prover e-full AtpWrapper.eprover-full* ⟫

on some problems better results

setup ⟪ *AtpManager.add-prover spass-no-tc* (*AtpWrapper.spass-opts 40 false*) ⟫

remote provers via SystemOnTPTP

setup ⟪ *AtpManager.add-prover remote-vampire*
(*AtpWrapper.remote-prover −s Vampire−−−9.0*) ⟫
setup ⟪ *AtpManager.add-prover remote-spass*
(*AtpWrapper.remote-prover −s SPASS−−−3.01*) ⟫
setup ⟪ *AtpManager.add-prover remote-e*
(*AtpWrapper.remote-prover −s EP−−−1.0*) ⟫

38.2 The Metis prover

use *Tools/metis-tools.ML*
setup *MetisTools.setup*

end

39 List: The datatype of finite lists

theory *List*

imports *Plain Relation-Power Presburger Recdef ATP-Linkup*
uses *Tools/string-syntax.ML*
begin

datatype *'a list* =
  Nil ([])
| *Cons 'a 'a list* (**infixr** # *65*)

39.1 Basic list processing functions

consts
filter:: (*'a => bool*) => *'a list => 'a list*
concat:: *'a list list => 'a list*
foldl :: (*'b => 'a => 'b*) => *'b => 'a list => 'b*
foldr :: (*'a => 'b => 'b*) => *'a list => 'b => 'b*
hd:: *'a list => 'a*
tl:: *'a list => 'a list*
last:: *'a list => 'a*
butlast :: *'a list => 'a list*
set :: *'a list => 'a set*
map :: (*'a=>'b*) => (*'a list => 'b list*)
listsum :: *'a list => 'a*::*monoid-add*
list-update :: *'a list => nat => 'a => 'a list*
take:: *nat => 'a list => 'a list*
drop:: *nat => 'a list => 'a list*
takeWhile :: (*'a => bool*) => *'a list => 'a list*
dropWhile :: (*'a => bool*) => *'a list => 'a list*
rev :: *'a list => 'a list*
zip :: *'a list => 'b list => ('a * 'b) list*
upt :: *nat => nat => nat list* ((*1*[-..</-]))
remdups :: *'a list => 'a list*
remove1 :: *'a => 'a list => 'a list*
removeAll :: *'a => 'a list => 'a list*
distinct:: *'a list => bool*
replicate :: *nat => 'a => 'a list*
splice :: *'a list ⇒ 'a list ⇒ 'a list*

nonterminals *lupdbinds lupdbind*

syntax
— list Enumeration
@*list* :: *args => 'a list* ([(-)])

— Special syntax for filter
@*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list* ((*1*[-<−-./ -]))

— list update
-*lupdbind*:: [*'a*, *'a*] => *lupdbind* ((*2*- :=/ -))

:: *lupdbind* => *lupdbinds* (-)
-lupdbinds :: [*lupdbind*, *lupdbinds*] => *lupdbinds* (-,/ -)
-LUpdate :: ['*a*, *lupdbinds*] => '*a* (-/[(-)] [*900*,*0*] *900*)

translations
[*x*, *xs*] == *x*#[*xs*]
[*x*] == *x*#[]
[*x*<−*xs* . *P*]== *filter* (%*x*. *P*) *xs*

-LUpdate xs (*-lupdbinds b bs*)== *-LUpdate* (*-LUpdate xs b*) *bs*
xs[*i*:=*x*] == *list-update xs i x*

syntax (*xsymbols*)
@*filter* :: [*pttrn*, '*a list*, *bool*] => '*a list*((*1*[-←- ./ -]))
syntax (*HTML* **output**)
@*filter* :: [*pttrn*, '*a list*, *bool*] => '*a list*((*1*[-←- ./ -]))

Function *size* is overloaded for all datatypes. Users may refer to the list version as *length*.

abbreviation
length :: '*a list* => *nat* **where**
length == *size*

primrec
hd(*x*#*xs*) = *x*

primrec
tl([]) = []
tl(*x*#*xs*) = *xs*

primrec
last(*x*#*xs*) = (*if xs*=[] *then x else last xs*)

primrec
butlast []= []
butlast(*x*#*xs*) = (*if xs*=[] *then* [] *else x*#*butlast xs*)

primrec
set [] = {}
set (*x*#*xs*) = *insert x* (*set xs*)

primrec
map f [] = []
map f (*x*#*xs*) = *f*(*x*)#*map f xs*

primrec
append :: '*a list* ⇒ '*a list* ⇒ '*a list* (**infixr** @ *65*)
where

append-Nil:[] @ *ys* = *ys*
| *append-Cons*: (*x*#*xs*) @ *ys* = *x* # *xs* @ *ys*

primrec
rev([]) = []
rev(*x*#*xs*) = *rev*(*xs*) @ [*x*]

primrec
filter P [] = []
filter P (*x*#*xs*) = (*if P x then x*#*filter P xs else filter P xs*)

primrec
foldl-Nil:*foldl f a* [] = *a*
foldl-Cons: *foldl f a* (*x*#*xs*) = *foldl f* (*f a x*) *xs*

primrec
foldr f [] *a* = *a*
foldr f (*x*#*xs*) *a* = *f x* (*foldr f xs a*)

primrec
concat([]) = []
concat(*x*#*xs*) = *x* @ *concat*(*xs*)

primrec
listsum [] = *0*
listsum (*x* # *xs*) = *x* + *listsum xs*

primrec
drop-Nil:*drop n* [] = []
drop-Cons: *drop n* (*x*#*xs*) = (*case n of 0* => *x*#*xs* | *Suc*(*m*) => *drop m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
take-Nil:*take n* [] = []
take-Cons: *take n* (*x*#*xs*) = (*case n of 0* => [] | *Suc*(*m*) => *x* # *take m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec *nth* :: *'a list* => *nat* => *'a* (**infixl** ! *100*) **where**
nth-Cons: (*x*#*xs*)!*n* = (*case n of 0* => *x* | (*Suc k*) => *xs*!*k*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
[][*i*:=*v*] = []
(*x*#*xs*)[*i*:=*v*] = (*case i of 0* => *v* # *xs* | *Suc j* => *x* # *xs*[*j*:=*v*])

primrec

takeWhile P [] = []
takeWhile P (x#xs) = (if P x then x#takeWhile P xs else [])

primrec
dropWhile P [] = []
dropWhile P (x#xs) = (if P x then dropWhile P xs else x#xs)

primrec
zip xs [] = []
zip-Cons: *zip xs (y#ys) = (case xs of [] => [] | z#zs => (z,y)#zip zs ys)*
— Warning: simpset does not contain this definition, but separate theorems for *xs = []* and *xs = z # zs*

primrec
upt-0: *[i..<0] = []*
upt-Suc: *[i..<(Suc j)] = (if i <= j then [i..<j] @ [j] else [])*

primrec
distinct [] = True
distinct (x#xs) = (x ~: set xs ∧ distinct xs)

primrec
remdups [] = []
remdups (x#xs) = (if x : set xs then remdups xs else x # remdups xs)

primrec
remove1 x [] = []
remove1 x (y#xs) = (if x=y then xs else y # remove1 x xs)

primrec
removeAll x [] = []
removeAll x (y#xs) = (if x=y then removeAll x xs else y # removeAll x xs)

primrec
replicate-0: *replicate 0 x = []*
replicate-Suc: *replicate (Suc n) x = x # replicate n x*

definition
rotate1 :: 'a list ⇒ 'a list **where**
rotate1 xs = (case xs of [] ⇒ [] | x#xs ⇒ xs @ [x])

definition
rotate :: nat ⇒ 'a list ⇒ 'a list **where**
rotate n = rotate1 ^ n

definition
list-all2 :: ('a => 'b => bool) => 'a list => 'b list => bool **where**
[*code del*]: *list-all2 P xs ys =*
(length xs = length ys ∧ (∀(x, y) ∈ set (zip xs ys). P x y))

definition
sublist :: *'a list* => *nat set* => *'a list* **where**
sublist xs A = *map fst* (*filter* (λ*p*. *snd p* ∈ *A*) (*zip xs* [*0*..<*size xs*]))

primrec
splice [] *ys* = *ys*
splice (*x*#*xs*) *ys* = (*if ys*=[] *then x*#*xs else x* # *hd ys* # *splice xs* (*tl ys*))
— Warning: simpset does not contain the second eqn but a derived one.

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort functions are intended for proofs, not for efficient implementations.

context *linorder*
begin

fun *sorted* :: *'a list* ⇒ *bool* **where**
sorted [] ⟷ *True* |
sorted [*x*] ⟷ *True* |
sorted (*x*#*y*#*zs*) ⟷ *x* <= *y* ∧ *sorted* (*y*#*zs*)

primrec *insort* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
insort x [] = [*x*] |
insort x (*y*#*ys*) = (*if x* <= *y then* (*x*#*y*#*ys*) *else y*#(*insort x ys*))

primrec *sort* :: *'a list* ⇒ *'a list* **where**
sort [] = [] |
sort (*x*#*xs*) = *insort x* (*sort xs*)

end

39.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: [(*x*,*y*). *x* ← *xs*, *y* ← *ys*, *x* ≠ *y*], the list of all pairs of distinct elements from *xs* and *ys*. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle's set comprehension): [*e*. *x* ← *xs*, ...] rather than `[e| x <- xs, ...]`.

The qualifiers after the dot are

generators *p* ← *xs*, where *p* is a pattern and *xs* an expression of list type, or

guards *b*, where *b* is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon

$[a, b] @ [c, d] = [a, b, c, d]$
length $[a, b, c] = 3$
set $[a, b, c] = \{a, b, c\}$
map $f\ [a, b, c] = [f\ a, f\ b, f\ c]$
rev $[a, b, c] = [c, b, a]$
hd $[a, b, c, d] = a$
tl $[a, b, c, d] = [b, c, d]$
last $[a, b, c, d] = d$
butlast $[a, b, c, d] = [a, b, c]$
filter $(\lambda n{::}nat.\ n{<}2)\ [0,2,1] = [0,1]$
concat $[[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]$
foldl $f\ x\ [a, b, c] = f\ (f\ (f\ x\ a)\ b)\ c$
foldr $f\ [a, b, c]\ x = f\ a\ (f\ b\ (f\ c\ x))$
zip $[a, b, c]\ [x, y, z] = [(a, x), (b, y), (c, z)]$
zip $[a, b]\ [x, y, z] = [(a, x), (b, y)]$
splice $[a, b, c]\ [x, y, z] = [a, x, b, y, c, z]$
splice $[a, b, c, d]\ [x, y] = [a, x, b, y, c, d]$
take $2\ [a, b, c, d] = [a, b]$
take $6\ [a, b, c, d] = [a, b, c, d]$
drop $2\ [a, b, c, d] = [c, d]$
drop $6\ [a, b, c, d] = []$
takeWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [1, 2]$
dropWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [3, 0]$
distinct $[2, 0, 1]$
remdups $[2, 0, 2, 1, 2] = [0, 1, 2]$
remove1 $2\ [2, 0, 2, 1, 2] = [0, 2, 1, 2]$
removeAll $2\ [2, 0, 2, 1, 2] = [0, 1]$
$[a, b, c, d]\ !\ 2 = c$
$[a, b, c, d][2 := x] = [a, b, x, d]$
sublist $[a, b, c, d, e]\ \{0, 2, 3\} = [a, c, d]$
rotate1 $[a, b, c, d] = [b, c, d, a]$
rotate $3\ [a, b, c, d] = [d, a, b, c]$
replicate $4\ a = [a, a, a, a]$
$[2..{<}5] = [2, 3, 4]$
listsum $[1, 2, 3] = 6$

Figure 1: Characteristic examples

output. Note that the translation of $[e.\ x \leftarrow xs]$ is optmized to $map\ (\lambda x.\ e)\ xs$.

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

nonterminals *lc-qual lc-quals*

syntax
-listcompr :: *'a* ⇒ *lc-qual* ⇒ *lc-quals* ⇒ *'a list* ([- . --)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- <− -)
-lc-test :: *bool* ⇒ *lc-qual* (-)

-lc-end :: *lc-quals* (])
-lc-quals :: *lc-qual* ⇒ *lc-quals* ⇒ *lc-quals* (, --)
-lc-abs :: *'a* => *'b list* => *'b list*

syntax (*xsymbols*)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- ← -)
syntax (*HTML* **output**)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- ← -)

parse-translation (**advanced**) ⟪
```
let
  val NilC = Syntax.const @{const-name Nil};
  val ConsC = Syntax.const @{const-name Cons};
  val mapC = Syntax.const @{const-name map};
  val concatC = Syntax.const @{const-name concat};
  val IfC = Syntax.const @{const-name If};
  fun singl x = ConsC $ x $ NilC;

   fun pat-tr ctxt p e opti = (* %x. case x of p => e | - => [] *)
    let
     val x = Free (Name.variant (fold Term.add-free-names [p, e] []) x, dummyT);
      val e = if opti then singl e else e;
      val case1 = Syntax.const -case1 $ p $ e;
      val case2 = Syntax.const -case1 $ Syntax.const Term.dummy-patternN
                                    $ NilC;
      val cs = Syntax.const -case2 $ case1 $ case2
      val ft = DatatypeCase.case-tr false DatatypePackage.datatype-of-constr
                ctxt [x, cs]
    in lambda x ft end;

  fun abs-tr ctxt (p as Free(s,T)) e opti =
        let val thy = ProofContext.theory-of ctxt;
            val s' = Sign.intern-const thy s
```

```
      in if Sign.declared-const thy s'
        then (pat-tr ctxt p e opti, false)
        else (lambda p e, true)
      end
  | abs-tr ctxt p e opti = (pat-tr ctxt p e opti, false);

fun lc-tr ctxt [e, Const(-lc-test,-)$b, qs] =
      let val res = case qs of Const(-lc-end,-) => singl e
                  | Const(-lc-quals,-)$q$qs => lc-tr ctxt [e,q,qs];
      in IfC $ b $ res $ NilC end
  | lc-tr ctxt [e, Const(-lc-gen,-) $ p $ es, Const(-lc-end,-)] =
      (case abs-tr ctxt p e true of
        (f,true) => mapC $ f $ es
       | (f, false) => concatC $ (mapC $ f $ es))
  | lc-tr ctxt [e, Const(-lc-gen,-) $ p $ es, Const(-lc-quals,-)$q$qs] =
      let val e' = lc-tr ctxt [e,q,qs];
      in concatC $ (mapC $ (fst(abs-tr ctxt p e' false)) $ es) end

in [(-listcompr, lc-tr)] end
⟩⟩
```

39.1.2 [] and *op #*

lemma *not-Cons-self* [*simp*]:
$xs \neq x \# xs$
by (*induct xs*) *auto*

lemmas *not-Cons-self2* [*simp*] = *not-Cons-self* [*symmetric*]

lemma *neq-Nil-conv*: $(xs \neq []) = (\exists y\ ys.\ xs = y \# ys)$
by (*induct xs*) *auto*

lemma *length-induct*:
$(\bigwedge xs.\ \forall ys.\ length\ ys < length\ xs \longrightarrow P\ ys \Longrightarrow P\ xs) \Longrightarrow P\ xs$
by (*rule measure-induct* [*of length*]) *iprover*

39.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: *length* (*xs* @ *ys*) = *length xs* + *length ys*
by (*induct xs*) *auto*

lemma *length-map* [*simp*]: *length* (*map f xs*) = *length xs*
by (*induct xs*) *auto*

lemma *length-rev* [*simp*]: *length* (*rev xs*) = *length xs*
by (*induct xs*) *auto*

lemma *length-tl* [*simp*]: *length* (*tl xs*) = *length xs* − *1*

by (*cases xs*) *auto*

lemma *length-0-conv* [*iff*]: (*length xs* = *0*) = (*xs* = []) **by** (*induct xs*) *auto*

lemma *length-greater-0-conv* [*iff*]: (*0* < *length xs*) = (*xs* ≠ []) **by** (*induct xs*) *auto*

lemma *length-pos-if-in-set*: *x* : *set xs* ⟹ *length xs* > *0* **by** *auto*

lemma *length-Suc-conv*:
(*length xs* = *Suc n*) = (∃ *y ys*. *xs* = *y* # *ys* ∧ *length ys* = *n*) **by** (*induct xs*) *auto*

lemma *Suc-length-conv*:
(*Suc n* = *length xs*) = (∃ *y ys*. *xs* = *y* # *ys* ∧ *length ys* = *n*)
apply (*induct xs*, *simp*, *simp*)
apply *blast*
done

lemma *impossible-Cons*: *length xs* <= *length ys* ==> *xs* = *x* # *ys* = *False*
by (*induct xs*) *auto*

lemma *list-induct2* [*consumes 1*, *case-names Nil Cons*]:
length xs = *length ys* ⟹ *P* [] [] ⟹
(⋀*x xs y ys*. *length xs* = *length ys* ⟹ *P xs ys* ⟹ *P* (*x*#*xs*) (*y*#*ys*))
⟹ *P xs ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs ys*) **then show** *?case* **by** (*cases ys*) *simp-all*
qed

lemma *list-induct3* [*consumes 2*, *case-names Nil Cons*]:
length xs = *length ys* ⟹ *length ys* = *length zs* ⟹ *P* [] [] [] ⟹
(⋀*x xs y ys z zs*. *length xs* = *length ys* ⟹ *length ys* = *length zs* ⟹ *P xs ys zs*
⟹ *P* (*x*#*xs*) (*y*#*ys*) (*z*#*zs*))
⟹ *P xs ys zs*
proof (*induct xs arbitrary*: *ys zs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs ys zs*) **then show** *?case* **by** (*cases ys*, *simp-all*)
(*cases zs*, *simp-all*)
qed

lemma *list-induct2′*:
⟦ *P* [] [];
⋀*x xs*. *P* (*x*#*xs*) [];

```
  ⋀y ys. P [] (y#ys);
   ⋀x xs y ys. P xs ys  ⟹ P (x#xs) (y#ys) ⟧
 ⟹ P xs ys
by (induct xs arbitrary: ys) (case-tac x, auto)+

lemma neq-if-length-neq: length xs ≠ length ys ⟹ (xs = ys) == False
by (rule Eq-FalseI) auto

simproc-setup list-neq ((xs::'a list) = ys) = ⟪
(*
Reduces xs=ys to False if xs and ys cannot be of the same length.
This is the case if the atomic sublists of one are a submultiset
of those of the other list and there are fewer Cons's in one than the other.
*)

let

fun len (Const(@{const-name Nil},-)) acc = acc
  | len (Const(@{const-name Cons},-) $ - $ xs) (ts,n) = len xs (ts,n+1)
  | len (Const(@{const-name append},-) $ xs $ ys) acc = len xs (len ys acc)
  | len (Const(@{const-name rev},-) $ xs) acc = len xs acc
  | len (Const(@{const-name map},-) $ - $ xs) acc = len xs acc
  | len t (ts,n) = (t::ts,n);

fun list-neq - ss ct =
  let
    val (Const(-,eqT) $ lhs $ rhs) = Thm.term-of ct;
    val (ls,m) = len lhs ([],0) and (rs,n) = len rhs ([],0);
    fun prove-neq() =
      let
        val Type(-,listT::-) = eqT;
        val size = HOLogic.size-const listT;
        val eq-len = HOLogic.mk-eq (size $ lhs, size $ rhs);
        val neq-len = HOLogic.mk-Trueprop (HOLogic.Not $ eq-len);
        val thm = Goal.prove (Simplifier.the-context ss) [] [] neq-len
          (K (simp-tac (Simplifier.inherit-context ss @{simpset}) 1));
      in SOME (thm RS @{thm neq-if-length-neq}) end
  in
    if m < n andalso submultiset (op aconv) (ls,rs) orelse
       n < m andalso submultiset (op aconv) (rs,ls)
    then prove-neq() else NONE
  end;
in list-neq end;
⟫
```

39.1.4 @ − append

```
lemma append-assoc [simp]: (xs @ ys) @ zs = xs @ (ys @ zs)
by (induct xs) auto
```

lemma *append-Nil2* [*simp*]: *xs* @ [] = *xs*
by (*induct xs*) *auto*

lemma *append-is-Nil-conv* [*iff*]: (*xs* @ *ys* = []) = (*xs* = [] ∧ *ys* = [])
by (*induct xs*) *auto*

lemma *Nil-is-append-conv* [*iff*]: ([] = *xs* @ *ys*) = (*xs* = [] ∧ *ys* = [])
by (*induct xs*) *auto*

lemma *append-self-conv* [*iff*]: (*xs* @ *ys* = *xs*) = (*ys* = [])
by (*induct xs*) *auto*

lemma *self-append-conv* [*iff*]: (*xs* = *xs* @ *ys*) = (*ys* = [])
by (*induct xs*) *auto*

lemma *append-eq-append-conv* [*simp*, *noatp*]:
length xs = *length ys* ∨ *length us* = *length vs*
==> (*xs*@*us* = *ys*@*vs*) = (*xs*=*ys* ∧ *us*=*vs*)
apply (*induct xs arbitrary*: *ys*)
apply (*case-tac ys*, *simp*, *force*)
apply (*case-tac ys*, *force*, *simp*)
done

lemma *append-eq-append-conv2*: (*xs* @ *ys* = *zs* @ *ts*) =
(*EX us*. *xs* = *zs* @ *us* & *us* @ *ys* = *ts* | *xs* @ *us* = *zs* & *ys* = *us*@ *ts*)
apply (*induct xs arbitrary*: *ys zs ts*)
apply *fastsimp*
apply(*case-tac zs*)
apply *simp*
apply *fastsimp*
done

lemma *same-append-eq* [*iff*]: (*xs* @ *ys* = *xs* @ *zs*) = (*ys* = *zs*)
by *simp*

lemma *append1-eq-conv* [*iff*]: (*xs* @ [*x*] = *ys* @ [*y*]) = (*xs* = *ys* ∧ *x* = *y*)
by *simp*

lemma *append-same-eq* [*iff*]: (*ys* @ *xs* = *zs* @ *xs*) = (*ys* = *zs*)
by *simp*

lemma *append-self-conv2* [*iff*]: (*xs* @ *ys* = *ys*) = (*xs* = [])
using *append-same-eq* [*of* - - []] **by** *auto*

lemma *self-append-conv2* [*iff*]: (*ys* = *xs* @ *ys*) = (*xs* = [])
using *append-same-eq* [*of* []] **by** *auto*

lemma *hd-Cons-tl* [*simp*,*noatp*]: *xs* ≠ [] ==> *hd xs* # *tl xs* = *xs*

by (*induct xs*) *auto*

lemma *hd-append*: *hd* (*xs* @ *ys*) = (*if xs* = [] *then hd ys else hd xs*)
by (*induct xs*) *auto*

lemma *hd-append2* [*simp*]: *xs* ≠ [] ==> *hd* (*xs* @ *ys*) = *hd xs*
by (*simp add*: *hd-append split*: *list.split*)

lemma *tl-append*: *tl* (*xs* @ *ys*) = (*case xs of* [] => *tl ys* | *z*#*zs* => *zs* @ *ys*)
by (*simp split*: *list.split*)

lemma *tl-append2* [*simp*]: *xs* ≠ [] ==> *tl* (*xs* @ *ys*) = *tl xs* @ *ys*
by (*simp add*: *tl-append split*: *list.split*)

lemma *Cons-eq-append-conv*: *x*#*xs* = *ys*@*zs* =
(*ys* = [] & *x*#*xs* = *zs* | (*EX ys'*. *x*#*ys'* = *ys* & *xs* = *ys'*@*zs*))
by(*cases ys*) *auto*

lemma *append-eq-Cons-conv*: (*ys*@*zs* = *x*#*xs*) =
(*ys* = [] & *zs* = *x*#*xs* | (*EX ys'*. *ys* = *x*#*ys'* & *ys'*@*zs* = *xs*))
by(*cases ys*) *auto*

Trivial rules for solving @-equations automatically.

lemma *eq-Nil-appendI*: *xs* = *ys* ==> *xs* = [] @ *ys*
by *simp*

lemma *Cons-eq-appendI*:
[| *x* # *xs1* = *ys*; *xs* = *xs1* @ *zs* |] ==> *x* # *xs* = *ys* @ *zs*
by (*drule sym*) *simp*

lemma *append-eq-appendI*:
[| *xs* @ *xs1* = *zs*; *ys* = *xs1* @ *us* |] ==> *xs* @ *ys* = *zs* @ *us*
by (*drule sym*) *simp*

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

ML ⟪
local

fun last (*cons as Const*(@{*const-name Cons*},-) $ - $ *xs*) =
(*case xs of Const*(@{*const-name Nil*},-) => *cons* | - => *last xs*)
| *last* (*Const*(@{*const-name append*},-) $ - $ *ys*) = *last ys*
| *last t* = *t*;

fun list1 (*Const*(@{*const-name Cons*},-) $ - $ *Const*(@{*const-name Nil*},-)) = *true*
| *list1* - = *false*;

```
fun butlast ((cons as Const(@{const-name Cons},-) $ x) $ xs) =
  (case xs of Const(@{const-name Nil},-) => xs | - => cons $ butlast xs)
  | butlast ((app as Const(@{const-name append},-) $ xs) $ ys) = app $ butlast ys
  | butlast xs = Const(@{const-name Nil},fastype-of xs);

val rearr-ss = HOL-basic-ss addsimps [@{thm append-assoc},
  @{thm append-Nil}, @{thm append-Cons}];

fun list-eq ss (F as (eq as Const(-,eqT)) $ lhs $ rhs) =
  let
    val lastl = last lhs and lastr = last rhs;
    fun rearr conv =
      let
        val lhs1 = butlast lhs and rhs1 = butlast rhs;
        val Type(-,listT::-) = eqT
        val appT = [listT,listT] ---> listT
        val app = Const(@{const-name append},appT)
        val F2 = eq $ (app$lhs1$lastl) $ (app$rhs1$lastr)
        val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (F,F2));
        val thm = Goal.prove (Simplifier.the-context ss) [] [] eq
          (K (simp-tac (Simplifier.inherit-context ss rearr-ss) 1));
      in SOME ((conv RS (thm RS trans)) RS eq-reflection) end;

  in
    if list1 lastl andalso list1 lastr then rearr @{thm append1-eq-conv}
    else if lastl aconv lastr then rearr @{thm append-same-eq}
    else NONE
  end;

in

val list-eq-simproc =
  Simplifier.simproc (the-context ()) list-eq [(xs::'a list) = ys] (K list-eq);

end;

Addsimprocs [list-eq-simproc];
⟫
```

39.1.5 *map*

lemma *map-ext*: (!!x. x : set xs ---> f x = g x) ==> map f xs = map g xs
by (*induct xs*) *simp-all*

lemma *map-ident* [*simp*]: map (λx. x) = (λxs. xs)
by (*rule ext*, *induct-tac xs*) *auto*

lemma *map-append* [*simp*]: map f (xs @ ys) = map f xs @ map f ys
by (*induct xs*) *auto*

lemma *map-compose*: *map* (*f o g*) *xs* = *map f* (*map g xs*)
by (*induct xs*) (*auto simp add*: *o-def*)

lemma *rev-map*: *rev* (*map f xs*) = *map f* (*rev xs*)
by (*induct xs*) *auto*

lemma *map-eq-conv*[*simp*]: (*map f xs* = *map g xs*) = (!*x* : *set xs*. *f x* = *g x*)
by (*induct xs*) *auto*

lemma *map-cong* [*fundef-cong*, *recdef-cong*]:
xs = *ys* ==> (!!*x*. *x* : *set ys* ==> *f x* = *g x*) ==> *map f xs* = *map g ys*
— a congruence rule for *map*
by *simp*

lemma *map-is-Nil-conv* [*iff*]: (*map f xs* = []) = (*xs* = [])
by (*cases xs*) *auto*

lemma *Nil-is-map-conv* [*iff*]: ([] = *map f xs*) = (*xs* = [])
by (*cases xs*) *auto*

lemma *map-eq-Cons-conv*:
(*map f xs* = *y*#*ys*) = (∃ *z zs*. *xs* = *z*#*zs* ∧ *f z* = *y* ∧ *map f zs* = *ys*)
by (*cases xs*) *auto*

lemma *Cons-eq-map-conv*:
(*x*#*xs* = *map f ys*) = (∃ *z zs*. *ys* = *z*#*zs* ∧ *x* = *f z* ∧ *xs* = *map f zs*)
by (*cases ys*) *auto*

lemmas *map-eq-Cons-D* = *map-eq-Cons-conv* [*THEN iffD1*]
lemmas *Cons-eq-map-D* = *Cons-eq-map-conv* [*THEN iffD1*]
declare *map-eq-Cons-D* [*dest*!] *Cons-eq-map-D* [*dest*!]

lemma *ex-map-conv*:
(*EX xs*. *ys* = *map f xs*) = (*ALL y* : *set ys*. *EX x*. *y* = *f x*)
by(*induct ys*, *auto simp add*: *Cons-eq-map-conv*)

lemma *map-eq-imp-length-eq*:
assumes *map f xs* = *map f ys*
shows *length xs* = *length ys*
using *assms* **proof** (*induct ys arbitrary*: *xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons y ys*) **then obtain** *z zs* **where** *xs*: *xs* = *z* # *zs* **by** *auto*
from *Cons xs* **have** *map f zs* = *map f ys* **by** *simp*
moreover with *Cons* **have** *length zs* = *length ys* **by** *blast*
with *xs* **show** *?case* **by** *simp*
qed

lemma *map-inj-on*:
[| *map f xs* = *map f ys*; *inj-on f* (*set xs Un set ys*) |]
==> *xs* = *ys*
apply(*frule map-eq-imp-length-eq*)
apply(*rotate-tac* −*1*)
apply(*induct rule*:*list-induct2*)
apply *simp*
apply(*simp*)
apply (*blast intro*:*sym*)
done

lemma *inj-on-map-eq-map*:
inj-on f (*set xs Un set ys*) ⟹ (*map f xs* = *map f ys*) = (*xs* = *ys*)
by(*blast dest*:*map-inj-on*)

lemma *map-injective*:
map f xs = *map f ys* ==> *inj f* ==> *xs* = *ys*
by (*induct ys arbitrary*: *xs*) (*auto dest*!:*injD*)

lemma *inj-map-eq-map*[*simp*]: *inj f* ⟹ (*map f xs* = *map f ys*) = (*xs* = *ys*)
by(*blast dest*:*map-injective*)

lemma *inj-mapI*: *inj f* ==> *inj* (*map f*)
by (*iprover dest*: *map-injective injD intro*: *inj-onI*)

lemma *inj-mapD*: *inj* (*map f*) ==> *inj f*
apply (*unfold inj-on-def*, *clarify*)
apply (*erule-tac x* = [*x*] **in** *ballE*)
apply (*erule-tac x* = [*y*] **in** *ballE*, *simp*, *blast*)
apply *blast*
done

lemma *inj-map*[*iff*]: *inj* (*map f*) = *inj f*
by (*blast dest*: *inj-mapD intro*: *inj-mapI*)

lemma *inj-on-mapI*: *inj-on f* (⋃(*set* ' *A*)) ⟹ *inj-on* (*map f*) *A*
apply(*rule inj-onI*)
apply(*erule map-inj-on*)
apply(*blast intro*:*inj-onI dest*:*inj-onD*)
done

lemma *map-idI*: (⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *x*) ⟹ *map f xs* = *xs*
by (*induct xs*, *auto*)

lemma *map-fun-upd* [*simp*]: *y* ∉ *set xs* ⟹ *map* (*f*(*y*:=*v*)) *xs* = *map f xs*
by (*induct xs*) *auto*

lemma *map-fst-zip*[*simp*]:
length xs = *length ys* ⟹ *map fst* (*zip xs ys*) = *xs*

by (*induct rule*:*list-induct2*, *simp-all*)

lemma *map-snd-zip*[*simp*]:
  length xs = *length ys* ⟹ *map snd* (*zip xs ys*) = *ys*
by (*induct rule*:*list-induct2*, *simp-all*)

39.1.6 *rev*

lemma *rev-append* [*simp*]: *rev* (*xs* @ *ys*) = *rev ys* @ *rev xs*
by (*induct xs*) *auto*

lemma *rev-rev-ident* [*simp*]: *rev* (*rev xs*) = *xs*
by (*induct xs*) *auto*

lemma *rev-swap*: (*rev xs* = *ys*) = (*xs* = *rev ys*)
by *auto*

lemma *rev-is-Nil-conv* [*iff*]: (*rev xs* = []) = (*xs* = [])
by (*induct xs*) *auto*

lemma *Nil-is-rev-conv* [*iff*]: ([] = *rev xs*) = (*xs* = [])
by (*induct xs*) *auto*

lemma *rev-singleton-conv* [*simp*]: (*rev xs* = [*x*]) = (*xs* = [*x*])
by (*cases xs*) *auto*

lemma *singleton-rev-conv* [*simp*]: ([*x*] = *rev xs*) = (*xs* = [*x*])
by (*cases xs*) *auto*

lemma *rev-is-rev-conv* [*iff*]: (*rev xs* = *rev ys*) = (*xs* = *ys*)
apply (*induct xs arbitrary*: *ys*, *force*)
apply (*case-tac ys*, *simp*, *force*)
done

lemma *inj-on-rev*[*iff*]: *inj-on rev A*
by(*simp add*:*inj-on-def*)

lemma *rev-induct* [*case-names Nil snoc*]:
  [| *P* []; !!*x xs*. *P xs* ==> *P* (*xs* @ [*x*]) |] ==> *P xs*
apply(*simplesubst rev-rev-ident*[*symmetric*])
apply(*rule-tac list* = *rev xs* **in** *list.induct*, *simp-all*)
done

lemma *rev-exhaust* [*case-names Nil snoc*]:
  (*xs* = [] ==> *P*) ==>(!!*ys y*. *xs* = *ys* @ [*y*] ==> *P*) ==> *P*
by (*induct xs rule*: *rev-induct*) *auto*

lemmas *rev-cases* = *rev-exhaust*

lemma *rev-eq-Cons-iff*[*iff*]: (*rev xs* = *y*#*ys*) = (*xs* = *rev ys* @ [*y*])
by(*rule rev-cases*[*of xs*]) *auto*

39.1.7 *set*

lemma *finite-set* [*iff*]: *finite* (*set xs*)
by (*induct xs*) *auto*

lemma *set-append* [*simp*]: *set* (*xs* @ *ys*) = (*set xs* ∪ *set ys*)
by (*induct xs*) *auto*

lemma *hd-in-set*[*simp*]: *xs* ≠ [] ⟹ *hd xs* : *set xs*
by(*cases xs*) *auto*

lemma *set-subset-Cons*: *set xs* ⊆ *set* (*x* # *xs*)
by *auto*

lemma *set-ConsD*: *y* ∈ *set* (*x* # *xs*) ⟹ *y*=*x* ∨ *y* ∈ *set xs*
by *auto*

lemma *set-empty* [*iff*]: (*set xs* = {}) = (*xs* = [])
by (*induct xs*) *auto*

lemma *set-empty2*[*iff*]: ({} = *set xs*) = (*xs* = [])
by(*induct xs*) *auto*

lemma *set-rev* [*simp*]: *set* (*rev xs*) = *set xs*
by (*induct xs*) *auto*

lemma *set-map* [*simp*]: *set* (*map f xs*) = *f*‘(*set xs*)
by (*induct xs*) *auto*

lemma *set-filter* [*simp*]: *set* (*filter P xs*) = {*x*. *x* : *set xs* ∧ *P x*}
by (*induct xs*) *auto*

lemma *set-upt* [*simp*]: *set*[*i*..<*j*] = {*k*. *i* ≤ *k* ∧ *k* < *j*}
apply (*induct j*, *simp-all*)
apply (*erule ssubst*, *auto*)
done

lemma *split-list*: *x* : *set xs* ⟹ ∃ *ys zs*. *xs* = *ys* @ *x* # *zs*
proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case *Cons* **thus** *?case* **by** (*auto intro*: *Cons-eq-appendI*)
qed

lemma *in-set-conv-decomp*: *x* ∈ *set xs* ⟷ (∃ *ys zs*. *xs* = *ys* @ *x* # *zs*)

by (*auto elim*: *split-list*)

lemma *split-list-first*: $x : set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ ys$
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a xs*)
show *?case*
proof *cases*
assume $x = a$ **thus** *?case* **using** *Cons* **by** *fastsimp*
next
assume $x \neq a$ **thus** *?case* **using** *Cons* **by**(*fastsimp intro*!: *Cons-eq-appendI*)
qed
qed

lemma *in-set-conv-decomp-first*:
$(x : set\ xs) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ ys)$
by (*auto dest*!: *split-list-first*)

lemma *split-list-last*: $x : set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ zs$
proof (*induct xs rule*:*rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc a xs*)
show *?case*
proof *cases*
assume $x = a$ **thus** *?case* **using** *snoc* **by** *simp* (*metis ex-in-conv set-empty2*)
next
assume $x \neq a$ **thus** *?case* **using** *snoc* **by** *fastsimp*
qed
qed

lemma *in-set-conv-decomp-last*:
$(x : set\ xs) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ zs)$
by (*auto dest*!: *split-list-last*)

lemma *split-list-prop*: $\exists\, x \in set\ xs.\ P\ x \Longrightarrow \exists\, ys\ x\ zs.\ xs = ys\ @\ x\ \#\ zs\ \&\ P\ x$
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case *Cons* **thus** *?case*
by(*simp add*:*Bex-def*)(*metis append-Cons append.simps*(*1*))
qed

lemma *split-list-propE*:
assumes $\exists\, x \in set\ xs.\ P\ x$
obtains *ys x zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** $P\ x$
using *split-list-prop* [*OF assms*] **by** *blast*

lemma *split-list-first-prop*:
  $\exists x \in set\ xs.\ P\ x \Longrightarrow$
   $\exists ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall y \in set\ ys.\ \neg\ P\ y)$
proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons x xs*)
  show *?case*
  proof *cases*
    assume *P x*
    thus *?thesis* **by** *simp*
      (*metis Un-upper1 contra-subsetD in-set-conv-decomp-first self-append-conv2 set-append*)
  next
    assume $\neg$ *P x*
    hence $\exists x{\in}set\ xs.\ P\ x$ **using** *Cons*(*2*) **by** *simp*
    thus *?thesis* **using** ‹$\neg$ *P x*› *Cons*(*1*) **by** (*metis append-Cons set-ConsD*)
  qed
qed

lemma *split-list-first-propE*:
  assumes $\exists x \in set\ xs.\ P\ x$
  obtains *ys x zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** *P x* **and** $\forall y \in set\ ys.\ \neg\ P\ y$
using *split-list-first-prop* [*OF assms*] **by** *blast*

lemma *split-list-first-prop-iff*:
  $(\exists x \in set\ xs.\ P\ x) \longleftrightarrow$
   $(\exists ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall y \in set\ ys.\ \neg\ P\ y))$
by (*rule*, *erule split-list-first-prop*) *auto*

lemma *split-list-last-prop*:
  $\exists x \in set\ xs.\ P\ x \Longrightarrow$
   $\exists ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall z \in set\ zs.\ \neg\ P\ z)$
proof(*induct xs rule:rev-induct*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*snoc x xs*)
  show *?case*
  proof *cases*
    assume *P x* **thus** *?thesis* **by** (*metis emptyE set-empty*)
  next
    assume $\neg$ *P x*
    hence $\exists x{\in}set\ xs.\ P\ x$ **using** *snoc*(*2*) **by** *simp*
    thus *?thesis* **using** ‹$\neg$ *P x*› *snoc*(*1*) **by** *fastsimp*
  qed
qed

lemma *split-list-last-propE*:
  assumes $\exists x \in set\ xs.\ P\ x$

obtains *ys x zs* **where** $xs = ys \mathbin{@} x \# zs$ **and** $P\ x$ **and** $\forall z \in set\ zs.\ \neg P\ z$
using *split-list-last-prop* [*OF assms*] **by** *blast*

lemma *split-list-last-prop-iff*:
$(\exists x \in set\ xs.\ P\ x) \longleftrightarrow$
$(\exists ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall z \in set\ zs.\ \neg P\ z))$
by (*metis split-list-last-prop* [**where** *P*=*P*] *in-set-conv-decomp*)

lemma *finite-list*: *finite A* ==> *EX xs. set xs* = *A*
by (*erule finite-induct*)
(*auto simp add*: *set.simps*(*2*) [*symmetric*] *simp del*: *set.simps*(*2*))

lemma *card-length*: *card* (*set xs*) ≤ *length xs*
by (*induct xs*) (*auto simp add*: *card-insert-if*)

lemma *set-minus-filter-out*:
set xs − {*y*} = *set* (*filter* (λ*x*. ¬ (*x* = *y*)) *xs*)
by (*induct xs*) *auto*

39.1.8 *filter*

lemma *filter-append* [*simp*]: *filter P* (*xs* @ *ys*) = *filter P xs* @ *filter P ys*
by (*induct xs*) *auto*

lemma *rev-filter*: *rev* (*filter P xs*) = *filter P* (*rev xs*)
by (*induct xs*) *simp-all*

lemma *filter-filter* [*simp*]: *filter P* (*filter Q xs*) = *filter* (λ*x*. *Q x* ∧ *P x*) *xs*
by (*induct xs*) *auto*

lemma *length-filter-le* [*simp*]: *length* (*filter P xs*) ≤ *length xs*
by (*induct xs*) (*auto simp add*: *le-SucI*)

lemma *sum-length-filter-compl*:
length(*filter P xs*) + *length*(*filter* (%*x*. ~*P x*) *xs*) = *length xs*
by(*induct xs*) *simp-all*

lemma *filter-True* [*simp*]: ∀ *x* ∈ *set xs*. *P x* ==> *filter P xs* = *xs*
by (*induct xs*) *auto*

lemma *filter-False* [*simp*]: ∀ *x* ∈ *set xs*. ¬ *P x* ==> *filter P xs* = []
by (*induct xs*) *auto*

lemma *filter-empty-conv*: (*filter P xs* = []) = (∀ *x*∈*set xs*. ¬ *P x*)
by (*induct xs*) *simp-all*

lemma *filter-id-conv*: (*filter P xs* = *xs*) = (∀ *x*∈*set xs*. *P x*)
apply (*induct xs*)
apply *auto*

apply(*cut-tac P=P* **and** *xs=xs* **in** *length-filter-le*)
apply *simp*
done

lemma *filter-map*:
filter P (map f xs) = map f (filter (P o f) xs)
by (*induct xs*) *simp-all*

lemma *length-filter-map*[*simp*]:
length (filter P (map f xs)) = length(*filter (P o f) xs*)
by (*simp add*:*filter-map*)

lemma *filter-is-subset* [*simp*]: *set (filter P xs) ≤ set xs*
by *auto*

lemma *length-filter-less*:
⟦ *x : set xs*; ~ *P x* ⟧ ⟹ *length*(*filter P xs*) < *length xs*
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case*
apply (*auto split*:*split-if-asm*)
using *length-filter-le*[*of P xs*] **apply** *arith*
done
qed

lemma *length-filter-conv-card*:
length(*filter p xs*) = *card*{*i*. *i* < *length xs* & *p*(*xs*!*i*)}
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
let *?S* = {*i*. *i* < *length xs* & *p*(*xs*!*i*)}
have *fin*: *finite ?S* **by**(*fast intro*: *bounded-nat-set-is-finite*)
show *?case* (**is** *?l* = *card ?S'*)
proof (*cases*)
assume *p x*
hence *eq*: *?S'* = *insert 0* (*Suc* ‘ *?S*)
by(*auto simp*: *image-def split*:*nat.split dest*:*gr0-implies-Suc*)
have *length* (*filter p* (*x* # *xs*)) = *Suc*(*card ?S*)
using *Cons* ‹*p x*› **by** *simp*
also have ... = *Suc*(*card*(*Suc* ‘ *?S*)) **using** *fin*
by (*simp add*: *card-image inj-Suc*)
also have ... = *card ?S'* **using** *eq fin*
by (*simp add*:*card-insert-if*) (*simp add*:*image-def*)
finally show *?thesis* **.**
next
assume ¬ *p x*
hence *eq*: *?S'* = *Suc* ‘ *?S*

```
    by(auto simp add: image-def split:nat.split elim:lessE)
  have length (filter p (x # xs)) = card ?S
    using Cons ‹¬ p x› by simp
  also have ... = card(Suc ` ?S) using fin
    by (simp add: card-image inj-Suc)
  also have ... = card ?S' using eq fin
    by (simp add:card-insert-if)
  finally show ?thesis .
 qed
qed

lemma Cons-eq-filterD:
 x#xs = filter P ys ⟹
  ∃us vs. ys = us @ x # vs ∧ (∀u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs
  (is - ⟹ ∃us vs. ?P ys us vs)
proof(induct ys)
 case Nil thus ?case by simp
next
 case (Cons y ys)
 show ?case (is ∃x. ?Q x)
 proof cases
   assume Py: P y
   show ?thesis
   proof cases
     assume x = y
     with Py Cons.prems have ?Q [] by simp
     then show ?thesis ..
   next
     assume x ≠ y
     with Py Cons.prems show ?thesis by simp
   qed
 next
   assume ¬ P y
   with Cons obtain us vs where ?P (y#ys) (y#us) vs by fastsimp
   then have ?Q (y#us) by simp
   then show ?thesis ..
 qed
qed

lemma filter-eq-ConsD:
 filter P ys = x#xs ⟹
  ∃us vs. ys = us @ x # vs ∧ (∀u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs
by(rule Cons-eq-filterD) simp

lemma filter-eq-Cons-iff:
 (filter P ys = x#xs) =
  (∃us vs. ys = us @ x # vs ∧ (∀u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs)
by(auto dest:filter-eq-ConsD)
```

lemma *Cons-eq-filter-iff*:
(*x*#*xs* = *filter P ys*) =
(∃ *us vs*. *ys* = *us* @ *x* # *vs* ∧ (∀ *u*∈*set us*. ¬ *P u*) ∧ *P x* ∧ *xs* = *filter P vs*)
by(*auto dest*:*Cons-eq-filterD*)

lemma *filter-cong*[*fundef-cong*, *recdef-cong*]:
xs = *ys* ⟹ (⋀*x*. *x* ∈ *set ys* ⟹ *P x* = *Q x*) ⟹ *filter P xs* = *filter Q ys*
apply *simp*
apply(*erule thin-rl*)
by (*induct ys*) *simp-all*

39.1.9 List partitioning

primrec *partition* :: ('*a* ⇒ *bool*) ⇒'*a list* ⇒ '*a list* × '*a list* **where**
partition P [] = ([], [])
| *partition P* (*x* # *xs*) =
(*let* (*yes*, *no*) = *partition P xs*
in if P x then (*x* # *yes*, *no*) *else* (*yes*, *x* # *no*))

lemma *partition-filter1*:
fst (*partition P xs*) = *filter P xs*
by (*induct xs*) (*auto simp add*: *Let-def split-def*)

lemma *partition-filter2*:
snd (*partition P xs*) = *filter* (*Not o P*) *xs*
by (*induct xs*) (*auto simp add*: *Let-def split-def*)

lemma *partition-P*:
assumes *partition P xs* = (*yes*, *no*)
shows (∀ *p* ∈ *set yes*. *P p*) ∧ (∀ *p* ∈ *set no*. ¬ *P p*)
proof −
from *assms* **have** *yes* = *fst* (*partition P xs*) **and** *no* = *snd* (*partition P xs*)
by *simp-all*
then show *?thesis* **by** (*simp-all add*: *partition-filter1 partition-filter2*)
qed

lemma *partition-set*:
assumes *partition P xs* = (*yes*, *no*)
shows *set yes* ∪ *set no* = *set xs*
proof −
from *assms* **have** *yes* = *fst* (*partition P xs*) **and** *no* = *snd* (*partition P xs*)
by *simp-all*
then show *?thesis* **by** (*auto simp add*: *partition-filter1 partition-filter2*)
qed

39.1.10 *concat*

lemma *concat-append* [*simp*]: *concat* (*xs* @ *ys*) = *concat xs* @ *concat ys*
by (*induct xs*) *auto*

lemma *concat-eq-Nil-conv* [*simp*]: (*concat xss* = []) = (∀ *xs* ∈ *set xss*. *xs* = [])
by (*induct xss*) *auto*

lemma *Nil-eq-concat-conv* [*simp*]: ([] = *concat xss*) = (∀ *xs* ∈ *set xss*. *xs* = [])
by (*induct xss*) *auto*

lemma *set-concat* [*simp*]: *set* (*concat xs*) = (*UN x*:*set xs*. *set x*)
by (*induct xs*) *auto*

lemma *concat-map-singleton*[*simp*]: *concat*(*map* (%*x*. [*f x*]) *xs*) = *map f xs*
by (*induct xs*) *auto*

lemma *map-concat*: *map f* (*concat xs*) = *concat* (*map* (*map f*) *xs*)
by (*induct xs*) *auto*

lemma *filter-concat*: *filter p* (*concat xs*) = *concat* (*map* (*filter p*) *xs*)
by (*induct xs*) *auto*

lemma *rev-concat*: *rev* (*concat xs*) = *concat* (*map rev* (*rev xs*))
by (*induct xs*) *auto*

39.1.11 *nth*

lemma *nth-Cons-0* [*simp*, *code*]: (*x* # *xs*)!*0* = *x*
by *auto*

lemma *nth-Cons-Suc* [*simp*, *code*]: (*x* # *xs*)!(*Suc n*) = *xs*!*n*
by *auto*

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

lemma *nth-append*:
(*xs* @ *ys*)!*n* = (*if n* < *length xs then xs*!*n else ys*!(*n* − *length xs*))
apply (*induct xs arbitrary*: *n*, *simp*)
apply (*case-tac n*, *auto*)
done

lemma *nth-append-length* [*simp*]: (*xs* @ *x* # *ys*) ! *length xs* = *x*
by (*induct xs*) *auto*

lemma *nth-append-length-plus*[*simp*]: (*xs* @ *ys*) ! (*length xs* + *n*) = *ys* ! *n*
by (*induct xs*) *auto*

lemma *nth-map* [*simp*]: *n* < *length xs* ==> (*map f xs*)!*n* = *f*(*xs*!*n*)
apply (*induct xs arbitrary*: *n*, *simp*)
apply (*case-tac n*, *auto*)
done

lemma *hd-conv-nth*: *xs* ≠ [] ⟹ *hd xs* = *xs*!*0*

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

lemma *list-eq-iff-nth-eq*:
(*xs* = *ys*) = (*length xs* = *length ys* ∧ (*ALL i*<*length xs*. *xs*!*i* = *ys*!*i*))
apply(*induct xs arbitrary*: *ys*)
apply *force*
apply(*case-tac ys*)
apply *simp*
apply(*simp add*:*nth-Cons split*:*nat.split*)**apply** *blast*
done

lemma *set-conv-nth*: *set xs* = {*xs*!*i* | *i*. *i* < *length xs*}
apply (*induct xs*, *simp*, *simp*)
apply *safe*
apply (*metis nat-case-0 nth.simps zero-less-Suc*)
apply (*metis less-Suc-eq-0-disj nth-Cons-Suc*)
apply (*case-tac i*, *simp*)
apply (*metis diff-Suc-Suc nat-case-Suc nth.simps zero-less-diff*)
done

lemma *in-set-conv-nth*: (*x* ∈ *set xs*) = (∃ *i* < *length xs*. *xs*!*i* = *x*)
by(*auto simp*:*set-conv-nth*)

lemma *list-ball-nth*: [| *n* < *length xs*; !*x* : *set xs*. *P x*|] ==> *P*(*xs*!*n*)
by (*auto simp add*: *set-conv-nth*)

lemma *nth-mem* [*simp*]: *n* < *length xs* ==> *xs*!*n* : *set xs*
by (*auto simp add*: *set-conv-nth*)

lemma *all-nth-imp-all-set*:
[| !*i* < *length xs*. *P*(*xs*!*i*); *x* : *set xs*|] ==> *P x*
by (*auto simp add*: *set-conv-nth*)

lemma *all-set-conv-all-nth*:
(∀ *x* ∈ *set xs*. *P x*) = (∀ *i*. *i* < *length xs* −−> *P* (*xs* ! *i*))
by (*auto simp add*: *set-conv-nth*)

lemma *rev-nth*:
n < *size xs* ⟹ *rev xs* ! *n* = *xs* ! (*length xs* − *Suc n*)
proof (*induct xs arbitrary*: *n*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
hence *n*: *n* < *Suc* (*length xs*) **by** *simp*
moreover
{ assume *n* < *length xs*
with *n* **obtain** *n'* **where** *length xs* − *n* = *Suc n'*
by (*cases length xs* − *n*, *auto*)

```
    moreover
    then have length xs − Suc n = n' by simp
    ultimately
    have xs ! (length xs − Suc n) = (x # xs) ! (length xs − n) by simp
  }
  ultimately
  show ?case by (clarsimp simp add: Cons nth-append)
qed
```

39.1.12 *list-update*

```
lemma length-list-update [simp]: length(xs[i:=x]) = length xs
by (induct xs arbitrary: i) (auto split: nat.split)

lemma nth-list-update:
i < length xs==> (xs[i:=x])!j = (if i = j then x else xs!j)
by (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

lemma nth-list-update-eq [simp]: i < length xs ==> (xs[i:=x])!i = x
by (simp add: nth-list-update)

lemma nth-list-update-neq [simp]: i ≠ j ==> xs[i:=x]!j = xs!j
by (induct xs arbitrary: i j) (auto simp add: nth-Cons split: nat.split)

lemma list-update-id[simp]: xs[i := xs!i] = xs
by (induct xs arbitrary: i) (simp-all split:nat.splits)

lemma list-update-beyond[simp]: length xs ≤ i ⟹ xs[i:=x] = xs
apply (induct xs arbitrary: i)
 apply simp
apply (case-tac i)
apply simp-all
done

lemma list-update-same-conv:
i < length xs ==> (xs[i := x] = xs) = (xs!i = x)
by (induct xs arbitrary: i) (auto split: nat.split)

lemma list-update-append1:
 i < size xs ⟹ (xs @ ys)[i:=x] = xs[i:=x] @ ys
apply (induct xs arbitrary: i, simp)
apply(simp split:nat.split)
done

lemma list-update-append:
  (xs @ ys) [n:= x] =
  (if n < length xs then xs[n:= x] @ ys else xs @ (ys [n−length xs:= x]))
by (induct xs arbitrary: n) (auto split:nat.splits)
```

lemma *list-update-length* [*simp*]:
(*xs* @ *x* # *ys*)[*length xs* := *y*] = (*xs* @ *y* # *ys*)
by (*induct xs*, *auto*)

lemma *update-zip*:
length xs = *length ys* ==>
(*zip xs ys*)[*i*:=*xy*] = *zip* (*xs*[*i*:=*fst xy*]) (*ys*[*i*:=*snd xy*])
by (*induct ys arbitrary*: *i xy xs*) (*auto*, *case-tac xs*, *auto split*: *nat.split*)

lemma *set-update-subset-insert*: *set*(*xs*[*i*:=*x*]) <= *insert x* (*set xs*)
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *set-update-subsetI*: [| *set xs* <= *A*; *x*:*A* |] ==> *set*(*xs*[*i* := *x*]) <= *A*
by (*blast dest*!: *set-update-subset-insert* [*THEN subsetD*])

lemma *set-update-memI*: *n* < *length xs* ⟹ *x* ∈ *set* (*xs*[*n* := *x*])
by (*induct xs arbitrary*: *n*) (*auto split*:*nat.splits*)

lemma *list-update-overwrite*:
xs [*i* := *x*, *i* := *y*] = *xs* [*i* := *y*]
apply (*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
done

lemma *list-update-swap*:
i ≠ *i*′ ⟹ *xs* [*i* := *x*, *i*′ := *x*′] = *xs* [*i*′ := *x*′, *i* := *x*]
apply (*induct xs arbitrary*: *i i*′)
apply *simp*
apply (*case-tac i*, *case-tac i*′)
apply *auto*
apply (*case-tac i*′)
apply *auto*
done

lemma *list-update-code* [*code*]:
[][*i* := *y*] = []
(*x* # *xs*)[0 := *y*] = *y* # *xs*
(*x* # *xs*)[*Suc i* := *y*] = *x* # *xs*[*i* := *y*]
by *simp-all*

39.1.13 *last* and *butlast*

lemma *last-snoc* [*simp*]: *last* (*xs* @ [*x*]) = *x*
by (*induct xs*) *auto*

lemma *butlast-snoc* [*simp*]: *butlast* (*xs* @ [*x*]) = *xs*
by (*induct xs*) *auto*

lemma *last-ConsL*: $xs = [] \Longrightarrow last(x\#xs) = x$
by(*simp add*:*last.simps*)

lemma *last-ConsR*: $xs \neq [] \Longrightarrow last(x\#xs) = last\ xs$
by(*simp add*:*last.simps*)

lemma *last-append*: $last(xs\ @\ ys) = (if\ ys = []\ then\ last\ xs\ else\ last\ ys)$
by (*induct xs*) (*auto*)

lemma *last-appendL*[*simp*]: $ys = [] \Longrightarrow last(xs\ @\ ys) = last\ xs$
by(*simp add*:*last-append*)

lemma *last-appendR*[*simp*]: $ys \neq [] \Longrightarrow last(xs\ @\ ys) = last\ ys$
by(*simp add*:*last-append*)

lemma *hd-rev*: $xs \neq [] \Longrightarrow hd(rev\ xs) = last\ xs$
by(*rule rev-exhaust*[*of xs*]) *simp-all*

lemma *last-rev*: $xs \neq [] \Longrightarrow last(rev\ xs) = hd\ xs$
by(*cases xs*) *simp-all*

lemma *last-in-set*[*simp*]: $as \neq [] \Longrightarrow last\ as \in set\ as$
by (*induct as*) *auto*

lemma *length-butlast* [*simp*]: $length\ (butlast\ xs) = length\ xs - 1$
by (*induct xs rule*: *rev-induct*) *auto*

lemma *butlast-append*:
$butlast\ (xs\ @\ ys) = (if\ ys = []\ then\ butlast\ xs\ else\ xs\ @\ butlast\ ys)$
by (*induct xs arbitrary*: *ys*) *auto*

lemma *append-butlast-last-id* [*simp*]:
$xs \neq [] ==> butlast\ xs\ @\ [last\ xs] = xs$
by (*induct xs*) *auto*

lemma *in-set-butlastD*: $x : set\ (butlast\ xs) ==> x : set\ xs$
by (*induct xs*) (*auto split*: *split-if-asm*)

lemma *in-set-butlast-appendI*:
$x : set\ (butlast\ xs) \mid x : set\ (butlast\ ys) ==> x : set\ (butlast\ (xs\ @\ ys))$
by (*auto dest*: *in-set-butlastD simp add*: *butlast-append*)

lemma *last-drop*[*simp*]: $n < length\ xs \Longrightarrow last\ (drop\ n\ xs) = last\ xs$
apply (*induct xs arbitrary*: *n*)
apply *simp*
apply (*auto split*:*nat.split*)
done

lemma *last-conv-nth*: *xs≠[] ⟹ last xs = xs!(length xs − 1)*
by(*induct xs*)(*auto simp*:*neq-Nil-conv*)

lemma *butlast-conv-take*: *butlast xs = take (length xs − 1) xs*
by (*induct xs*, *simp*, *case-tac xs*, *simp-all*)

39.1.14 *take* and *drop*

lemma *take-0* [*simp*]: *take 0 xs = []*
by (*induct xs*) *auto*

lemma *drop-0* [*simp*]: *drop 0 xs = xs*
by (*induct xs*) *auto*

lemma *take-Suc-Cons* [*simp*]: *take (Suc n) (x # xs) = x # take n xs*
by *simp*

lemma *drop-Suc-Cons* [*simp*]: *drop (Suc n) (x # xs) = drop n xs*
by *simp*

declare *take-Cons* [*simp del*] **and** *drop-Cons* [*simp del*]

lemma *take-1-Cons* [*simp*]: *take 1 (x # xs) = [x]*
unfolding *One-nat-def* **by** *simp*

lemma *drop-1-Cons* [*simp*]: *drop 1 (x # xs) = xs*
unfolding *One-nat-def* **by** *simp*

lemma *take-Suc*: *xs ~= [] ==> take (Suc n) xs = hd xs # take n (tl xs)*
by(*clarsimp simp add*:*neq-Nil-conv*)

lemma *drop-Suc*: *drop (Suc n) xs = drop n (tl xs)*
by(*cases xs*, *simp-all*)

lemma *take-tl*: *take n (tl xs) = tl (take (Suc n) xs)*
by (*induct xs arbitrary*: *n*) *simp-all*

lemma *drop-tl*: *drop n (tl xs) = tl*(*drop n xs*)
by(*induct xs arbitrary*: *n*, *simp-all add*:*drop-Cons drop-Suc split*:*nat.split*)

lemma *tl-take*: *tl (take n xs) = take (n − 1) (tl xs)*
by (*cases n*, *simp*, *cases xs*, *auto*)

lemma *tl-drop*: *tl (drop n xs) = drop n (tl xs)*
by (*simp only*: *drop-tl*)

lemma *nth-via-drop*: *drop n xs = y#ys ⟹ xs!n = y*
apply (*induct xs arbitrary*: *n*, *simp*)
apply(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)

done

lemma *take-Suc-conv-app-nth*:
i < length xs ⟹ take (Suc i) xs = take i xs @ [xs!i]
apply (*induct xs arbitrary*: *i*, *simp*)
apply (*case-tac i*, *auto*)
done

lemma *drop-Suc-conv-tl*:
i < length xs ⟹ (xs!i) # (drop (Suc i) xs) = drop i xs
apply (*induct xs arbitrary*: *i*, *simp*)
apply (*case-tac i*, *auto*)
done

lemma *length-take* [*simp*]: *length (take n xs) = min (length xs) n*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *length-drop* [*simp*]: *length (drop n xs) = (length xs − n)*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-all* [*simp*]: *length xs <= n ==> take n xs = xs*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-all* [*simp*]: *length xs <= n ==> drop n xs = []*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-append* [*simp*]:
take n (xs @ ys) = (take n xs @ take (n − length xs) ys)
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-append* [*simp*]:
drop n (xs @ ys) = drop n xs @ drop (n − length xs) ys
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-take* [*simp*]: *take n (take m xs) = take (min n m) xs*
apply (*induct m arbitrary*: *xs n*, *auto*)
apply (*case-tac xs*, *auto*)
apply (*case-tac n*, *auto*)
done

lemma *drop-drop* [*simp*]: *drop n (drop m xs) = drop (n + m) xs*
apply (*induct m arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *take-drop*: *take n (drop m xs) = drop m (take (n + m) xs)*
apply (*induct m arbitrary*: *xs n*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *drop-take*: *drop n* (*take m xs*) = *take* (*m*−*n*) (*drop n xs*)
apply(*induct xs arbitrary*: *m n*)
apply *simp*
apply(*simp add*: *take-Cons drop-Cons split*:*nat.split*)
done

lemma *append-take-drop-id* [*simp*]: *take n xs* @ *drop n xs* = *xs*
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *take-eq-Nil*[*simp*]: (*take n xs* = []) = (*n* = *0* ∨ *xs* = [])
apply(*induct xs arbitrary*: *n*)
apply *simp*
apply(*simp add*:*take-Cons split*:*nat.split*)
done

lemma *drop-eq-Nil*[*simp*]: (*drop n xs* = []) = (*length xs* <= *n*)
apply(*induct xs arbitrary*: *n*)
apply *simp*
apply(*simp add*:*drop-Cons split*:*nat.split*)
done

lemma *take-map*: *take n* (*map f xs*) = *map f* (*take n xs*)
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *drop-map*: *drop n* (*map f xs*) = *map f* (*drop n xs*)
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *rev-take*: *rev* (*take i xs*) = *drop* (*length xs* − *i*) (*rev xs*)
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *auto*)
done

lemma *rev-drop*: *rev* (*drop i xs*) = *take* (*length xs* − *i*) (*rev xs*)
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *auto*)
done

lemma *nth-take* [*simp*]: *i* < *n* ==> (*take n xs*)!*i* = *xs*!*i*
apply (*induct xs arbitrary*: *i n*, *auto*)
apply (*case-tac n*, *blast*)
apply (*case-tac i*, *auto*)
done

lemma *nth-drop* [*simp*]:
n + i <= length xs ==> (drop n xs)!i = xs!(n + i)
apply (*induct n arbitrary*: *xs i*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *butlast-take*:
n <= length xs ==> butlast (take n xs) = take (n - 1) xs
by (*simp add*: *butlast-conv-take min-max.inf-absorb1 min-max.inf-absorb2*)

lemma *butlast-drop*: *butlast (drop n xs) = drop n (butlast xs)*
by (*simp add*: *butlast-conv-take drop-take add-ac*)

lemma *take-butlast*: *n < length xs ==> take n (butlast xs) = take n xs*
by (*simp add*: *butlast-conv-take min-max.inf-absorb1*)

lemma *drop-butlast*: *drop n (butlast xs) = butlast (drop n xs)*
by (*simp add*: *butlast-conv-take drop-take add-ac*)

lemma *hd-drop-conv-nth*: ⟦ *xs ≠ []*; *n < length xs* ⟧ ⟹ *hd(drop n xs) = xs!n*
by(*simp add*: *hd-conv-nth*)

lemma *set-take-subset*: *set(take n xs) ⊆ set xs*
by(*induct xs arbitrary*: *n*)(*auto simp*:*take-Cons split*:*nat.split*)

lemma *set-drop-subset*: *set(drop n xs) ⊆ set xs*
by(*induct xs arbitrary*: *n*)(*auto simp*:*drop-Cons split*:*nat.split*)

lemma *in-set-takeD*: *x : set(take n xs)* ⟹ *x : set xs*
using *set-take-subset* **by** *fast*

lemma *in-set-dropD*: *x : set(drop n xs)* ⟹ *x : set xs*
using *set-drop-subset* **by** *fast*

lemma *append-eq-conv-conj*:
(xs @ ys = zs) = (xs = take (length xs) zs ∧ ys = drop (length xs) zs)
apply (*induct xs arbitrary*: *zs*, *simp*, *clarsimp*)
apply (*case-tac zs*, *auto*)
done

lemma *take-add*:
i+j ≤ length(xs) ⟹ *take (i+j) xs = take i xs @ take j (drop i xs)*
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *simp-all*)
done

lemma *append-eq-append-conv-if*:
(xs₁ @ xs₂ = ys₁ @ ys₂) =

(*if size* $xs_1 \leq$ *size* ys_1
then $xs_1 =$ *take* (*size* xs_1) $ys_1 \wedge xs_2 =$ *drop* (*size* xs_1) ys_1 @ ys_2
else take (*size* ys_1) $xs_1 = ys_1 \wedge$ *drop* (*size* ys_1) xs_1 @ $xs_2 = ys_2$)
apply(*induct* xs_1 *arbitrary*: ys_1)
apply *simp*
apply(*case-tac* ys_1)
apply *simp-all*
done

lemma *take-hd-drop*:
$n <$ *length xs* $\Longrightarrow$ *take n xs* @ [*hd* (*drop n xs*)] = *take* (*Suc n*) *xs*
apply(*induct xs arbitrary*: *n*)
apply *simp*
apply(*simp add*:*drop-Cons split*:*nat.split*)
done

lemma *id-take-nth-drop*:
$i <$ *length xs* $\Longrightarrow$ *xs* = *take i xs* @ *xs*!*i* # *drop* (*Suc i*) *xs*
proof −
assume *si*: $i <$ *length xs*
hence *xs* = *take* (*Suc i*) *xs* @ *drop* (*Suc i*) *xs* **by** *auto*
moreover
from *si* **have** *take* (*Suc i*) *xs* = *take i xs* @ [*xs*!*i*]
apply (*rule-tac take-Suc-conv-app-nth*) **by** *arith*
ultimately show *?thesis* **by** *auto*
qed

lemma *upd-conv-take-nth-drop*:
$i <$ *length xs* $\Longrightarrow$ *xs*[*i*:=*a*] = *take i xs* @ *a* # *drop* (*Suc i*) *xs*
proof −
assume *i*: $i <$ *length xs*
have *xs*[*i*:=*a*] = (*take i xs* @ *xs*!*i* # *drop* (*Suc i*) *xs*)[*i*:=*a*]
by(*rule arg-cong*[*OF id-take-nth-drop*[*OF i*]])
also have ... = *take i xs* @ *a* # *drop* (*Suc i*) *xs*
using *i* **by** (*simp add*: *list-update-append*)
finally show *?thesis* **.**
qed

lemma *nth-drop'*:
$i <$ *length xs* $\Longrightarrow$ *xs* ! *i* # *drop* (*Suc i*) *xs* = *drop i xs*
apply (*induct i arbitrary*: *xs*)
apply (*simp add*: *neq-Nil-conv*)
apply (*erule exE*)+
apply *simp*
apply (*case-tac xs*)
apply *simp-all*
done

39.1.15 *takeWhile* and *dropWhile*

lemma *takeWhile-dropWhile-id* [*simp*]: *takeWhile P xs @ dropWhile P xs = xs*
by (*induct xs*) *auto*

lemma *takeWhile-append1* [*simp*]:
[| *x*:*set xs*; ~*P*(*x*)|] ==> *takeWhile P* (*xs @ ys*) = *takeWhile P xs*
by (*induct xs*) *auto*

lemma *takeWhile-append2* [*simp*]:
(!!*x*. *x* : *set xs* ==> *P x*) ==> *takeWhile P* (*xs @ ys*) = *xs @ takeWhile P ys*
by (*induct xs*) *auto*

lemma *takeWhile-tail*: ¬ *P x* ==> *takeWhile P* (*xs @* (*x*#*l*)) = *takeWhile P xs*
by (*induct xs*) *auto*

lemma *dropWhile-append1* [*simp*]:
[| *x* : *set xs*; ~*P*(*x*)|] ==> *dropWhile P* (*xs @ ys*) = (*dropWhile P xs*)@*ys*
by (*induct xs*) *auto*

lemma *dropWhile-append2* [*simp*]:
(!!*x*. *x*:*set xs* ==> *P*(*x*)) ==> *dropWhile P* (*xs @ ys*) = *dropWhile P ys*
by (*induct xs*) *auto*

lemma *set-takeWhileD*: *x* : *set* (*takeWhile P xs*) ==> *x* : *set xs* ∧ *P x*
by (*induct xs*) (*auto split*: *split-if-asm*)

lemma *takeWhile-eq-all-conv*[*simp*]:
(*takeWhile P xs* = *xs*) = (∀ *x* ∈ *set xs*. *P x*)
by(*induct xs*, *auto*)

lemma *dropWhile-eq-Nil-conv*[*simp*]:
(*dropWhile P xs* = []) = (∀ *x* ∈ *set xs*. *P x*)
by(*induct xs*, *auto*)

lemma *dropWhile-eq-Cons-conv*:
(*dropWhile P xs* = *y*#*ys*) = (*xs* = *takeWhile P xs @ y* # *ys* & ¬ *P y*)
by(*induct xs*, *auto*)

The following two lemmmas could be generalized to an arbitrary property.

lemma *takeWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
takeWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *rev* (*tl* (*dropWhile* (λ*y*. *y* ≠ *x*) *xs*))
by(*induct xs*) (*auto simp*: *takeWhile-tail*[**where** *l*=[]])

lemma *dropWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
dropWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *x* # *rev* (*takeWhile* (λ*y*. *y* ≠ *x*) *xs*)
apply(*induct xs*)
apply *simp*
apply *auto*
apply(*subst dropWhile-append2*)

apply *auto*
done

lemma *takeWhile-not-last*:
⟦ *xs* ≠ []; *distinct xs*⟧ ⟹ *takeWhile* (λ*y*. *y* ≠ *last xs*) *xs* = *butlast xs*
apply(*induct xs*)
apply *simp*
apply(*case-tac xs*)
apply(*auto*)
done

lemma *takeWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| *l* = *k*; !!*x*. *x* : *set l* ==> *P x* = *Q x* |]
==> *takeWhile P l* = *takeWhile Q k*
by (*induct k arbitrary*: *l*) (*simp-all*)

lemma *dropWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| *l* = *k*; !!*x*. *x* : *set l* ==> *P x* = *Q x* |]
==> *dropWhile P l* = *dropWhile Q k*
by (*induct k arbitrary*: *l*, *simp-all*)

39.1.16 *zip*

lemma *zip-Nil* [*simp*]: *zip* [] *ys* = []
by (*induct ys*) *auto*

lemma *zip-Cons-Cons* [*simp*]: *zip* (*x* # *xs*) (*y* # *ys*) = (*x*, *y*) # *zip xs ys*
by *simp*

declare *zip-Cons* [*simp del*]

lemma *zip-Cons1*:
zip (*x*#*xs*) *ys* = (*case ys of* [] ⇒ [] | *y*#*ys* ⇒ (*x*,*y*)#*zip xs ys*)
by(*auto split*:*list*.*split*)

lemma *length-zip* [*simp*]:
length (*zip xs ys*) = *min* (*length xs*) (*length ys*)
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *zip-append1*:
zip (*xs* @ *ys*) *zs* =
zip xs (*take* (*length xs*) *zs*) @ *zip ys* (*drop* (*length xs*) *zs*)
by (*induct xs zs rule*:*list-induct2'*) *auto*

lemma *zip-append2*:
zip xs (*ys* @ *zs*) =
zip (*take* (*length ys*) *xs*) *ys* @ *zip* (*drop* (*length ys*) *xs*) *zs*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *zip-append* [*simp*]:
[| *length xs* = *length us*; *length ys* = *length vs* |] ==>
zip (*xs*@*ys*) (*us*@*vs*) = *zip xs us* @ *zip ys vs*
by (*simp add*: *zip-append1*)

lemma *zip-rev*:
length xs = *length ys* ==> *zip* (*rev xs*) (*rev ys*) = *rev* (*zip xs ys*)
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *map-zip-map*:
map f (*zip* (*map g xs*) *ys*) = *map* (%(*x*,*y*). *f*(*g x*, *y*)) (*zip xs ys*)
apply(*induct xs arbitrary*:*ys*) **apply** *simp*
apply(*case-tac ys*)
apply *simp-all*
done

lemma *map-zip-map2*:
map f (*zip xs* (*map g ys*)) = *map* (%(*x*,*y*). *f*(*x*, *g y*)) (*zip xs ys*)
apply(*induct xs arbitrary*:*ys*) **apply** *simp*
apply(*case-tac ys*)
apply *simp-all*
done

lemma *nth-zip* [*simp*]:
[| *i* < *length xs*; *i* < *length ys*|] ==> (*zip xs ys*)!*i* = (*xs*!*i*, *ys*!*i*)
apply (*induct ys arbitrary*: *i xs*, *simp*)
apply (*case-tac xs*)
apply (*simp-all add*: *nth.simps split*: *nat.split*)
done

lemma *set-zip*:
set (*zip xs ys*) = {(*xs*!*i*, *ys*!*i*) | *i*. *i* < *min* (*length xs*) (*length ys*)}
by (*simp add*: *set-conv-nth cong*: *rev-conj-cong*)

lemma *zip-update*:
length xs = *length ys* ==> *zip* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*]) = (*zip xs ys*)[*i*:=(*x*,*y*)]
by (*rule sym*, *simp add*: *update-zip*)

lemma *zip-replicate* [*simp*]:
zip (*replicate i x*) (*replicate j y*) = *replicate* (*min i j*) (*x*,*y*)
apply (*induct i arbitrary*: *j*, *auto*)
apply (*case-tac j*, *auto*)
done

lemma *take-zip*:
take n (*zip xs ys*) = *zip* (*take n xs*) (*take n ys*)
apply (*induct n arbitrary*: *xs ys*)
apply *simp*
apply (*case-tac xs*, *simp*)

apply (*case-tac ys*, *simp-all*)
done

lemma *drop-zip*:
drop n (*zip xs ys*) = *zip* (*drop n xs*) (*drop n ys*)
apply (*induct n arbitrary*: *xs ys*)
apply *simp*
apply (*case-tac xs*, *simp*)
apply (*case-tac ys*, *simp-all*)
done

lemma *set-zip-leftD*:
(*x*,*y*)∈ *set* (*zip xs ys*) ⟹ *x* ∈ *set xs*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *set-zip-rightD*:
(*x*,*y*)∈ *set* (*zip xs ys*) ⟹ *y* ∈ *set ys*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *in-set-zipE*:
(*x*,*y*) : *set*(*zip xs ys*) ⟹ (⟦ *x* : *set xs*; *y* : *set ys* ⟧ ⟹ *R*) ⟹ *R*
by(*blast dest*: *set-zip-leftD set-zip-rightD*)

lemma *zip-map-fst-snd*:
zip (*map fst zs*) (*map snd zs*) = *zs*
by (*induct zs*) *simp-all*

lemma *zip-eq-conv*:
length xs = *length ys* ⟹ *zip xs ys* = *zs* ⟷ *map fst zs* = *xs* ∧ *map snd zs* = *ys*
by (*auto simp add*: *zip-map-fst-snd*)

39.1.17 *list-all2*

lemma *list-all2-lengthD* [*intro?*]:
list-all2 P xs ys ==> *length xs* = *length ys*
by (*simp add*: *list-all2-def*)

lemma *list-all2-Nil* [*iff*, *code*]: *list-all2 P* [] *ys* = (*ys* = [])
by (*simp add*: *list-all2-def*)

lemma *list-all2-Nil2* [*iff*, *code*]: *list-all2 P xs* [] = (*xs* = [])
by (*simp add*: *list-all2-def*)

lemma *list-all2-Cons* [*iff*, *code*]:
list-all2 P (*x* # *xs*) (*y* # *ys*) = (*P x y* ∧ *list-all2 P xs ys*)
by (*auto simp add*: *list-all2-def*)

lemma *list-all2-Cons1*:
list-all2 P (*x* # *xs*) *ys* = (∃ *z zs*. *ys* = *z* # *zs* ∧ *P x z* ∧ *list-all2 P xs zs*)

by (*cases ys*) *auto*

lemma *list-all2-Cons2*:
list-all2 P xs (*y # ys*) = (∃ *z zs*. *xs* = *z # zs* ∧ *P z y* ∧ *list-all2 P zs ys*)
by (*cases xs*) *auto*

lemma *list-all2-rev* [*iff*]:
list-all2 P (*rev xs*) (*rev ys*) = *list-all2 P xs ys*
by (*simp add*: *list-all2-def zip-rev cong*: *conj-cong*)

lemma *list-all2-rev1*:
list-all2 P (*rev xs*) *ys* = *list-all2 P xs* (*rev ys*)
by (*subst list-all2-rev* [*symmetric*]) *simp*

lemma *list-all2-append1*:
list-all2 P (*xs* @ *ys*) *zs* =
(*EX us vs*. *zs* = *us* @ *vs* ∧ *length us* = *length xs* ∧ *length vs* = *length ys* ∧
list-all2 P xs us ∧ *list-all2 P ys vs*)
apply (*simp add*: *list-all2-def zip-append1*)
apply (*rule iffI*)
apply (*rule-tac x* = *take* (*length xs*) *zs* **in** *exI*)
apply (*rule-tac x* = *drop* (*length xs*) *zs* **in** *exI*)
apply (*force split*: *nat-diff-split simp add*: *min-def*, *clarify*)
apply (*simp add*: *ball-Un*)
done

lemma *list-all2-append2*:
list-all2 P xs (*ys* @ *zs*) =
(*EX us vs*. *xs* = *us* @ *vs* ∧ *length us* = *length ys* ∧ *length vs* = *length zs* ∧
list-all2 P us ys ∧ *list-all2 P vs zs*)
apply (*simp add*: *list-all2-def zip-append2*)
apply (*rule iffI*)
apply (*rule-tac x* = *take* (*length ys*) *xs* **in** *exI*)
apply (*rule-tac x* = *drop* (*length ys*) *xs* **in** *exI*)
apply (*force split*: *nat-diff-split simp add*: *min-def*, *clarify*)
apply (*simp add*: *ball-Un*)
done

lemma *list-all2-append*:
length xs = *length ys* ⟹
list-all2 P (*xs*@*us*) (*ys*@*vs*) = (*list-all2 P xs ys* ∧ *list-all2 P us vs*)
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *list-all2-appendI* [*intro?*, *trans*]:
⟦ *list-all2 P a b*; *list-all2 P c d* ⟧ ⟹ *list-all2 P* (*a*@*c*) (*b*@*d*)
by (*simp add*: *list-all2-append list-all2-lengthD*)

lemma *list-all2-conv-all-nth*:
list-all2 P xs ys =

(*length xs = length ys* ∧ (∀ *i* < *length xs*. *P* (*xs*!*i*) (*ys*!*i*)))
by (*force simp add*: *list-all2-def set-zip*)

lemma *list-all2-trans*:
assumes *tr*: !!*a b c*. *P1 a b* ==> *P2 b c* ==> *P3 a c*
shows !!*bs cs*. *list-all2 P1 as bs* ==> *list-all2 P2 bs cs* ==> *list-all2 P3 as cs*
(**is** !!*bs cs*. *PROP ?Q as bs cs*)
proof (*induct as*)
fix *x xs bs* **assume** *I1*: !!*bs cs*. *PROP ?Q xs bs cs*
show !!*cs*. *PROP ?Q* (*x* # *xs*) *bs cs*
proof (*induct bs*)
fix *y ys cs* **assume** *I2*: !!*cs*. *PROP ?Q* (*x* # *xs*) *ys cs*
show *PROP ?Q* (*x* # *xs*) (*y* # *ys*) *cs*
by (*induct cs*) (*auto intro*: *tr I1 I2*)
qed *simp*
qed *simp*

lemma *list-all2-all-nthI* [*intro?*]:
length a = length b ⟹ (⋀*n*. *n* < *length a* ⟹ *P* (*a*!*n*) (*b*!*n*)) ⟹ *list-all2 P a b*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2I*:
∀ *x* ∈ *set* (*zip a b*). *split P x* ⟹ *length a = length b* ⟹ *list-all2 P a b*
by (*simp add*: *list-all2-def*)

lemma *list-all2-nthD*:
⟦ *list-all2 P xs ys*; *p* < *size xs* ⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-nthD2*:
⟦*list-all2 P xs ys*; *p* < *size ys*⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
by (*frule list-all2-lengthD*) (*auto intro*: *list-all2-nthD*)

lemma *list-all2-map1*:
list-all2 P (*map f as*) *bs* = *list-all2* (λ*x y*. *P* (*f x*) *y*) *as bs*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-map2*:
list-all2 P as (*map f bs*) = *list-all2* (λ*x y*. *P x* (*f y*)) *as bs*
by (*auto simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-refl* [*intro?*]:
(⋀*x*. *P x x*) ⟹ *list-all2 P xs xs*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-update-cong*:
⟦ *i*<*size xs*; *list-all2 P xs ys*; *P x y* ⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
by (*simp add*: *list-all2-conv-all-nth nth-list-update*)

lemma *list-all2-update-cong2*:
⟦*list-all2 P xs ys*; *P x y*; *i* < *length ys*⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
by (*simp add*: *list-all2-lengthD list-all2-update-cong*)

lemma *list-all2-takeI* [*simp*,*intro?*]:
list-all2 P xs ys ⟹ *list-all2 P* (*take n xs*) (*take n ys*)
apply (*induct xs arbitrary*: *n ys*)
apply *simp*
apply (*clarsimp simp add*: *list-all2-Cons1*)
apply (*case-tac n*)
apply *auto*
done

lemma *list-all2-dropI* [*simp*,*intro?*]:
list-all2 P as bs ⟹ *list-all2 P* (*drop n as*) (*drop n bs*)
apply (*induct as arbitrary*: *n bs*, *simp*)
apply (*clarsimp simp add*: *list-all2-Cons1*)
apply (*case-tac n*, *simp*, *simp*)
done

lemma *list-all2-mono* [*intro?*]:
list-all2 P xs ys ⟹ (⋀*xs ys*. *P xs ys* ⟹ *Q xs ys*) ⟹ *list-all2 Q xs ys*
apply (*induct xs arbitrary*: *ys*, *simp*)
apply (*case-tac ys*, *auto*)
done

lemma *list-all2-eq*:
xs = *ys* ⟷ *list-all2* (*op* =) *xs ys*
by (*induct xs ys rule*: *list-induct2′*) *auto*

39.1.18 *foldl* and *foldr*

lemma *foldl-append* [*simp*]:
foldl f a (*xs* @ *ys*) = *foldl f* (*foldl f a xs*) *ys*
by (*induct xs arbitrary*: *a*) *auto*

lemma *foldr-append*[*simp*]: *foldr f* (*xs* @ *ys*) *a* = *foldr f xs* (*foldr f ys a*)
by (*induct xs*) *auto*

lemma *foldr-map*: *foldr g* (*map f xs*) *a* = *foldr* (*g o f*) *xs a*
by(*induct xs*) *simp-all*

For efficient code generation: avoid intermediate list.

lemma *foldl-map*[*code unfold*]:
foldl g a (*map f xs*) = *foldl* (%*a x*. *g a* (*f x*)) *a xs*
by(*induct xs arbitrary*:*a*) *simp-all*

lemma *foldl-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f a x* = *g a x* |]

```
  ==> foldl f a l = foldl g b k
by (induct k arbitrary: a b l) simp-all

lemma foldr-cong [fundef-cong, recdef-cong]:
  [| a = b; l = k; !!a x. x : set l ==> f x a = g x a |]
  ==> foldr f l a = foldr g k b
by (induct k arbitrary: a b l) simp-all

lemma (in semigroup-add) foldl-assoc:
shows foldl op+ (x+y) zs = x + (foldl op+ y zs)
by (induct zs arbitrary: y) (simp-all add:add-assoc)

lemma (in monoid-add) foldl-absorb0:
shows x + (foldl op+ 0 zs) = foldl op+ x zs
by (induct zs) (simp-all add:foldl-assoc)
```

The "First Duality Theorem" in Bird & Wadler:

```
lemma foldl-foldr1-lemma:
 foldl op + a xs = a + foldr op + xs (0::'a::monoid-add)
by (induct xs arbitrary: a) (auto simp:add-assoc)

corollary foldl-foldr1:
 foldl op + 0 xs = foldr op + xs (0::'a::monoid-add)
by (simp add:foldl-foldr1-lemma)
```

The "Third Duality Theorem" in Bird & Wadler:

```
lemma foldr-foldl: foldr f xs a = foldl (%x y. f y x) a (rev xs)
by (induct xs) auto

lemma foldl-foldr: foldl f a xs = foldr (%x y. f y x) (rev xs) a
by (simp add: foldr-foldl [of %x y. f y x rev xs])

lemma (in ab-semigroup-add) foldr-conv-foldl: foldr op + xs a = foldl op + a xs
  by (induct xs, auto simp add: foldl-assoc add-commute)
```

Note: $n \leq$ *foldl* $(op +)$ *n ns* looks simpler, but is more difficult to use because it requires an additional transitivity step.

```
lemma start-le-sum: (m::nat) <= n ==> m <= foldl (op +) n ns
by (induct ns arbitrary: n) auto

lemma elem-le-sum: (n::nat) : set ns ==> n <= foldl (op +) 0 ns
by (force intro: start-le-sum simp add: in-set-conv-decomp)

lemma sum-eq-0-conv [iff]:
  (foldl (op +) (m::nat) ns = 0) = (m = 0 ∧ (∀ n ∈ set ns. n = 0))
by (induct ns arbitrary: m) auto

lemma foldr-invariant:
  ⟦Q x ; ∀ x∈ set xs. P x; ∀ x y. P x ∧ Q y ⟶ Q (f x y) ⟧ ⟹ Q (foldr f xs x)
```

by (*induct xs*, *simp-all*)

lemma *foldl-invariant*:
⟦*Q x* ; ∀ *x*∈ *set xs*. *P x*; ∀ *x y*. *P x* ∧ *Q y* ⟶ *Q* (*f y x*) ⟧ ⟹ *Q* (*foldl f x xs*)
by (*induct xs arbitrary*: *x*, *simp-all*)

foldl and *concat*

lemma *foldl-conv-concat*:
foldl (*op* @) *xs xss* = *xs* @ *concat xss*
proof (*induct xss arbitrary*: *xs*)
case *Nil* **show** *?case* **by** *simp*
next
interpret *monoid-add* [] *op* @ **proof qed** *simp-all*
case *Cons* **then show** *?case* **by** (*simp add*: *foldl-absorb0*)
qed

lemma *concat-conv-foldl*: *concat xss* = *foldl* (*op* @) [] *xss*
by (*simp add*: *foldl-conv-concat*)

39.1.19 List summation: *listsum* and ∑

lemma *listsum-append* [*simp*]: *listsum* (*xs* @ *ys*) = *listsum xs* + *listsum ys*
by (*induct xs*) (*simp-all add*:*add-assoc*)

lemma *listsum-rev* [*simp*]:
fixes *xs* :: *'a*::*comm-monoid-add list*
shows *listsum* (*rev xs*) = *listsum xs*
by (*induct xs*) (*simp-all add*:*add-ac*)

lemma *listsum-foldr*: *listsum xs* = *foldr* (*op* +) *xs 0*
by (*induct xs*) *auto*

lemma *length-concat*: *length* (*concat xss*) = *listsum* (*map length xss*)
by (*induct xss*) *simp-all*

For efficient code generation — *listsum* is not tail recursive but *foldl* is.

lemma *listsum*[*code unfold*]: *listsum xs* = *foldl* (*op* +) *0 xs*
by(*simp add*:*listsum-foldr foldl-foldr1*)

Some syntactic sugar for summing a function over a list:

syntax
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3SUM -<−-. -*) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*∑ *-←-. -*) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*∑ *-←-. -*) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM x<−xs. b == *CONST listsum* (*map* (%*x. b*) *xs*)

$\sum x{\leftarrow}xs.\ b == CONST\ listsum\ (map\ (\%x.\ b)\ xs)$

lemma *listsum-triv*: $(\sum x{\leftarrow}xs.\ r) = of\text{-}nat\ (length\ xs) * r$
by (*induct xs*) (*simp-all add*: *left-distrib*)

lemma *listsum-0* [*simp*]: $(\sum x{\leftarrow}xs.\ 0) = 0$
by (*induct xs*) (*simp-all add*: *left-distrib*)

For non-Abelian groups *xs* needs to be reversed on one side:

lemma *uminus-listsum-map*:
fixes $f :: 'a \Rightarrow 'b$::*ab-group-add*
shows $-\ listsum\ (map\ f\ xs) = (listsum\ (map\ (uminus\ o\ f)\ xs))$
by (*induct xs*) *simp-all*

39.1.20 *upt*

lemma *upt-rec*[*code*]: $[i..<j] = (if\ i<j\ then\ i\#[Suc\ i..<j]\ else\ [])$
— simp does not terminate!
by (*induct j*) *auto*

lemma *upt-conv-Nil* [*simp*]: $j <= i ==> [i..<j] = []$
by (*subst upt-rec*) *simp*

lemma *upt-eq-Nil-conv*[*simp*]: $([i..<j] = []) = (j = 0 \vee j <= i)$
by(*induct j*)*simp-all*

lemma *upt-eq-Cons-conv*:
$([i..<j] = x\#xs) = (i < j\ \&\ i = x\ \&\ [i+1..<j] = xs)$
apply(*induct j arbitrary*: *x xs*)
apply *simp*
apply(*clarsimp simp add*: *append-eq-Cons-conv*)
apply *arith*
done

lemma *upt-Suc-append*: $i <= j ==> [i..<(Suc\ j)] = [i..<j]@[j]$
— Only needed if *upt-Suc* is deleted from the simpset.
by *simp*

lemma *upt-conv-Cons*: $i < j ==> [i..<j] = i\ \#\ [Suc\ i..<j]$
by (*simp add*: *upt-rec*)

lemma *upt-add-eq-append*: $i<=j ==> [i..<j+k] = [i..<j]@[j..<j+k]$
— LOOPS as a simprule, since $j <= j$.
by (*induct k*) *auto*

lemma *length-upt* [*simp*]: $length\ [i..<j] = j - i$
by (*induct j*) (*auto simp add*: *Suc-diff-le*)

lemma *nth-upt* [*simp*]: $i + k < j ==> [i..<j]\ !\ k = i + k$

apply (*induct j*)
apply (*auto simp add*: *less-Suc-eq nth-append split*: *nat-diff-split*)
done

lemma *hd-upt*[*simp*]: $i < j \Longrightarrow hd[i..<j] = i$
by(*simp add*:*upt-conv-Cons*)

lemma *last-upt*[*simp*]: $i < j \Longrightarrow last[i..<j] = j - 1$
apply(*cases j*)
apply *simp*
by(*simp add*:*upt-Suc-append*)

lemma *take-upt* [*simp*]: $i+m <= n ==> take\ m\ [i..<n] = [i..<i+m]$
apply (*induct m arbitrary*: *i*, *simp*)
apply (*subst upt-rec*)
apply (*rule sym*)
apply (*subst upt-rec*)
apply (*simp del*: *upt.simps*)
done

lemma *drop-upt*[*simp*]: *drop m* $[i..<j] = [i+m..<j]$
apply(*induct j*)
apply *auto*
done

lemma *map-Suc-upt*: *map Suc* $[m..<n] = [Suc\ m..<Suc\ n]$
by (*induct n*) *auto*

lemma *nth-map-upt*: $i < n-m ==> (map\ f\ [m..<n])\ !\ i = f(m+i)$
apply (*induct n m arbitrary*: *i rule*: *diff-induct*)
prefer *3* **apply** (*subst map-Suc-upt*[*symmetric*])
apply (*auto simp add*: *less-diff-conv nth-upt*)
done

lemma *nth-take-lemma*:
$k <= length\ xs ==> k <= length\ ys ==>$
$(!!i.\ i < k --> xs!i = ys!i) ==> take\ k\ xs = take\ k\ ys$
apply (*atomize*, *induct k arbitrary*: *xs ys*)
apply (*simp-all add*: *less-Suc-eq-0-disj all-conj-distrib*, *clarify*)

Both lists must be non-empty

apply (*case-tac xs*, *simp*)
apply (*case-tac ys*, *clarify*)
apply (*simp* (*no-asm-use*))
apply *clarify*

prenexing's needed, not miniscoping

apply (*simp* (*no-asm-use*) *add*: *all-simps* [*symmetric*] *del*: *all-simps*)

apply *blast*
done

lemma *nth-equalityI*:
[| *length xs* = *length ys*; *ALL i* < *length xs*. *xs*!*i* = *ys*!*i* |] ==> *xs* = *ys*
apply (*frule nth-take-lemma* [*OF le-refl eq-imp-le*])
apply (*simp-all add*: *take-all*)
done

lemma *map-nth*:
map (λ*i*. *xs* ! *i*) [*0*..<*length xs*] = *xs*
by (*rule nth-equalityI*, *auto*)

lemma *list-all2-antisym*:
⟦ (⋀*x y*. ⟦*P x y*; *Q y x*⟧ ⟹ *x* = *y*); *list-all2 P xs ys*; *list-all2 Q ys xs* ⟧
⟹ *xs* = *ys*
apply (*simp add*: *list-all2-conv-all-nth*)
apply (*rule nth-equalityI*, *blast*, *simp*)
done

lemma *take-equalityI*: (∀ *i*. *take i xs* = *take i ys*) ==> *xs* = *ys*
— The famous take-lemma.
apply (*drule-tac x* = *max* (*length xs*) (*length ys*) **in** *spec*)
apply (*simp add*: *le-max-iff-disj take-all*)
done

lemma *take-Cons'*:
take n (*x* # *xs*) = (*if n* = *0 then* [] *else x* # *take* (*n* − *1*) *xs*)
by (*cases n*) *simp-all*

lemma *drop-Cons'*:
drop n (*x* # *xs*) = (*if n* = *0 then x* # *xs else drop* (*n* − *1*) *xs*)
by (*cases n*) *simp-all*

lemma *nth-Cons'*: (*x* # *xs*)!*n* = (*if n* = *0 then x else xs*!(*n* − *1*))
by (*cases n*) *simp-all*

lemmas *take-Cons-number-of* = *take-Cons'*[*of number-of v*,*standard*]
lemmas *drop-Cons-number-of* = *drop-Cons'*[*of number-of v*,*standard*]
lemmas *nth-Cons-number-of* = *nth-Cons'*[*of - - number-of v*,*standard*]

declare *take-Cons-number-of* [*simp*]
drop-Cons-number-of [*simp*]
nth-Cons-number-of [*simp*]

39.1.21 *distinct* and *remdups*

lemma *distinct-append* [*simp*]:
distinct (*xs* @ *ys*) = (*distinct xs* ∧ *distinct ys* ∧ *set xs* ∩ *set ys* = {})
by (*induct xs*) *auto*

lemma *distinct-rev*[*simp*]: *distinct*(*rev xs*) = *distinct xs*
by(*induct xs*) *auto*

lemma *set-remdups* [*simp*]: *set* (*remdups xs*) = *set xs*
by (*induct xs*) (*auto simp add*: *insert-absorb*)

lemma *distinct-remdups* [*iff*]: *distinct* (*remdups xs*)
by (*induct xs*) *auto*

lemma *distinct-remdups-id*: *distinct xs* ==> *remdups xs* = *xs*
by (*induct xs*, *auto*)

lemma *remdups-id-iff-distinct* [*simp*]: *remdups xs* = *xs* ⟷ *distinct xs*
by (*metis distinct-remdups distinct-remdups-id*)

lemma *finite-distinct-list*: *finite A* ⟹ *EX xs*. *set xs* = *A* & *distinct xs*
by (*metis distinct-remdups finite-list set-remdups*)

lemma *remdups-eq-nil-iff* [*simp*]: (*remdups x* = []) = (*x* = [])
by (*induct x*, *auto*)

lemma *remdups-eq-nil-right-iff* [*simp*]: ([] = *remdups x*) = (*x* = [])
by (*induct x*, *auto*)

lemma *length-remdups-leq*[*iff*]: *length*(*remdups xs*) <= *length xs*
by (*induct xs*) *auto*

lemma *length-remdups-eq*[*iff*]:
(*length* (*remdups xs*) = *length xs*) = (*remdups xs* = *xs*)
apply(*induct xs*)
apply *auto*
apply(*subgoal-tac length* (*remdups xs*) <= *length xs*)
apply *arith*
apply(*rule length-remdups-leq*)
done

lemma *distinct-map*:
distinct(*map f xs*) = (*distinct xs* & *inj-on f* (*set xs*))
by (*induct xs*) *auto*

lemma *distinct-filter* [*simp*]: *distinct xs* ==> *distinct* (*filter P xs*)
by (*induct xs*) *auto*

lemma *distinct-upt*[*simp*]: *distinct*[*i*..<*j*]
by (*induct j*) *auto*

lemma *distinct-take*[*simp*]: *distinct xs* ⟹ *distinct* (*take i xs*)
apply(*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
apply(*blast dest*:*in-set-takeD*)
done

lemma *distinct-drop*[*simp*]: *distinct xs* ⟹ *distinct* (*drop i xs*)
apply(*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
done

lemma *distinct-list-update*:
assumes *d*: *distinct xs* **and** *a*: *a* ∉ *set xs* − {*xs*!*i*}
shows *distinct* (*xs*[*i*:=*a*])
proof (*cases i* < *length xs*)
case *True*
with *a* **have** *a* ∉ *set* (*take i xs* @ *xs* ! *i* # *drop* (*Suc i*) *xs*) − {*xs*!*i*}
apply (*drule-tac id-take-nth-drop*) **by** *simp*
with *d True* **show** *?thesis*
apply (*simp add*: *upd-conv-take-nth-drop*)
apply (*drule subst* [*OF id-take-nth-drop*]) **apply** *assumption*
apply *simp* **apply** (*cases a* = *xs*!*i*) **apply** *simp* **by** *blast*
next
case *False* **with** *d* **show** *?thesis* **by** *auto*
qed

It is best to avoid this indexed version of distinct, but sometimes it is useful.

lemma *distinct-conv-nth*:
distinct xs = (∀ *i* < *size xs*. ∀ *j* < *size xs*. *i* ≠ *j* −−> *xs*!*i* ≠ *xs*!*j*)
apply (*induct xs*, *simp*, *simp*)
apply (*rule iffI*, *clarsimp*)
apply (*case-tac i*)
apply (*case-tac j*, *simp*)
apply (*simp add*: *set-conv-nth*)
apply (*case-tac j*)
apply (*clarsimp simp add*: *set-conv-nth*, *simp*)
apply (*rule conjI*)

apply (*clarsimp simp add*: *set-conv-nth*)
apply (*erule-tac x* = *0* **in** *allE*, *simp*)
apply (*erule-tac x* = *Suc i* **in** *allE*, *simp*, *clarsimp*)

apply (*erule-tac x = Suc i* **in** *allE*, *simp*)
apply (*erule-tac x = Suc j* **in** *allE*, *simp*)
done

lemma *nth-eq-iff-index-eq*:
⟦ *distinct xs*; *i < length xs*; *j < length xs* ⟧ ⟹ (*xs*!*i* = *xs*!*j*) = (*i* = *j*)
by(*auto simp*: *distinct-conv-nth*)

lemma *distinct-card*: *distinct xs* ==> *card* (*set xs*) = *size xs*
by (*induct xs*) *auto*

lemma *card-distinct*: *card* (*set xs*) = *size xs* ==> *distinct xs*
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof (*cases x ∈ set xs*)
case *False* **with** *Cons* **show** *?thesis* **by** *simp*
next
case *True* **with** *Cons.prems*
have *card* (*set xs*) = *Suc* (*length xs*)
by (*simp add*: *card-insert-if split*: *split-if-asm*)
moreover have *card* (*set xs*) ≤ *length xs* **by** (*rule card-length*)
ultimately have *False* **by** *simp*
thus *?thesis* **..**
qed
qed

lemma *not-distinct-decomp*: ~ *distinct ws* ==> *EX xs ys zs y*. *ws* = *xs*@[*y*]@*ys*@[*y*]@*zs*
apply (*induct n* == *length ws arbitrary*:*ws*) **apply** *simp*
apply(*case-tac ws*) **apply** *simp*
apply (*simp split*:*split-if-asm*)
apply (*metis Cons-eq-appendI eq-Nil-appendI split-list*)
done

lemma *length-remdups-concat*:
length(*remdups*(*concat xss*)) = *card*(⋃ *xs* ∈ *set xss*. *set xs*)
by(*simp add*: *set-concat distinct-card*[*symmetric*])

39.1.22 *remove1*

lemma *remove1-append*:
remove1 x (*xs* @ *ys*) =
(*if x ∈ set xs then remove1 x xs* @ *ys else xs* @ *remove1 x ys*)
by (*induct xs*) *auto*

lemma *in-set-remove1*[*simp*]:

$a \neq b \Longrightarrow a : set(remove1\ b\ xs) = (a : set\ xs)$
apply (*induct xs*)
apply *auto*
done

lemma *set-remove1-subset*: $set(remove1\ x\ xs) <= set\ xs$
apply(*induct xs*)
apply *simp*
apply *simp*
apply *blast*
done

lemma *set-remove1-eq* [*simp*]: $distinct\ xs ==> set(remove1\ x\ xs) = set\ xs - \{x\}$
apply(*induct xs*)
apply *simp*
apply *simp*
apply *blast*
done

lemma *length-remove1*:
$length(remove1\ x\ xs) = (if\ x : set\ xs\ then\ length\ xs - 1\ else\ length\ xs)$
apply (*induct xs*)
apply (*auto dest*!:*length-pos-if-in-set*)
done

lemma *remove1-filter-not*[*simp*]:
$\neg\ P\ x \Longrightarrow remove1\ x\ (filter\ P\ xs) = filter\ P\ xs$
by(*induct xs*) *auto*

lemma *notin-set-remove1*[*simp*]: $x\ ^{\sim}: set\ xs ==> x\ ^{\sim}: set(remove1\ y\ xs)$
apply(*insert set-remove1-subset*)
apply *fast*
done

lemma *distinct-remove1*[*simp*]: $distinct\ xs ==> distinct(remove1\ x\ xs)$
by (*induct xs*) *simp-all*

39.1.23 *removeAll*

lemma *removeAll-append*[*simp*]:
$removeAll\ x\ (xs\ @\ ys) = removeAll\ x\ xs\ @\ removeAll\ x\ ys$
by (*induct xs*) *auto*

lemma *set-removeAll*[*simp*]: $set(removeAll\ x\ xs) = set\ xs - \{x\}$
by (*induct xs*) *auto*

lemma *removeAll-id*[*simp*]: $x \notin set\ xs \Longrightarrow removeAll\ x\ xs = xs$
by (*induct xs*) *auto*

lemma *removeAll-filter-not*[*simp*]:
  $\neg\ P\ x \Longrightarrow$ *removeAll x* (*filter P xs*) = *filter P xs*
by(*induct xs*) *auto*

lemma *distinct-remove1-removeAll*:
  distinct xs ==> *remove1 x xs* = *removeAll x xs*
by (*induct xs*) *simp-all*

lemma *map-removeAll-inj-on*: *inj-on f* (*insert x* (*set xs*)) $\Longrightarrow$
  map f (*removeAll x xs*) = *removeAll* (*f x*) (*map f xs*)
by (*induct xs*) (*simp-all add*:*inj-on-def*)

lemma *map-removeAll-inj*: *inj f* $\Longrightarrow$
  map f (*removeAll x xs*) = *removeAll* (*f x*) (*map f xs*)
by(*metis map-removeAll-inj-on subset-inj-on subset-UNIV*)

39.1.24 *replicate*

lemma *length-replicate* [*simp*]: *length* (*replicate n x*) = *n*
by (*induct n*) *auto*

lemma *map-replicate* [*simp*]: *map f* (*replicate n x*) = *replicate n* (*f x*)
by (*induct n*) *auto*

lemma *replicate-app-Cons-same*:
(*replicate n x*) @ (*x* # *xs*) = *x* # *replicate n x* @ *xs*
by (*induct n*) *auto*

lemma *rev-replicate* [*simp*]: *rev* (*replicate n x*) = *replicate n x*
apply (*induct n*, *simp*)
apply (*simp add*: *replicate-app-Cons-same*)
done

lemma *replicate-add*: *replicate* (*n* + *m*) *x* = *replicate n x* @ *replicate m x*
by (*induct n*) *auto*

Courtesy of Matthias Daum:

lemma *append-replicate-commute*:
  replicate n x @ *replicate k x* = *replicate k x* @ *replicate n x*
apply (*simp add*: *replicate-add* [*THEN sym*])
apply (*simp add*: *add-commute*)
done

lemma *hd-replicate* [*simp*]: *n* ≠ *0* ==> *hd* (*replicate n x*) = *x*
by (*induct n*) *auto*

lemma *tl-replicate* [*simp*]: $n \neq 0 \Longrightarrow tl\ (replicate\ n\ x) = replicate\ (n - 1)\ x$
by (*induct n*) *auto*

lemma *last-replicate* [*simp*]: $n \neq 0 \Longrightarrow last\ (replicate\ n\ x) = x$
by (*atomize* (*full*), *induct n*) *auto*

lemma *nth-replicate*[*simp*]: $i < n \Longrightarrow (replicate\ n\ x)!i = x$
apply (*induct n arbitrary*: *i*, *simp*)
apply (*simp add*: *nth-Cons split*: *nat.split*)
done

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate*[*simp*]: *take i* (*replicate k x*) = *replicate* (*min i k*) *x*
apply (*case-tac* $k \leq i$)
apply (*simp add*: *min-def*)
apply (*drule not-leE*)
apply (*simp add*: *min-def*)
apply (*subgoal-tac replicate k x* = *replicate i x* @ *replicate* ($k - i$) *x*)
apply *simp*
apply (*simp add*: *replicate-add* [*symmetric*])
done

lemma *drop-replicate*[*simp*]: *drop i* (*replicate k x*) = *replicate* ($k-i$) *x*
apply (*induct k arbitrary*: *i*)
apply *simp*
apply *clarsimp*
apply (*case-tac i*)
apply *simp*
apply *clarsimp*
done

lemma *set-replicate-Suc*: *set* (*replicate* (*Suc n*) *x*) = $\{x\}$
by (*induct n*) *auto*

lemma *set-replicate* [*simp*]: $n \neq 0 \Longrightarrow set\ (replicate\ n\ x) = \{x\}$
by (*fast dest*!: *not0-implies-Suc intro*!: *set-replicate-Suc*)

lemma *set-replicate-conv-if*: *set* (*replicate n x*) = (*if n* = *0 then* {} *else* $\{x\}$)
by *auto*

lemma *in-set-replicateD*: $x : set\ (replicate\ n\ y) \Longrightarrow x = y$
by (*simp add*: *set-replicate-conv-if split*: *split-if-asm*)

lemma *replicate-append-same*:
replicate i x @ [*x*] = *x* # *replicate i x*
by (*induct i*) *simp-all*

lemma *map-replicate-trivial*:

map (λ*i*. *x*) [*0*..<*i*] = *replicate i x*
by (*induct i*) (*simp-all add*: *replicate-append-same*)

lemma *replicate-empty*[*simp*]: (*replicate n x* = []) ⟷ *n=0*
by (*induct n*) *auto*

lemma *empty-replicate*[*simp*]: ([] = *replicate n x*) ⟷ *n=0*
by (*induct n*) *auto*

lemma *replicate-eq-replicate*[*simp*]:
(*replicate m x* = *replicate n y*) ⟷ (*m=n* & (*m≠0* ⟶ *x=y*))
apply(*induct m arbitrary*: *n*)
apply *simp*
apply(*induct-tac n*)
apply *auto*
done

39.1.25 *rotate1* and *rotate*

lemma *rotate-simps*[*simp*]: *rotate1* [] = [] ∧ *rotate1* (*x*#*xs*) = *xs* @ [*x*]
by(*simp add*:*rotate1-def*)

lemma *rotate0*[*simp*]: *rotate 0* = *id*
by(*simp add*:*rotate-def*)

lemma *rotate-Suc*[*simp*]: *rotate* (*Suc n*) *xs* = *rotate1*(*rotate n xs*)
by(*simp add*:*rotate-def*)

lemma *rotate-add*:
rotate (*m*+*n*) = *rotate m o rotate n*
by(*simp add*:*rotate-def funpow-add*)

lemma *rotate-rotate*: *rotate m* (*rotate n xs*) = *rotate* (*m*+*n*) *xs*
by(*simp add*:*rotate-add*)

lemma *rotate1-rotate-swap*: *rotate1* (*rotate n xs*) = *rotate n* (*rotate1 xs*)
by(*simp add*:*rotate-def funpow-swap1*)

lemma *rotate1-length01*[*simp*]: *length xs* <= *1* ⟹ *rotate1 xs* = *xs*
by(*cases xs*) *simp-all*

lemma *rotate-length01*[*simp*]: *length xs* <= *1* ⟹ *rotate n xs* = *xs*
apply(*induct n*)
apply *simp*
apply (*simp add*:*rotate-def*)
done

lemma *rotate1-hd-tl*: *xs* ≠ [] ⟹ *rotate1 xs* = *tl xs* @ [*hd xs*]

by(*simp add*:*rotate1-def split*:*list.split*)

lemma *rotate-drop-take*:
  rotate n xs = *drop* (*n mod length xs*) *xs* @ *take* (*n mod length xs*) *xs*
apply(*induct n*)
 apply *simp*
apply(*simp add*:*rotate-def*)
apply(*cases xs* = [])
 apply (*simp*)
apply(*case-tac n mod length xs* = *0*)
 apply(*simp add*:*mod-Suc*)
 apply(*simp add*: *rotate1-hd-tl drop-Suc take-Suc*)
apply(*simp add*:*mod-Suc rotate1-hd-tl drop-Suc*[*symmetric*] *drop-tl*[*symmetric*]
            take-hd-drop linorder-not-le)
done

lemma *rotate-conv-mod*: *rotate n xs* = *rotate* (*n mod length xs*) *xs*
by(*simp add*:*rotate-drop-take*)

lemma *rotate-id*[*simp*]: *n mod length xs* = *0* ⟹ *rotate n xs* = *xs*
by(*simp add*:*rotate-drop-take*)

lemma *length-rotate1*[*simp*]: *length*(*rotate1 xs*) = *length xs*
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *length-rotate*[*simp*]: *length*(*rotate n xs*) = *length xs*
by (*induct n arbitrary*: *xs*) (*simp-all add*:*rotate-def*)

lemma *distinct1-rotate*[*simp*]: *distinct*(*rotate1 xs*) = *distinct xs*
by(*simp add*:*rotate1-def split*:*list.split*) *blast*

lemma *distinct-rotate*[*simp*]: *distinct*(*rotate n xs*) = *distinct xs*
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate-map*: *rotate n* (*map f xs*) = *map f* (*rotate n xs*)
by(*simp add*:*rotate-drop-take take-map drop-map*)

lemma *set-rotate1*[*simp*]: *set*(*rotate1 xs*) = *set xs*
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *set-rotate*[*simp*]: *set*(*rotate n xs*) = *set xs*
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate1-is-Nil-conv*[*simp*]: (*rotate1 xs* = []) = (*xs* = [])
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *rotate-is-Nil-conv*[*simp*]: (*rotate n xs* = []) = (*xs* = [])
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate-rev*:
  rotate n (rev xs) = rev(rotate (length xs − (n mod length xs)) xs)
apply(*simp add*:*rotate-drop-take rev-drop rev-take*)
apply(*cases length xs = 0*)
 apply *simp*
apply(*cases n mod length xs = 0*)
 apply *simp*
apply(*simp add*:*rotate-drop-take rev-drop rev-take*)
done

lemma *hd-rotate-conv-nth*: *xs ≠ [] ⟹ hd(rotate n xs) = xs!(n mod length xs)*
apply(*simp add*:*rotate-drop-take hd-append hd-drop-conv-nth hd-conv-nth*)
apply(*subgoal-tac length xs ≠ 0*)
 prefer *2* **apply** *simp*
using *mod-less-divisor*[*of length xs n*] **by** *arith*

39.1.26 *sublist* — a generalization of *nth* to sets

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

lemma *sublist-nil* [*simp*]: *sublist [] A = []*
by (*auto simp add*: *sublist-def*)

lemma *length-sublist*:
  length(*sublist xs I*) = *card*{*i*. *i* < *length xs* ∧ *i* : *I*}
by(*simp add*: *sublist-def length-filter-conv-card cong*:*conj-cong*)

lemma *sublist-shift-lemma-Suc*:
  map fst (*filter* (%*p*. *P*(*Suc*(*snd p*))) (*zip xs is*)) =
   map fst (*filter* (%*p*. *P*(*snd p*)) (*zip xs* (*map Suc is*)))
apply(*induct xs arbitrary*: *is*)
 apply *simp*
apply (*case-tac is*)
 apply *simp*
apply *simp*
done

lemma *sublist-shift-lemma*:
    map fst [*p*<−*zip xs* [*i*..<*i* + *length xs*] . *snd p* : *A*] =
     map fst [*p*<−*zip xs* [*0*..<*length xs*] . *snd p* + *i* : *A*]
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *add-commute*)

lemma *sublist-append*:
    sublist (*l* @ *l'*) *A* = *sublist l A* @ *sublist l'* {*j*. *j* + *length l* : *A*}
apply (*unfold sublist-def*)
apply (*induct l' rule*: *rev-induct*, *simp*)
apply (*simp add*: *upt-add-eq-append*[*of 0*] *zip-append sublist-shift-lemma*)
apply (*simp add*: *add-commute*)

done

lemma *sublist-Cons*:
sublist $(x \# l)$ A = (*if* 0:A *then* $[x]$ *else* $[]$) @ *sublist* l $\{j.\ Suc\ j : A\}$
apply (*induct* l *rule*: *rev-induct*)
apply (*simp add*: *sublist-def*)
apply (*simp del*: *append-Cons add*: *append-Cons*[*symmetric*] *sublist-append*)
done

lemma *set-sublist*: *set*(*sublist xs* I) = $\{xs!i|i.\ i<size\ xs \wedge i \in I\}$
apply(*induct xs arbitrary*: I)
apply(*auto simp*: *sublist-Cons nth-Cons split*:*nat.split dest*!: *gr0-implies-Suc*)
done

lemma *set-sublist-subset*: *set*(*sublist xs* I) $\subseteq$ *set xs*
by(*auto simp add*:*set-sublist*)

lemma *notin-set-sublistI*[*simp*]: $x \notin$ *set xs* $\Longrightarrow$ $x \notin$ *set*(*sublist xs* I)
by(*auto simp add*:*set-sublist*)

lemma *in-set-sublistD*: $x \in$ *set*(*sublist xs* I) $\Longrightarrow$ $x \in$ *set xs*
by(*auto simp add*:*set-sublist*)

lemma *sublist-singleton* [*simp*]: *sublist* $[x]$ A = (*if* 0 : A *then* $[x]$ *else* $[]$)
by (*simp add*: *sublist-Cons*)

lemma *distinct-sublistI*[*simp*]: *distinct xs* $\Longrightarrow$ *distinct*(*sublist xs* I)
apply(*induct xs arbitrary*: I)
apply *simp*
apply(*auto simp add*:*sublist-Cons*)
done

lemma *sublist-upt-eq-take* [*simp*]: *sublist* l $\{..<n\}$ = *take* n l
apply (*induct* l *rule*: *rev-induct*, *simp*)
apply (*simp split*: *nat-diff-split add*: *sublist-append*)
done

lemma *filter-in-sublist*:
distinct xs $\Longrightarrow$ *filter* $(\%x.\ x \in$ *set*(*sublist xs* s)) *xs* = *sublist xs* s
proof (*induct xs arbitrary*: s)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a xs*)
moreover hence !x. x: *set xs* $\longrightarrow$ $x \neq a$ **by** *auto*
ultimately show *?case* **by**(*simp add*: *sublist-Cons cong*:*filter-cong*)
qed

39.1.27 *splice*

lemma *splice-Nil2* [*simp*, *code*]:
splice xs [] = *xs*
by (*cases xs*) *simp-all*

lemma *splice-Cons-Cons* [*simp*, *code*]:
splice (*x*#*xs*) (*y*#*ys*) = *x* # *y* # *splice xs ys*
by *simp*

declare *splice.simps*(*2*) [*simp del*, *code del*]

lemma *length-splice*[*simp*]: *length*(*splice xs ys*) = *length xs* + *length ys*
apply(*induct xs arbitrary*: *ys*) **apply** *simp*
apply(*case-tac ys*)
apply *auto*
done

39.1.28 Infiniteness

lemma *finite-maxlen*:
finite (*M*::'*a list set*) ==> *EX n. ALL s*:*M. size s* < *n*
proof (*induct rule*: *finite.induct*)
case *emptyI* **show** *?case* **by** *simp*
next
case (*insertI M xs*)
then obtain *n* **where** ∀ *s*∈*M. length s* < *n* **by** *blast*
hence *ALL s*:*insert xs M. size s* < *max n* (*size xs*) + *1* **by** *auto*
thus *?case* **..**
qed

lemma *infinite-UNIV-listI*: ~ *finite*(*UNIV*::'*a list set*)
apply(*rule notI*)
apply(*drule finite-maxlen*)
apply (*metis UNIV-I length-replicate less-not-refl*)
done

39.2 Sorting

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not yet available. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

context *linorder*
begin

lemma *sorted-Cons*: *sorted* (*x*#*xs*) = (*sorted xs* & (*ALL y*:*set xs. x* <= *y*))
apply(*induct xs arbitrary*: *x*) **apply** *simp*

by *simp* (*blast intro*: *order-trans*)

lemma *sorted-append*:
sorted (*xs*@*ys*) = (*sorted xs* & *sorted ys* & (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *x*≤*y*))
by (*induct xs*) (*auto simp add*:*sorted-Cons*)

lemma *set-insort*: *set*(*insort x xs*) = *insert x* (*set xs*)
by (*induct xs*) *auto*

lemma *set-sort*[*simp*]: *set*(*sort xs*) = *set xs*
by (*induct xs*) (*simp-all add*:*set-insort*)

lemma *distinct-insort*: *distinct* (*insort x xs*) = (*x* ∉ *set xs* ∧ *distinct xs*)
by(*induct xs*)(*auto simp*:*set-insort*)

lemma *distinct-sort*[*simp*]: *distinct* (*sort xs*) = *distinct xs*
by(*induct xs*)(*simp-all add*:*distinct-insort set-sort*)

lemma *sorted-insort*: *sorted* (*insort x xs*) = *sorted xs*
apply (*induct xs*)
apply(*auto simp*:*sorted-Cons set-insort*)
done

theorem *sorted-sort*[*simp*]: *sorted* (*sort xs*)
by (*induct xs*) (*auto simp*:*sorted-insort*)

lemma *insort-is-Cons*: ∀ *x*∈*set xs*. *a* ≤ *x* ⟹ *insort a xs* = *a* # *xs*
by (*cases xs*) *auto*

lemma *sorted-remove1*: *sorted xs* ⟹ *sorted* (*remove1 a xs*)
by (*induct xs*, *auto simp add*: *sorted-Cons*)

lemma *insort-remove1*: ⟦ *a* ∈ *set xs*; *sorted xs* ⟧ ⟹ *insort a* (*remove1 a xs*) = *xs*
by (*induct xs*, *auto simp add*: *sorted-Cons insort-is-Cons*)

lemma *sorted-remdups*[*simp*]:
sorted l ⟹ *sorted* (*remdups l*)
by (*induct l*) (*auto simp*: *sorted-Cons*)

lemma *sorted-distinct-set-unique*:
assumes *sorted xs distinct xs sorted ys distinct ys set xs* = *set ys*
shows *xs* = *ys*
proof −
from *assms* **have** *1*: *length xs* = *length ys* **by** (*auto dest*!: *distinct-card*)
from *assms* **show** *?thesis*
proof(*induct rule*:*list-induct2*[*OF 1*])
case *1* **show** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** (*simp add*:*sorted-Cons*)

```
    (metis Diff-insert-absorb antisym insertE insert-iff)
  qed
qed

lemma finite-sorted-distinct-unique:
shows finite A ⟹ EX! xs. set xs = A & sorted xs & distinct xs
apply(drule finite-distinct-list)
apply clarify
apply(rule-tac a=sort xs in ex1I)
apply (auto simp: sorted-distinct-set-unique)
done

lemma sorted-take:
  sorted xs ⟹ sorted (take n xs)
proof (induct xs arbitrary: n rule: sorted.induct)
  case 1 show ?case by simp
next
  case 2 show ?case by (cases n) simp-all
next
  case (3 x y xs)
  then have x ≤ y by simp
  show ?case proof (cases n)
    case 0 then show ?thesis by simp
  next
    case (Suc m)
    with 3 have sorted (take m (y # xs)) by simp
    with Suc ‹x ≤ y› show ?thesis by (cases m) simp-all
  qed
qed

lemma sorted-drop:
  sorted xs ⟹ sorted (drop n xs)
proof (induct xs arbitrary: n rule: sorted.induct)
  case 1 show ?case by simp
next
  case 2 show ?case by (cases n) simp-all
next
  case 3 then show ?case by (cases n) simp-all
qed


end

lemma sorted-upt[simp]: sorted[i..<j]
by (induct j) (simp-all add:sorted-append)
```

39.2.1 *sorted-list-of-set*

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via *set*).

context *linorder*
begin

definition
sorted-list-of-set :: *'a set* ⇒ *'a list* **where**
[*code del*]: *sorted-list-of-set A* == *THE xs. set xs* = *A* & *sorted xs* & *distinct xs*

lemma *sorted-list-of-set*[*simp*]: *finite A* ⟹
set(*sorted-list-of-set A*) = *A* &
sorted(*sorted-list-of-set A*) & *distinct*(*sorted-list-of-set A*)
apply(*simp add*:*sorted-list-of-set-def*)
apply(*rule the1I2*)
apply(*simp-all add*: *finite-sorted-distinct-unique*)
done

lemma *sorted-list-of-empty*[*simp*]: *sorted-list-of-set* {} = []
unfolding *sorted-list-of-set-def*
apply(*subst the-equality*[*of* - []])
apply *simp-all*
done

end

39.2.2 *upto*: the generic interval-list

class *finite-intvl-succ* = *linorder* +
fixes *successor* :: *'a* ⇒ *'a*
assumes *finite-intvl*: *finite*{*a..b*}
and *successor-incr*: *a* < *successor a*
and *ord-discrete*: ¬(∃ *x*. *a* < *x* & *x* < *successor a*)

context *finite-intvl-succ*
begin

definition
upto :: *'a* ⇒ *'a* ⇒ *'a list* ((*1*[*-../-*])) **where**
upto i j == *sorted-list-of-set* {*i..j*}

lemma *upto*[*simp*]: *set*[*a..b*] = {*a..b*} & *sorted*[*a..b*] & *distinct*[*a..b*]
by(*simp add*:*upto-def finite-intvl*)

lemma *insert-intvl*: *i* ≤ *j* ⟹ *insert i* {*successor i..j*} = {*i..j*}
apply(*insert successor-incr*[*of i*])
apply(*auto simp*: *atLeastAtMost-def atLeast-def atMost-def*)

apply(*metis ord-discrete less-le not-le*)
done

lemma *sorted-list-of-set-rec*: $i \leq j \Longrightarrow$
  sorted-list-of-set $\{i..j\} = i \# $ *sorted-list-of-set* $\{$*successor* $i..j\}$
apply(*simp add*:*sorted-list-of-set-def upto-def*)
apply (*rule the1-equality*[*OF finite-sorted-distinct-unique*])
 apply (*simp add*:*finite-intvl*)
apply(*rule the1I2*[*OF finite-sorted-distinct-unique*])
 apply (*simp add*:*finite-intvl*)
apply (*simp add*: *sorted-Cons insert-intvl Ball-def*)
apply (*metis successor-incr leD less-imp-le order-trans*)
done

lemma *sorted-list-of-set-rec2*: $i \leq j \Longrightarrow$
  sorted-list-of-set $\{i..$*successor* $j\} =$
  sorted-list-of-set $\{i..j\}$ @ [*successor* j]
apply(*simp add*:*sorted-list-of-set-def upto-def*)
apply (*rule the1-equality*[*OF finite-sorted-distinct-unique*])
 apply (*simp add*:*finite-intvl*)
apply(*rule the1I2*[*OF finite-sorted-distinct-unique*])
 apply (*simp add*:*finite-intvl*)
apply (*simp add*: *sorted-append Ball-def expand-set-eq*)
apply(*rule conjI*)
apply (*metis eq-iff leD linear not-leE ord-discrete order-trans successor-incr*)
apply (*metis leD linear order-trans successor-incr*)
done

lemma *upto-rec*[*code*]: $[i..j] =$ (*if* $i \leq j$ *then* $i \# [$*successor* $i..j]$ *else* $[]$)
by(*simp add*: *upto-def sorted-list-of-set-rec*)

lemma *upto-empty*[*simp*]: $j < i \Longrightarrow [i..j] = []$
by(*simp add*: *upto-rec*)

lemma *upto-rec2*: $i \leq j \Longrightarrow [i..$*successor* $j] = [i..j]$ @ [*successor* j]
by(*simp add*: *upto-def sorted-list-of-set-rec2*)

end

The integers are an instance of the above class:

instantiation *int*:: *finite-intvl-succ*
begin

definition
successor-int-def: *successor* = (%*i*::*int*. *i*+*1*)

instance
by *intro-classes* (*simp-all add*: *successor-int-def*)

end

Now $[i..j]$ is defined for integers.

hide (**open**) *const successor*

lemma *upto-rec2-int*: $(i::int) \leq j \Longrightarrow [i..j+1] = [i..j] \ @ \ [j+1]$
by(*rule upto-rec2*[**where** *'a* = *int*, *simplified successor-int-def*])

39.2.3 *lists*: the list-forming operator over sets

inductive-set
  lists :: *'a set* => *'a list set*
  for *A* :: *'a set*
where
    Nil [*intro*!]: []: *lists A*
  | *Cons* [*intro*!,*noatp*]: [| *a*: *A*; *l*: *lists A*|] ==> *a#l* : *lists A*

inductive-cases *listsE* [*elim*!,*noatp*]: *x#l* : *lists A*
inductive-cases *listspE* [*elim*!,*noatp*]: *listsp A* (*x* # *l*)

lemma *listsp-mono* [*mono*]: *A* ≤ *B* ==> *listsp A* ≤ *listsp B*
by (*rule predicate1I*, *erule listsp.induct*, *blast*+)

lemmas *lists-mono* = *listsp-mono* [*to-set pred-subset-eq*]

lemma *listsp-infI*:
  assumes *l*: *listsp A l* **shows** *listsp B l* ==> *listsp* (*inf A B*) *l* **using** *l*
by *induct blast*+

lemmas *lists-IntI* = *listsp-infI* [*to-set*]

lemma *listsp-inf-eq* [*simp*]: *listsp* (*inf A B*) = *inf* (*listsp A*) (*listsp B*)
proof (*rule mono-inf* [**where** *f*=*listsp*, *THEN order-antisym*])
  show *mono listsp* **by** (*simp add*: *mono-def listsp-mono*)
  show *inf* (*listsp A*) (*listsp B*) ≤ *listsp* (*inf A B*) **by** (*blast intro*!: *listsp-infI predicate1I*)
qed

lemmas *listsp-conj-eq* [*simp*] = *listsp-inf-eq* [*simplified inf-fun-eq inf-bool-eq*]

lemmas *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set pred-equals-eq*]

lemma *append-in-listsp-conv* [*iff*]:
    (*listsp A* (*xs* @ *ys*)) = (*listsp A xs* ∧ *listsp A ys*)
by (*induct xs*) *auto*

lemmas *append-in-lists-conv* [*iff*] = *append-in-listsp-conv* [*to-set*]

lemma *in-listsp-conv-set*: (*listsp A xs*) = (∀ *x* ∈ *set xs*. *A x*)

— eliminate *listsp* in favour of *set*
by (*induct xs*) *auto*

lemmas *in-lists-conv-set* = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest*!,*noatp*]: *listsp A xs* ==> ∀ *x*∈*set xs*. *A x*
by (*rule in-listsp-conv-set* [*THEN iffD1*])

lemmas *in-listsD* [*dest*!,*noatp*] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro*!,*noatp*]: ∀ *x*∈*set xs*. *A x* ==> *listsp A xs*
by (*rule in-listsp-conv-set* [*THEN iffD2*])

lemmas *in-listsI* [*intro*!,*noatp*] = *in-listspI* [*to-set*]

lemma *lists-UNIV* [*simp*]: *lists UNIV* = *UNIV*
by *auto*

39.2.4 Inductive definition for membership

inductive *ListMem* :: *'a* ⇒ *'a list* ⇒ *bool*
where
  elem: *ListMem x* (*x* # *xs*)
| *insert*: *ListMem x xs* ⟹ *ListMem x* (*y* # *xs*)

lemma *ListMem-iff*: (*ListMem x xs*) = (*x* ∈ *set xs*)
apply (*rule iffI*)
apply (*induct set*: *ListMem*)
apply *auto*
apply (*induct xs*)
apply (*auto intro*: *ListMem.intros*)
done

39.2.5 Lists as Cartesian products

set-Cons A Xs: the set of lists with head drawn from *A* and tail drawn from *Xs*.

constdefs
set-Cons :: *'a set* ⇒ *'a list set* ⇒ *'a list set*
set-Cons A XS == {*z*. ∃ *x xs*. *z* = *x*#*xs* & *x* ∈ *A* & *xs* ∈ *XS*}
declare *set-Cons-def* [*code del*]

lemma *set-Cons-sing-Nil* [*simp*]: *set-Cons A* {[]} = (%*x*. [*x*])‘*A*
by (*auto simp add*: *set-Cons-def*)

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

consts *listset* :: *'a set list* ⇒ *'a list set*

primrec
  listset [] = {[]}
  listset(*A*#*As*) = *set-Cons A* (*listset As*)

39.3 Relations on Lists

39.3.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

consts *lexn* :: (*'a* * *'a*)*set* => *nat* => (*'a list* * *'a list*)*set*
      — The lexicographic ordering for lists of the specified length
primrec
  lexn r 0 = {}
  lexn r (*Suc n*) =
    (*prod-fun* (%(*x*,*xs*). *x*#*xs*) (%(*x*,*xs*). *x*#*xs*) ` (*r* <*lex*> *lexn r n*)) *Int*
    {(*xs*,*ys*). *length xs* = *Suc n* ∧ *length ys* = *Suc n*}

constdefs
  lex :: (*'a* × *'a*) *set* => (*'a list* × *'a list*) *set*
    lex r == ⋃ *n*. *lexn r n*
      — Holds only between lists of the same length

  lenlex :: (*'a* × *'a*) *set* => (*'a list* × *'a list*) *set*
    lenlex r == *inv-image* (*less-than* <*lex*> *lex r*) (%*xs*. (*length xs*, *xs*))
      — Compares lists by their length and then lexicographically

declare *lex-def* [*code del*]

lemma *wf-lexn*: *wf r* ==> *wf* (*lexn r n*)
apply (*induct n*, *simp*, *simp*)
apply(*rule wf-subset*)
 prefer *2* **apply** (*rule Int-lower1*)
apply(*rule wf-prod-fun-image*)
 prefer *2* **apply** (*rule inj-onI*, *auto*)
done

lemma *lexn-length*:
  (*xs*, *ys*) : *lexn r n* ==> *length xs* = *n* ∧ *length ys* = *n*
by (*induct n arbitrary*: *xs ys*) *auto*

lemma *wf-lex* [*intro*!]: *wf r* ==> *wf* (*lex r*)
apply (*unfold lex-def*)
apply (*rule wf-UN*)
apply (*blast intro*: *wf-lexn*, *clarify*)
apply (*rename-tac m n*)
apply (*subgoal-tac m* ≠ *n*)
 prefer *2* **apply** *blast*

apply (*blast dest*: *lexn-length not-sym*)
done

lemma *lexn-conv*:
  lexn r n =
    {(*xs*,*ys*). *length xs* = *n* ∧ *length ys* = *n* ∧
    (∃ *xys x y xs′ ys′*. *xs*= *xys* @ *x*#*xs′* ∧ *ys*= *xys* @ *y* # *ys′* ∧ (*x*, *y*):*r*)}
apply (*induct n*, *simp*)
apply (*simp add*: *image-Collect lex-prod-def*, *safe*, *blast*)
 apply (*rule-tac x* = *ab* # *xys* **in** *exI*, *simp*)
apply (*case-tac xys*, *simp-all*, *blast*)
done

lemma *lex-conv*:
  lex r =
    {(*xs*,*ys*). *length xs* = *length ys* ∧
    (∃ *xys x y xs′ ys′*. *xs* = *xys* @ *x* # *xs′* ∧ *ys* = *xys* @ *y* # *ys′* ∧ (*x*, *y*):*r*)}
by (*force simp add*: *lex-def lexn-conv*)

lemma *wf-lenlex* [*intro*!]: *wf r* ==> *wf* (*lenlex r*)
by (*unfold lenlex-def*) *blast*

lemma *lenlex-conv*:
    lenlex r = {(*xs*,*ys*). *length xs* < *length ys* |
                length xs = *length ys* ∧ (*xs*, *ys*) : *lex r*}
by (*simp add*: *lenlex-def Id-on-def lex-prod-def inv-image-def*)

lemma *Nil-notin-lex* [*iff*]: ([], *ys*) ∉ *lex r*
by (*simp add*: *lex-conv*)

lemma *Nil2-notin-lex* [*iff*]: (*xs*, []) ∉ *lex r*
by (*simp add*:*lex-conv*)

lemma *Cons-in-lex* [*simp*]:
    ((*x* # *xs*, *y* # *ys*) : *lex r*) =
      ((*x*, *y*) : *r* ∧ *length xs* = *length ys* | *x* = *y* ∧ (*xs*, *ys*) : *lex r*)
apply (*simp add*: *lex-conv*)
apply (*rule iffI*)
 prefer *2* **apply** (*blast intro*: *Cons-eq-appendI*, *clarify*)
apply (*case-tac xys*, *simp*, *simp*)
apply *blast*
done

39.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" ¡ "ab" ¡ "b". This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

constdefs
  lexord :: (*′a* ∗ *′a*)*set* ⇒ (*′a list* ∗ *′a list*) *set*

lexord r == {(*x*,*y*). ∃ *a v*. *y* = *x* @ *a* # *v* ∨
(∃ *u a b v w*. (*a*,*b*) ∈ *r* ∧ *x* = *u* @ (*a* # *v*) ∧ *y* = *u* @ (*b* # *w*))}
declare *lexord-def* [*code del*]

lemma *lexord-Nil-left*[*simp*]: ([],*y*) ∈ *lexord r* = (∃ *a x*. *y* = *a* # *x*)
by (*unfold lexord-def*, *induct-tac y*, *auto*)

lemma *lexord-Nil-right*[*simp*]: (*x*,[]) ∉ *lexord r*
by (*unfold lexord-def*, *induct-tac x*, *auto*)

lemma *lexord-cons-cons*[*simp*]:
((*a* # *x*, *b* # *y*) ∈ *lexord r*) = ((*a*,*b*)∈ *r* | (*a* = *b* & (*x*,*y*)∈ *lexord r*))
apply (*unfold lexord-def*, *safe*, *simp-all*)
apply (*case-tac u*, *simp*, *simp*)
apply (*case-tac u*, *simp*, *clarsimp*, *blast*, *blast*, *clarsimp*)
apply (*erule-tac x*=*b* # *u* **in** *allE*)
by *force*

lemmas *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

lemma *lexord-append-rightI*: ∃ *b z*. *y* = *b* # *z* ⟹ (*x*, *x* @ *y*) ∈ *lexord r*
by (*induct-tac x*, *auto*)

lemma *lexord-append-left-rightI*:
(*a*,*b*) ∈ *r* ⟹ (*u* @ *a* # *x*, *u* @ *b* # *y*) ∈ *lexord r*
by (*induct-tac u*, *auto*)

lemma *lexord-append-leftI*: (*u*,*v*) ∈ *lexord r* ⟹ (*x* @ *u*, *x* @ *v*) ∈ *lexord r*
by (*induct x*, *auto*)

lemma *lexord-append-leftD*:
⟦ (*x* @ *u*, *x* @ *v*) ∈ *lexord r*; (! *a*. (*a*,*a*) ∉ *r*) ⟧ ⟹ (*u*,*v*) ∈ *lexord r*
by (*erule rev-mp*, *induct-tac x*, *auto*)

lemma *lexord-take-index-conv*:
((*x*,*y*) : *lexord r*) =
((*length x* < *length y* ∧ *take* (*length x*) *y* = *x*) ∨
(∃ *i*. *i* < *min*(*length x*)(*length y*) & *take i x* = *take i y* & (*x*!*i*,*y*!*i*) ∈ *r*))
apply (*unfold lexord-def Let-def*, *clarsimp*)
apply (*rule-tac f* = (% *a b*. *a* ∨ *b*) **in** *arg-cong2*)
apply *auto*
apply (*rule-tac x*=*hd* (*drop* (*length x*) *y*) **in** *exI*)
apply (*rule-tac x*=*tl* (*drop* (*length x*) *y*) **in** *exI*)
apply (*erule subst*, *simp add*: *min-def*)
apply (*rule-tac x* =*length u* **in** *exI*, *simp*)
apply (*rule-tac x* =*take i x* **in** *exI*)
apply (*rule-tac x* =*x* ! *i* **in** *exI*)
apply (*rule-tac x* =*y* ! *i* **in** *exI*, *safe*)
apply (*rule-tac x*=*drop* (*Suc i*) *x* **in** *exI*)

apply (*drule sym*, *simp add*: *drop-Suc-conv-tl*)
apply (*rule-tac x*=*drop* (*Suc i*) *y* **in** *exI*)
by (*simp add*: *drop-Suc-conv-tl*)

— lexord is extension of partial ordering List.lex
lemma *lexord-lex*: (*x*,*y*) ∈ *lex r* = ((*x*,*y*) ∈ *lexord r* ∧ *length x* = *length y*)
apply (*rule-tac x* = *y* **in** *spec*)
apply (*induct-tac x*, *clarsimp*)
by (*clarify*, *case-tac x*, *simp*, *force*)

lemma *lexord-irreflexive*: (! *x*. (*x*,*x*) ∉ *r*) ⟹ (*y*,*y*) ∉ *lexord r*
by (*induct y*, *auto*)

lemma *lexord-trans*:
⟦ (*x*, *y*) ∈ *lexord r*; (*y*, *z*) ∈ *lexord r*; *trans r* ⟧ ⟹ (*x*, *z*) ∈ *lexord r*
apply (*erule rev-mp*)+
apply (*rule-tac x* = *x* **in** *spec*)
apply (*rule-tac x* = *z* **in** *spec*)
apply (*induct-tac y*, *simp*, *clarify*)
apply (*case-tac xa*, *erule ssubst*)
apply (*erule allE*, *erule allE*) — avoid simp recursion
apply (*case-tac x*, *simp*, *simp*)
apply (*case-tac x*, *erule allE*, *erule allE*, *simp*)
apply (*erule-tac x* = *listb* **in** *allE*)
apply (*erule-tac x* = *lista* **in** *allE*, *simp*)
apply (*unfold trans-def*)
by *blast*

lemma *lexord-transI*: *trans r* ⟹ *trans* (*lexord r*)
by (*rule transI*, *drule lexord-trans*, *blast*)

lemma *lexord-linear*: (! *a b*. (*a*,*b*)∈ *r* | *a* = *b* | (*b*,*a*) ∈ *r*) ⟹ (*x*,*y*) : *lexord r* | *x* = *y* | (*y*,*x*) : *lexord r*
apply (*rule-tac x* = *y* **in** *spec*)
apply (*induct-tac x*, *rule allI*)
apply (*case-tac x*, *simp*, *simp*)
apply (*rule allI*, *case-tac x*, *simp*, *simp*)
by *blast*

39.4 Lexicographic combination of measure functions

These are useful for termination proofs

definition
measures fs = *inv-image* (*lex less-than*) (%*a*. *map* (%*f*. *f a*) *fs*)

lemma *wf-measures*[*recdef-wf*, *simp*]: *wf* (*measures fs*)
unfolding *measures-def*
by *blast*

lemma *in-measures*[*simp*]:
(*x*, *y*) ∈ *measures* [] = *False*
(*x*, *y*) ∈ *measures* (*f* # *fs*)
= (*f x* < *f y* ∨ (*f x* = *f y* ∧ (*x*, *y*) ∈ *measures fs*))
unfolding *measures-def*
by *auto*

lemma *measures-less*: *f x* < *f y* ==> (*x*, *y*) ∈ *measures* (*f*#*fs*)
by *simp*

lemma *measures-lesseq*: *f x* <= *f y* ==> (*x*, *y*) ∈ *measures fs* ==> (*x*, *y*) ∈ *measures* (*f*#*fs*)
by *auto*

39.4.1 Lifting a Relation on List Elements to the Lists

inductive-set
listrel :: ('*a* ∗ '*a*)*set* => ('*a list* ∗ '*a list*)*set*
for *r* :: ('*a* ∗ '*a*)*set*
where
Nil: ([],[]) ∈ *listrel r*
| *Cons*: [| (*x*,*y*) ∈ *r*; (*xs*,*ys*) ∈ *listrel r* |] ==> (*x*#*xs*, *y*#*ys*) ∈ *listrel r*

inductive-cases *listrel-Nil1* [*elim*!]: ([],*xs*) ∈ *listrel r*
inductive-cases *listrel-Nil2* [*elim*!]: (*xs*,[]) ∈ *listrel r*
inductive-cases *listrel-Cons1* [*elim*!]: (*y*#*ys*,*xs*) ∈ *listrel r*
inductive-cases *listrel-Cons2* [*elim*!]: (*xs*,*y*#*ys*) ∈ *listrel r*

lemma *listrel-mono*: *r* ⊆ *s* ⟹ *listrel r* ⊆ *listrel s*
apply *clarify*
apply (*erule listrel.induct*)
apply (*blast intro*: *listrel.intros*)+
done

lemma *listrel-subset*: *r* ⊆ *A* × *A* ⟹ *listrel r* ⊆ *lists A* × *lists A*
apply *clarify*
apply (*erule listrel.induct*, *auto*)
done

lemma *listrel-refl-on*: *refl-on A r* ⟹ *refl-on* (*lists A*) (*listrel r*)
apply (*simp add*: *refl-on-def listrel-subset Ball-def*)
apply (*rule allI*)
apply (*induct-tac x*)
apply (*auto intro*: *listrel.intros*)
done

lemma *listrel-sym*: *sym r* ⟹ *sym* (*listrel r*)
apply (*auto simp add*: *sym-def*)

apply (*erule listrel.induct*)
apply (*blast intro*: *listrel.intros*)+
done

lemma *listrel-trans*: *trans r* ⟹ *trans* (*listrel r*)
apply (*simp add*: *trans-def*)
apply (*intro allI*)
apply (*rule impI*)
apply (*erule listrel.induct*)
apply (*blast intro*: *listrel.intros*)+
done

theorem *equiv-listrel*: *equiv A r* ⟹ *equiv* (*lists A*) (*listrel r*)
by (*simp add*: *equiv-def listrel-refl-on listrel-sym listrel-trans*)

lemma *listrel-Nil* [*simp*]: *listrel r* `` {[]} = {[]}
by (*blast intro*: *listrel.intros*)

lemma *listrel-Cons*:
  listrel r `` {*x*#*xs*} = *set-Cons* (*r*``{*x*}) (*listrel r* `` {*xs*})
by (*auto simp add*: *set-Cons-def intro*: *listrel.intros*)

39.5 Miscellany

39.5.1 Characters and strings

datatype *nibble* =
  Nibble0 | *Nibble1* | *Nibble2* | *Nibble3* | *Nibble4* | *Nibble5* | *Nibble6* | *Nibble7*
| *Nibble8* | *Nibble9* | *NibbleA* | *NibbleB* | *NibbleC* | *NibbleD* | *NibbleE* | *NibbleF*

lemma *UNIV-nibble*:
 UNIV = {*Nibble0*, *Nibble1*, *Nibble2*, *Nibble3*, *Nibble4*, *Nibble5*, *Nibble6*, *Nibble7*,
  Nibble8, *Nibble9*, *NibbleA*, *NibbleB*, *NibbleC*, *NibbleD*, *NibbleE*, *NibbleF*} (**is** - = *?A*)
proof (*rule UNIV-eq-I*)
 fix *x* **show** *x* ∈ *?A* **by** (*cases x*) *simp-all*
qed

instance *nibble* :: *finite*
 by *default* (*simp add*: *UNIV-nibble*)

datatype *char* = *Char nibble nibble*
 — Note: canonical order of character encoding coincides with standard term ordering

lemma *UNIV-char*:
 UNIV = *image* (*split Char*) (*UNIV* × *UNIV*)
proof (*rule UNIV-eq-I*)
 fix *x* **show** *x* ∈ *image* (*split Char*) (*UNIV* × *UNIV*) **by** (*cases x*) *auto*
qed

instance *char* :: *finite*
by *default* (*simp add*: *UNIV-char*)

lemma *size-char* [*code*, *simp*]:
size (*c*::*char*) = *0* **by** (*cases c*) *simp*

lemma *char-size* [*code*, *simp*]:
char-size (*c*::*char*) = *0* **by** (*cases c*) *simp*

primrec *nibble-pair-of-char* :: *char* ⇒ *nibble* × *nibble* **where**
nibble-pair-of-char (*Char n m*) = (*n*, *m*)

declare *nibble-pair-of-char.simps* [*code del*]

setup ⟪
let
val nibbles = *map* (*Thm.cterm-of* @{*theory*} *o HOLogic.mk-nibble*) (*0 upto 15*);
val thms = *map-product*
(*fn n => fn m => Drule.instantiate'* [] [*SOME n*, *SOME m*] @{*thm nibble-pair-of-char.simps*})
nibbles nibbles;
in
PureThy.note-thmss Thm.lemmaK [((*Binding.name nibble-pair-of-char-simps*,
[]), [(*thms*, [])])]
#−> (*fn* [(-, *thms*)] => *fold-rev Code.add-eqn thms*)
end
⟫

lemma *char-case-nibble-pair* [*code*, *code inline*]:
char-case f = *split f o nibble-pair-of-char*
by (*simp add*: *expand-fun-eq split*: *char.split*)

lemma *char-rec-nibble-pair* [*code*, *code inline*]:
char-rec f = *split f o nibble-pair-of-char*
unfolding *char-case-nibble-pair* [*symmetric*]
by (*simp add*: *expand-fun-eq split*: *char.split*)

types *string* = *char list*

syntax
-*Char* :: *xstr* => *char* (*CHR* -)
-*String* :: *xstr* => *string* (-)

setup *StringSyntax.setup*

39.6 Size function

lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (*list-size f*)
by (*rule is-measure-trivial*)

lemma [*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* (*option-size f*)
by (*rule is-measure-trivial*)

lemma *list-size-estimation*[*termination-simp*]:
x $\in$ *set xs* $\Longrightarrow$ *y* $<$ *f x* $\Longrightarrow$ *y* $<$ *list-size f xs*
by (*induct xs*) *auto*

lemma *list-size-estimation'*[*termination-simp*]:
x $\in$ *set xs* $\Longrightarrow$ *y* $\leq$ *f x* $\Longrightarrow$ *y* $\leq$ *list-size f xs*
by (*induct xs*) *auto*

lemma *list-size-map*[*simp*]: *list-size f* (*map g xs*) = *list-size* (*f o g*) *xs*
by (*induct xs*) *auto*

lemma *list-size-pointwise*[*termination-simp*]:
($\bigwedge$*x*. *x* $\in$ *set xs* $\Longrightarrow$ *f x* $<$ *g x*) $\Longrightarrow$ *list-size f xs* $\leq$ *list-size g xs*
by (*induct xs*) *force+*

39.7 Code generator

39.7.1 Setup

types-code
list (- *list*)
attach (*term-of*) ⟪
fun term-of-list f T = HOLogic.mk-list T o map f;
⟫
attach (*test*) ⟪
fun gen-list' aG aT i j = frequency
[(*i*, *fn* () =>
let
val (*x*, *t*) = *aG j*;
val (*xs*, *ts*) = *gen-list' aG aT* (*i*−*1*) *j*
in (*x* :: *xs*, *fn* () => *HOLogic.cons-const aT* $ *t* () $ *ts* ()) *end*),
(*1*, *fn* () => ([], *fn* () => *HOLogic.nil-const aT*))] ()
and gen-list aG aT i = gen-list' aG aT i i;
⟫
char (*string*)
attach (*term-of*) ⟪
val term-of-char = HOLogic.mk-char o ord;
⟫
attach (*test*) ⟪
fun gen-char i =
let val j = random-range (*ord a*) (*Int.min* (*ord a* + *i*, *ord z*))
in (*chr j*, *fn* () => *HOLogic.mk-char j*) *end;*
⟫

consts-code *Cons* ((- ::/ -))

code-type *list*
(*SML* - *list*)
(*OCaml* - *list*)
(*Haskell* ![-])

code-reserved *SML*
list

code-reserved *OCaml*
list

code-const *Nil*
(*SML* [])
(*OCaml* [])
(*Haskell* [])

ML ⟪
local

open Basic-Code-Thingol;

fun implode-list naming t = case pairself
(*Code-Thingol.lookup-const naming*) (@{*const-name Nil*}, @{*const-name Cons*})
of (*SOME nil′*, *SOME cons′*) => *let*
fun dest-cons (*IConst* (*c*, -) `$ *t1* `$ *t2*) =
if c = cons′
then SOME (*t1*, *t2*)
else NONE
| *dest-cons* - = *NONE*;
val (*ts*, *t′*) = *Code-Thingol.unfoldr dest-cons t*;
in case t′
of IConst (*c*, -) => *if c = nil′ then SOME ts else NONE*
| - => *NONE*
end
| - => *NONE*

fun decode-char naming (*IConst* (*c1*, -), *IConst* (*c2*, -)) = (*case map-filter*
(*Code-Thingol.lookup-const naming*)[@{*const-name Nibble0*}, @{*const-name Nibble1*},
@{*const-name Nibble2*}, @{*const-name Nibble3*},
@{*const-name Nibble4*}, @{*const-name Nibble5*},
@{*const-name Nibble6*}, @{*const-name Nibble7*},
@{*const-name Nibble8*}, @{*const-name Nibble9*},
@{*const-name NibbleA*}, @{*const-name NibbleB*},
@{*const-name NibbleC*}, @{*const-name NibbleD*},
@{*const-name NibbleE*}, @{*const-name NibbleF*}]
of nibbles′ as [-, -, -, -, -, -, -, -, -, -, -, -, -, -, -, -] => *let*
fun idx c = find-index (*curry* (*op* =) *c*) *nibbles′*;
fun decode ~1 - = *NONE*

```
          | decode - ~1 = NONE
          | decode n m = SOME (chr (n * 16 + m));
      in decode (idx c1) (idx c2) end
  | - => NONE)
| decode-char - - = NONE

fun implode-string naming mk-char mk-string ts = case
  Code-Thingol.lookup-const naming @{const-name Char}
   of SOME char' => let
       fun implode-char (IConst (c, -) `$ t1 `$ t2) =
             if c = char' then decode-char naming (t1, t2) else NONE
         | implode-char - = NONE;
       val ts' = map implode-char ts;
      in if forall is-some ts'
       then (SOME o Code-Printer.str o mk-string o implode o map-filter I) ts'
       else NONE
      end
   | - => NONE;

fun default-list (target-fxy, target-cons) pr fxy t1 t2 =
  Code-Printer.brackify-infix (target-fxy, Code-Printer.R) fxy [
    pr (Code-Printer.INFX (target-fxy, Code-Printer.X)) t1,
    Code-Printer.str target-cons,
    pr (Code-Printer.INFX (target-fxy, Code-Printer.R)) t2
  ];

fun pretty-list literals =
  let
    val mk-list = Code-Printer.literal-list literals;
    fun pretty pr naming thm vars fxy [(t1, -), (t2, -)] =
      case Option.map (cons t1) (implode-list naming t2)
       of SOME ts => mk-list (map (pr vars Code-Printer.NOBR) ts)
        | NONE => default-list (Code-Printer.infix-cons literals) (pr vars) fxy t1
t2;
  in (2, pretty) end;

fun pretty-list-string literals =
  let
    val mk-list = Code-Printer.literal-list literals;
    val mk-char = Code-Printer.literal-char literals;
    val mk-string = Code-Printer.literal-string literals;
    fun pretty pr naming thm vars fxy [(t1, -), (t2, -)] =
      case Option.map (cons t1) (implode-list naming t2)
       of SOME ts => (case implode-string naming mk-char mk-string ts
           of SOME p => p
            | NONE => mk-list (map (pr vars Code-Printer.NOBR) ts))
        | NONE => default-list (Code-Printer.infix-cons literals) (pr vars) fxy t1
t2;
  in (2, pretty) end;
```

```
fun pretty-char literals =
  let
    val mk-char = Code-Printer.literal-char literals;
    fun pretty - naming thm - - [(t1, -), (t2, -)] =
      case decode-char naming (t1, t2)
       of SOME c => (Code-Printer.str o mk-char) c
        | NONE => Code-Printer.nerror thm Illegal character expression;
  in (2, pretty) end;

fun pretty-message literals =
  let
    val mk-char = Code-Printer.literal-char literals;
    val mk-string = Code-Printer.literal-string literals;
    fun pretty - naming thm - - [(t, -)] =
      case implode-list naming t
       of SOME ts => (case implode-string naming mk-char mk-string ts
           of SOME p => p
            | NONE => Code-Printer.nerror thm Illegal message expression)
        | NONE => Code-Printer.nerror thm Illegal message expression;
  in (1, pretty) end;

in

fun add-literal-list target thy =
  let
    val pr = pretty-list (Code-Target.the-literals thy target);
  in
    thy
    |> Code-Target.add-syntax-const target @{const-name Cons} (SOME pr)
  end;

fun add-literal-list-string target thy =
  let
    val pr = pretty-list-string (Code-Target.the-literals thy target);
  in
    thy
    |> Code-Target.add-syntax-const target @{const-name Cons} (SOME pr)
  end;

fun add-literal-char target thy =
  let
    val pr = pretty-char (Code-Target.the-literals thy target);
  in
    thy
    |> Code-Target.add-syntax-const target @{const-name Char} (SOME pr)
  end;

fun add-literal-message str target thy =
```

```
  let
    val pr = pretty-message (Code-Target.the-literals thy target);
  in
    thy
    |> Code-Target.add-syntax-const target str (SOME pr)
  end;

end;
⟩⟩

setup ⟨⟨
  fold (fn target => add-literal-list target) [SML, OCaml, Haskell]
⟩⟩

code-instance list :: eq
  (Haskell -)

code-const eq-class.eq :: 'a::eq list ⇒ 'a list ⇒ bool
  (Haskell infixl 4 ==)

setup ⟨⟨
let

fun list-codegen thy defs dep thyname b t gr =
  let
    val ts = HOLogic.dest-list t;
    val (-, gr') = Codegen.invoke-tycodegen thy defs dep thyname false
      (fastype-of t) gr;
    val (ps, gr'') = fold-map
      (Codegen.invoke-codegen thy defs dep thyname false) ts gr'
  in SOME (Pretty.list [ ] ps, gr'') end handle TERM - => NONE;

fun char-codegen thy defs dep thyname b t gr =
  let
    val i = HOLogic.dest-char t;
    val (-, gr') = Codegen.invoke-tycodegen thy defs dep thyname false
      (fastype-of t) gr;
  in SOME (Codegen.str (ML-Syntax.print-string (chr i)), gr')
  end handle TERM - => NONE;

in
  Codegen.add-codegen list-codegen list-codegen
  #> Codegen.add-codegen char-codegen char-codegen
end;
⟩⟩
```

39.7.2 Generation of efficient code

primrec

```
  member :: 'a ⇒ 'a list ⇒ bool (infixl mem 55)
where
  x mem [] ⟷ False
| x mem (y#ys) ⟷ x = y ∨ x mem ys

primrec
  null:: 'a list ⇒ bool
where
  null [] = True
| null (x#xs) = False

primrec
  list-inter :: 'a list ⇒ 'a list ⇒ 'a list
where
  list-inter [] bs = []
| list-inter (a#as) bs =
    (if a ∈ set bs then a # list-inter as bs else list-inter as bs)

primrec
  list-all :: ('a ⇒ bool) ⇒ ('a list ⇒ bool)
where
  list-all P [] = True
| list-all P (x#xs) = (P x ∧ list-all P xs)

primrec
  list-ex :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
where
  list-ex P [] = False
| list-ex P (x#xs) = (P x ∨ list-ex P xs)

primrec
  filtermap :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list
where
  filtermap f [] = []
| filtermap f (x#xs) =
    (case f x of None ⇒ filtermap f xs
     | Some y ⇒ y # filtermap f xs)

primrec
  map-filter :: ('a ⇒ 'b) ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ 'b list
where
  map-filter f P [] = []
| map-filter f P (x#xs) =
    (if P x then f x # map-filter f P xs else map-filter f P xs)

primrec
  length-unique :: 'a list ⇒ nat
where
  length-unique [] = 0
```

```
| length-unique (x#xs) =
    (if x ∈ set xs then length-unique xs else Suc (length-unique xs))
```

Only use *mem* for generating executable code. Otherwise use $x \in set\ xs$ instead — it is much easier to reason about. The same is true for *list-all* and *list-ex*: write $\forall x{\in}set\ xs$ and $\exists x{\in}set\ xs$ instead because the HOL quantifiers are aleady known to the automatic provers. In fact, the declarations in the code subsection make sure that $\in$, $\forall x{\in}set\ xs$ and $\exists x{\in}set\ xs$ are implemented efficiently.

Efficient emptyness check is implemented by *null*.

The functions *filtermap* and *map-filter* are just there to generate efficient code. Do not use them for modelling and proving.

```
lemma rev-foldl-cons [code]:
  rev xs = foldl (λxs x. x # xs) [] xs
proof (induct xs)
  case Nil then show ?case by simp
next
  case Cons
  {
    fix x xs ys
    have foldl (λxs x. x # xs) ys xs @ [x]
      = foldl (λxs x. x # xs) (ys @ [x]) xs
    by (induct xs arbitrary: ys) auto
  }
  note aux = this
  show ?case by (induct xs) (auto simp add: Cons aux)
qed

lemma mem-iff [code post]:
  x mem xs ⟷ x ∈ set xs
by (induct xs) auto

lemmas in-set-code [code unfold] = mem-iff [symmetric]

lemma empty-null [code inline]:
  xs = [] ⟷ null xs
by (cases xs) simp-all

lemmas null-empty [code post] =
  empty-null [symmetric]

lemma list-inter-conv:
  set (list-inter xs ys) = set xs ∩ set ys
by (induct xs) auto

lemma list-all-iff [code post]:
  list-all P xs ⟷ (∀ x ∈ set xs. P x)
by (induct xs) auto
```

lemmas *list-ball-code* [*code unfold*] = *list-all-iff* [*symmetric*]

lemma *list-all-append* [*simp*]:
list-all P (*xs* @ *ys*) ⟷ (*list-all P xs* ∧ *list-all P ys*)
by (*induct xs*) *auto*

lemma *list-all-rev* [*simp*]:
list-all P (*rev xs*) ⟷ *list-all P xs*
by (*simp add*: *list-all-iff*)

lemma *list-all-length*:
list-all P xs ⟷ (∀ *n* < *length xs*. *P* (*xs* ! *n*))
unfolding *list-all-iff* **by** (*auto intro*: *all-nth-imp-all-set*)

lemma *list-ex-iff* [*code post*]:
list-ex P xs ⟷ (∃ *x* ∈ *set xs*. *P x*)
by (*induct xs*) *simp-all*

lemmas *list-bex-code* [*code unfold*] =
list-ex-iff [*symmetric*]

lemma *list-ex-length*:
list-ex P xs ⟷ (∃ *n* < *length xs*. *P* (*xs* ! *n*))
unfolding *list-ex-iff set-conv-nth* **by** *auto*

lemma *filtermap-conv*:
filtermap f xs = *map* (λ*x*. *the* (*f x*)) (*filter* (λ*x*. *f x* ≠ *None*) *xs*)
by (*induct xs*) (*simp-all split*: *option.split*)

lemma *map-filter-conv* [*simp*]:
map-filter f P xs = *map f* (*filter P xs*)
by (*induct xs*) *auto*

lemma *length-remdups-length-unique* [*code inline*]:
length (*remdups xs*) = *length-unique xs*
by (*induct xs*) *simp-all*

hide (**open**) *const length-unique*

Code for bounded quantification and summation over nats.

lemma *atMost-upto* [*code unfold*]:
{..*n*} = *set* [*0*..<*Suc n*]
by *auto*

lemma *atLeast-upt* [*code unfold*]:
{..<*n*} = *set* [*0*..<*n*]
by *auto*

lemma *greaterThanLessThan-upt* [*code unfold*]:
$\{n<..<m\} = set\ [Suc\ n..<m]$
by *auto*

lemma *atLeastLessThan-upt* [*code unfold*]:
$\{n..<m\} = set\ [n..<m]$
by *auto*

lemma *greaterThanAtMost-upt* [*code unfold*]:
$\{n<..m\} = set\ [Suc\ n..<Suc\ m]$
by *auto*

lemma *atLeastAtMost-upt* [*code unfold*]:
$\{n..m\} = set\ [n..<Suc\ m]$
by *auto*

lemma *all-nat-less-eq* [*code unfold*]:
$(\forall\, m<n::nat.\ P\ m) \longleftrightarrow (\forall\, m \in \{0..<n\}.\ P\ m)$
by *auto*

lemma *ex-nat-less-eq* [*code unfold*]:
$(\exists\, m<n::nat.\ P\ m) \longleftrightarrow (\exists\, m \in \{0..<n\}.\ P\ m)$
by *auto*

lemma *all-nat-less* [*code unfold*]:
$(\forall\, m\leq n::nat.\ P\ m) \longleftrightarrow (\forall\, m \in \{0..n\}.\ P\ m)$
by *auto*

lemma *ex-nat-less* [*code unfold*]:
$(\exists\, m\leq n::nat.\ P\ m) \longleftrightarrow (\exists\, m \in \{0..n\}.\ P\ m)$
by *auto*

lemma *setsum-set-distinct-conv-listsum*:
$distinct\ xs \Longrightarrow setsum\ f\ (set\ xs) = listsum\ (map\ f\ xs)$
by (*induct xs*) *simp-all*

lemma *setsum-set-upt-conv-listsum* [*code unfold*]:
$setsum\ f\ (set\ [m..<n]) = listsum\ (map\ f\ [m..<n])$
by (*rule setsum-set-distinct-conv-listsum*) *simp*

Code for summation over ints.

lemma *greaterThanLessThan-upto* [*code unfold*]:
$\{i<..<j::int\} = set\ [i+1..j - 1]$
by *auto*

lemma *atLeastLessThan-upto* [*code unfold*]:
$\{i..<j::int\} = set\ [i..j - 1]$
by *auto*

lemma *greaterThanAtMost-upto* [*code unfold*]:
  {*i*<..*j*::*int*} = *set* [*i*+1..*j*]
by *auto*

lemma *atLeastAtMost-upto* [*code unfold*]:
  {*i*..*j*::*int*} = *set* [*i*..*j*]
by *auto*

lemma *setsum-set-upto-conv-listsum* [*code unfold*]:
  setsum f (*set* [*i*..*j*::*int*]) = *listsum* (*map f* [*i*..*j*])
by (*rule setsum-set-distinct-conv-listsum*) *simp*

end

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

theory *Code-Message*
imports *Plain* ~~/*src*/*HOL*/*List*
begin

40.1 Datatype of messages

datatype *message-string* = *STR string*

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

lemma [*code*]: *size* (*s*::*message-string*) = *0*
  by (*cases s*) *simp-all*

lemma [*code*]: *message-string-size* (*s*::*message-string*) = *0*
  by (*cases s*) *simp-all*

40.2 ML interface

ML ⟪
structure Message-String =
struct

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

end;
⟫

40.3 Code serialization

code-type *message-string*
  (*SML string*)

```
  (OCaml string)
  (Haskell String)

setup ⟪
  fold (fn target => add-literal-message @{const-name STR} target)
    [SML, OCaml, Haskell]
⟫

code-reserved SML string
code-reserved OCaml string

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

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

end
```

41 Typerep: Reflecting Pure types into HOL

```
theory Typerep
imports Plain List Code-Message
begin

datatype typerep = Typerep message-string typerep list

class typerep =
  fixes typerep :: 'a::{} itself ⇒ typerep
begin

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

end

setup ⟪
let
  fun typerep-tr (∗-TYPEREP∗) [ty] =
        Lexicon.const @{const-syntax typerep} $ (Lexicon.const -constrain $ Lexi-
con.const TYPE $
          (Lexicon.const itself $ ty))
    | typerep-tr (∗-TYPEREP∗) ts = raise TERM (typerep-tr, ts);
  fun typerep-tr' show-sorts (∗typerep∗)
          (Type (fun, [Type (itself, [T]), -])) (Const (@{const-syntax TYPE}, -) ::
ts) =
```

```
      Term.list-comb (Lexicon.const -TYPEREP $ Syntax.term-of-typ show-sorts
T, ts)
    | typerep-tr' - T ts = raise Match;
in
  Sign.add-syntax-i
   [(-TYPEREP, SimpleSyntax.read-typ type => logic, Delimfix (1TYPEREP/(1'(-'))))]
  #> Sign.add-trfuns ([], [(-TYPEREP, typerep-tr)], [], [])
  #> Sign.add-trfunsT [(@{const-syntax typerep}, typerep-tr')]
end
⟩⟩
```

ML ⟨⟨

```
structure Typerep =
struct

fun mk f (Type (tyco, tys)) =
      @{term Typerep} $ Message-String.mk tyco
        $ HOLogic.mk-list @{typ typerep} (map (mk f) tys)
  | mk f (TFree v) =
      f v;

fun typerep ty =
  Const (@{const-name typerep}, Term.itselfT ty --> @{typ typerep})
    $ Logic.mk-type ty;

fun add-def tyco thy =
  let
    val sorts = replicate (Sign.arity-number thy tyco) @{sort typerep};
    val vs = Name.names Name.context 'a sorts;
    val ty = Type (tyco, map TFree vs);
   val lhs = Const (@{const-name typerep}, Term.itselfT ty --> @{typ typerep})
      $ Free (T, Term.itselfT ty);
    val rhs = mk (typerep o TFree) ty;
    val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
  in
    thy
    |> TheoryTarget.instantiation ([tyco], vs, @{sort typerep})
    |> '(fn lthy => Syntax.check-term lthy eq)
    |-> (fn eq => Specification.definition (NONE, (Attrib.empty-binding, eq)))
    |> snd
    |> Class.prove-instantiation-instance (K (Class.intro-classes-tac []))
    |> LocalTheory.exit-global
  end;

fun perhaps-add-def tyco thy =
  let
    val inst = can (Sorts.mg-domain (Sign.classes-of thy) tyco) @{sort typerep}
  in if inst then thy else add-def tyco thy end;
```

end;
⟫

setup ⟪
Typerep.add-def @{*type-name prop*}
#> *Typerep.add-def* @{*type-name fun*}
#> *Typerep.add-def* @{*type-name itself*}
#> *Typerep.add-def* @{*type-name bool*}
#> *TypedefPackage.interpretation Typerep.perhaps-add-def*
⟫

lemma [*code*]:
eq-class.eq (*Typerep tyco1 tys1*) (*Typerep tyco2 tys2*) ⟷ *eq-class.eq tyco1 tyco2*
∧ *list-all2 eq-class.eq tys1 tys2*
by (*auto simp add*: *equals-eq* [*symmetric*] *list-all2-eq* [*symmetric*])

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

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

code-reserved *SML Term*

hide (**open**) *const typerep Typerep*

end

42 Code-Eval: Term evaluation using the generic code generator

theory *Code-Eval*
imports *Plain Typerep*
begin

42.1 Term representation

42.1.1 Terms and class *term-of*

datatype *term* = *dummy-term*

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

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

code-datatype *Const App*

class *term-of* = *typerep* +
  fixes *term-of* :: *'a*::{} ⇒ *term*

lemma *term-of-anything*: *term-of x* ≡ *t*
  by (*rule eq-reflection*) (*cases term-of x*, *cases t*, *simp*)

ML ⟪
structure Eval =
struct

fun mk-term f g (*Const* (*c*, *ty*)) =
      @{*term Const*} $ *Message-String.mk c* $ *g ty*
  | *mk-term f g* (*t1* $ *t2*) =
      @{*term App*} $ *mk-term f g t1* $ *mk-term f g t2*
  | *mk-term f g* (*Free v*) = *f v*
  | *mk-term f g* (*Bound i*) = *Bound i*
  | *mk-term f g* (*Abs* (*v*, -, *t*)) = *Abs* (*v*, @{*typ term*}, *mk-term f g t*);

fun mk-term-of ty t = *Const* (@{*const-name term-of*}, *ty* −−> @{*typ term*}) $ *t*;

end;
⟫

42.1.2 *term-of* instances

setup ⟪
let
  fun add-term-of-def ty vs tyco thy =
    let
      val lhs = *Const* (@{*const-name term-of*}, *ty* −−> @{*typ term*})
        $ *Free* (*x*, *ty*);
      val rhs = @{*term undefined* :: *term*};
      val eq = *HOLogic.mk-Trueprop* (*HOLogic.mk-eq* (*lhs*, *rhs*));
      fun triv-name-of t = (*fst o dest-Free o fst o strip-comb o fst*
        o HOLogic.dest-eq o HOLogic.dest-Trueprop) *t* ^ *-triv*;
    in
      thy
      |> *TheoryTarget.instantiation* ([*tyco*], *vs*, @{*sort term-of*})
      |> '(*fn lthy* => *Syntax.check-term lthy eq*)
     |−> (*fn eq* => *Specification.definition* (*NONE*, ((*Binding.name* (*triv-name-of*
eq), []), *eq*)))
      |> *snd*
      |> *Class.prove-instantiation-instance* (*K* (*Class.intro-classes-tac* []))
      |> *LocalTheory.exit-global*
    end;
  fun interpretator (*tyco*, (*raw-vs*, -)) *thy* =
    let
        val has-inst = *can* (*Sorts.mg-domain* (*Sign.classes-of thy*) *tyco*) @{*sort*

```
term-of};
      val constrain-sort =
        curry (Sorts.inter-sort (Sign.classes-of thy)) @{sort term-of};
      val vs = (map o apsnd) constrain-sort raw-vs;
      val ty = Type (tyco, map TFree vs);
    in
      thy
      |> Typerep.perhaps-add-def tyco
      |> not has-inst ? add-term-of-def ty vs tyco
    end;
in
  Code.type-interpretation interpretator
end
⟫

setup ⟪
let
  fun mk-term-of-eq ty vs tyco (c, tys) =
    let
      val t = list-comb (Const (c, tys ---> ty),
        map Free (Name.names Name.context a tys));
    in (map-aterms (fn Free (v, ty) => Var ((v, 0), ty) | t => t) t, Eval.mk-term
      (fn (v, ty) => Eval.mk-term-of ty (Var ((v, 0), ty)))
      (Typerep.mk (fn (v, sort) => Typerep.typerep (TFree (v, sort)))) t)
    end;
  fun prove-term-of-eq ty eq thy =
    let
      val cty = Thm.ctyp-of thy ty;
      val (arg, rhs) = pairself (Thm.cterm-of thy) eq;
      val thm = @{thm term-of-anything}
        |> Drule.instantiate' [SOME cty] [SOME arg, SOME rhs]
        |> Thm.varifyT;
    in
      thy
      |> Code.add-eqn thm
    end;
  fun interpretator (tyco, (raw-vs, raw-cs)) thy =
    let
      val constrain-sort =
        curry (Sorts.inter-sort (Sign.classes-of thy)) @{sort term-of};
      val vs = (map o apsnd) constrain-sort raw-vs;
      val cs = (map o apsnd o map o map-atyps)
        (fn TFree (v, sort) => TFree (v, constrain-sort sort)) raw-cs;
      val ty = Type (tyco, map TFree vs);
      val eqs = map (mk-term-of-eq ty vs tyco) cs;
      val const = AxClass.param-of-inst thy (@{const-name term-of}, tyco);
    in
      thy
      |> Code.del-eqns const
```

```
    |> fold (prove-term-of-eq ty) eqs
  end;
in
  Code.type-interpretation interpretator
end
⟫
```

42.1.3 Code generator setup

lemmas [*code del*] = *term.recs term.cases term.size*
lemma [*code*, *code del*]: *eq-class.eq* (*t1*::*term*) *t2* ⟷ *eq-class.eq t1 t2* **..**

lemma [*code*, *code del*]: (*term-of* :: *typerep* ⇒ *term*) = *term-of* **..**
lemma [*code*, *code del*]: (*term-of* :: *term* ⇒ *term*) = *term-of* **..**
lemma [*code*, *code del*]: (*term-of* :: *message-string* ⇒ *term*) = *term-of* **..**
lemma [*code*, *code del*]:
(*Code-Eval.term-of* :: *'a*::{*type*, *term-of*} *Predicate.pred* ⇒ *Code-Eval.term*) = *Code-Eval.term-of* **..**
lemma [*code*, *code del*]:
(*Code-Eval.term-of* :: *'a*::{*type*, *term-of*} *Predicate.seq* ⇒ *Code-Eval.term*) = *Code-Eval.term-of* **..**

lemma *term-of-char* [*unfolded typerep-fun-def typerep-char-def typerep-nibble-def*, *code*]: *Code-Eval.term-of c* =
(*let* (*n*, *m*) = *nibble-pair-of-char c*
in Code-Eval.App (*Code-Eval.App* (*Code-Eval.Const* (*STR ''Pair''*) (*TYPEREP*(*nibble* ⇒ *nibble* ⇒ *char*)))
(*Code-Eval.term-of n*)) (*Code-Eval.term-of m*))
by (*subst term-of-anything*) *rule*

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

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

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

42.2 Evaluation setup

ML ⟪
```
signature EVAL =
sig
  val mk-term: ((string * typ) -> term) -> (typ -> term) -> term -> term
  val eval-ref: (unit -> term) option ref
  val eval-term: theory -> term -> term
end;

structure Eval : EVAL =
```

```
struct

open Eval;

val eval-ref = ref (NONE : (unit -> term) option);

fun eval-term thy t =
  t
  |> Eval.mk-term-of (fastype-of t)
  |> (fn t => Code-ML.eval-term (Eval.eval-ref, eval-ref) thy t [])
  |> Code.postprocess-term thy;

end;
⟩⟩

setup ⟨⟨
  Value.add-evaluator (code, Eval.eval-term o ProofContext.theory-of)
⟩⟩
```

42.2.1 Syntax

```
print-translation ⟨⟨
let
  val term = Const (<TERM>, dummyT);
  fun tr1' [-, -] = term;
  fun tr2' [] = term;
in
  [(@{const-syntax Const}, tr1'),
    (@{const-syntax App}, tr1'),
    (@{const-syntax dummy-term}, tr2')]
end
⟩⟩

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

end
```

43 Map: Maps

```
theory Map
imports List
begin

types ('a,'b) ~=> = 'a => 'b option  (infixr 0)
translations (type) a ~=> b  <= (type) a => b option
```

syntax (*xsymbols*)
  ~=> :: [*type*, *type*] => *type* (**infixr** ⇀ *0*)

abbreviation
  empty :: *'a* ~=> *'b* **where**
  empty == %*x*. *None*

definition
  map-comp :: (*'b* ~=> *'c*) => (*'a* ~=> *'b*) => (*'a* ~=> *'c*) (**infixl** *o'-m 55*)
where
  f o-m g = (λ*k*. *case g k of None* ⇒ *None* | *Some v* ⇒ *f v*)

notation (*xsymbols*)
  map-comp (**infixl** ∘$_m$ *55*)

definition
  map-add :: (*'a* ~=> *'b*) => (*'a* ~=> *'b*) => (*'a* ~=> *'b*) (**infixl** ++ *100*)
where
  m1 ++ *m2* = (λ*x*. *case m2 x of None => m1 x* | *Some y => Some y*)

definition
  restrict-map :: (*'a* ~=> *'b*) => *'a set* => (*'a* ~=> *'b*) (**infixl** |` *110*) **where**
  m|`*A* = (λ*x*. *if x* : *A then m x else None*)

notation (*latex* **output**)
  restrict-map (-|`- [*111*,*110*] *110*)

definition
  dom :: (*'a* ~=> *'b*) => *'a set* **where**
  dom m = {*a*. *m a* ~= *None*}

definition
  ran :: (*'a* ~=> *'b*) => *'b set* **where**
  ran m = {*b*. *EX a*. *m a* = *Some b*}

definition
  map-le :: (*'a* ~=> *'b*) => (*'a* ~=> *'b*) => *bool* (**infix** ⊆$_m$ *50*) **where**
  (*m*$_1$ ⊆$_m$ *m*$_2$) = (∀ *a* ∈ *dom m*$_1$. *m*$_1$ *a* = *m*$_2$ *a*)

consts
  map-of :: (*'a* * *'b*) *list* => *'a* ~=> *'b*
  map-upds :: (*'a* ~=> *'b*) => *'a list* => *'b list* => (*'a* ~=> *'b*)

nonterminals
  maplets maplet

syntax
  -maplet :: [*'a*, *'a*] => *maplet* (- /|->/ -)
  -maplets :: [*'a*, *'a*] => *maplet* (- /[|->]/ -)

```
        :: maplet => maplets              (-)
  -Maplets :: [maplet, maplets] => maplets (-,/ -)
  -MapUpd  :: ['a ~=> 'b, maplets] => 'a ~=> 'b (-/'(-') [900,0]900)
  -Map     :: maplets => 'a ~=> 'b          ((1[-]))

syntax (xsymbols)
  -maplet  :: ['a, 'a] => maplet            (- /↦/ -)
  -maplets :: ['a, 'a] => maplet            (- /[↦]/ -)

translations
  -MapUpd m (-Maplets xy ms)  == -MapUpd (-MapUpd m xy) ms
  -MapUpd m (-maplet  x y)    == m(x:=Some y)
  -MapUpd m (-maplets x y)    == map-upds m x y
  -Map ms                     == -MapUpd (CONST empty) ms
  -Map (-Maplets ms1 ms2)     <= -MapUpd (-Map ms1) ms2
  -Maplets ms1 (-Maplets ms2 ms3) <= -Maplets (-Maplets ms1 ms2) ms3

primrec
  map-of [] = empty
  map-of (p#ps) = (map-of ps)(fst p |-> snd p)

declare map-of.simps [code del]

lemma map-of-Cons-code [code]:
  map-of [] k = None
  map-of ((l, v) # ps) k = (if l = k then Some v else map-of ps k)
  by simp-all

defs
  map-upds-def [code]: m(xs [|->] ys) == m ++ map-of (rev(zip xs ys))
```

43.1 *empty*

```
lemma empty-upd-none [simp]: empty(x := None) = empty
by (rule ext) simp
```

43.2 *map-upd*

```
lemma map-upd-triv: t k = Some x ==> t(k|->x) = t
by (rule ext) simp

lemma map-upd-nonempty [simp]: t(k|->x) ~= empty
proof
  assume t(k ↦ x) = empty
  then have (t(k ↦ x)) k = None by simp
  then show False by simp
qed

lemma map-upd-eqD1:
  assumes m(a↦x) = n(a↦y)
```

shows $x = y$
proof −
from *prems* **have** $(m(a\mapsto x))\ a = (n(a\mapsto y))\ a$ **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *map-upd-Some-unfold*:
$((m(a|->b))\ x = Some\ y) = (x = a \wedge b = y \vee x \neq a \wedge m\ x = Some\ y)$
by *auto*

lemma *image-map-upd* [*simp*]: $x \notin A \Longrightarrow m(x \mapsto y)\ `\ A = m\ `\ A$
by *auto*

lemma *finite-range-updI*: *finite* (*range f*) ==> *finite* (*range* ($f(a|->b)$))
unfolding *image-def*
apply (*simp* (*no-asm-use*) *add*:*full-SetCompr-eq*)
apply (*rule finite-subset*)
prefer *2* **apply** *assumption*
apply (*auto*)
done

43.3 *map-of*

lemma *map-of-eq-None-iff*:
(*map-of xys x* = *None*) = ($x \notin$ *fst* ` (*set xys*))
by (*induct xys*) *simp-all*

lemma *map-of-is-SomeD*: *map-of xys x* = *Some y* $\Longrightarrow$ (x,y) $\in$ *set xys*
apply (*induct xys*)
apply *simp*
apply (*clarsimp split*: *if-splits*)
done

lemma *map-of-eq-Some-iff* [*simp*]:
distinct(*map fst xys*) $\Longrightarrow$ (*map-of xys x* = *Some y*) = ((x,y) $\in$ *set xys*)
apply (*induct xys*)
apply *simp*
apply (*auto simp*: *map-of-eq-None-iff* [*symmetric*])
done

lemma *Some-eq-map-of-iff* [*simp*]:
distinct(*map fst xys*) $\Longrightarrow$ (*Some y* = *map-of xys x*) = ((x,y) $\in$ *set xys*)
by (*auto simp del*:*map-of-eq-Some-iff simp add*: *map-of-eq-Some-iff* [*symmetric*])

lemma *map-of-is-SomeI* [*simp*]: $[\![$ *distinct*(*map fst xys*); (x,y) $\in$ *set xys* $]\!]$
$\Longrightarrow$ *map-of xys x* = *Some y*
apply (*induct xys*)
apply *simp*
apply *force*

done

lemma *map-of-zip-is-None* [*simp*]:
length xs = *length ys* ⟹ (*map-of* (*zip xs ys*) *x* = *None*) = (*x* ∉ *set xs*)
by (*induct rule*: *list-induct2*) *simp-all*

lemma *map-of-zip-is-Some*:
assumes *length xs* = *length ys*
shows *x* ∈ *set xs* ⟷ (∃ *y*. *map-of* (*zip xs ys*) *x* = *Some y*)
using *assms* **by** (*induct rule*: *list-induct2*) *simp-all*

lemma *map-of-zip-upd*:
fixes *x* :: *'a* **and** *xs* :: *'a list* **and** *ys zs* :: *'b list*
assumes *length ys* = *length xs*
and *length zs* = *length xs*
and *x* ∉ *set xs*
and *map-of* (*zip xs ys*)(*x* ↦ *y*) = *map-of* (*zip xs zs*)(*x* ↦ *z*)
shows *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*)
proof
fix *x'* :: *'a*
show *map-of* (*zip xs ys*) *x'* = *map-of* (*zip xs zs*) *x'*
proof (*cases x* = *x'*)
case *True*
from *assms True map-of-zip-is-None* [*of xs ys x'*]
have *map-of* (*zip xs ys*) *x'* = *None* **by** *simp*
moreover from *assms True map-of-zip-is-None* [*of xs zs x'*]
have *map-of* (*zip xs zs*) *x'* = *None* **by** *simp*
ultimately show *?thesis* **by** *simp*
next
case *False* **from** *assms*
have (*map-of* (*zip xs ys*)(*x* ↦ *y*)) *x'* = (*map-of* (*zip xs zs*)(*x* ↦ *z*)) *x'* **by** *auto*
with *False* **show** *?thesis* **by** *simp*
qed
qed

lemma *map-of-zip-inject*:
assumes *length ys* = *length xs*
and *length zs* = *length xs*
and *dist*: *distinct xs*
and *map-of*: *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*)
shows *ys* = *zs*
using *assms*(*1*) *assms*(*2*)[*symmetric*] **using** *dist map-of* **proof** (*induct ys xs zs rule*: *list-induct3*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons y ys x xs z zs*)
from ‹*map-of* (*zip* (*x*#*xs*) (*y*#*ys*)) = *map-of* (*zip* (*x*#*xs*) (*z*#*zs*))›
have *map-of*: *map-of* (*zip xs ys*)(*x* ↦ *y*) = *map-of* (*zip xs zs*)(*x* ↦ *z*) **by** *simp*

from *Cons* **have** *length ys = length xs* **and** *length zs = length xs*
and *x ∉ set xs* **by** *simp-all*
then have *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*) **using** *map-of* **by** (*rule map-of-zip-upd*)
with *Cons.hyps* ‹*distinct* (*x # xs*)› **have** *ys = zs* **by** *simp*
moreover from *map-of* **have** *y = z* **by** (*rule map-upd-eqD1*)
ultimately show *?case* **by** *simp*
qed

lemma *finite-range-map-of*: *finite* (*range* (*map-of xys*))
apply (*induct xys*)
apply (*simp-all add*: *image-constant*)
apply (*rule finite-subset*)
prefer *2* **apply** *assumption*
apply *auto*
done

lemma *map-of-SomeD*: *map-of xs k = Some y ⟹* (*k*, *y*) ∈ *set xs*
by (*induct xs*) (*simp*, *atomize* (*full*), *auto*)

lemma *map-of-mapk-SomeI*:
inj f ==> map-of t k = Some x ==>
map-of (*map* (*split* (%*k*. *Pair* (*f k*))) *t*) (*f k*) = *Some x*
by (*induct t*) (*auto simp add*: *inj-eq*)

lemma *weak-map-of-SomeI*: (*k*, *x*) : *set l ==> ∃ x*. *map-of l k = Some x*
by (*induct l*) *auto*

lemma *map-of-filter-in*:
map-of xs k = Some z ⟹ P k z ⟹ map-of (*filter* (*split P*) *xs*) *k = Some z*
by (*induct xs*) *auto*

lemma *map-of-map*: *map-of* (*map* (%(*a*,*b*). (*a*,*f b*)) *xs*) *x = Option.map f* (*map-of xs x*)
by (*induct xs*) *auto*

43.4 *Option.map* related

lemma *option-map-o-empty* [*simp*]: *Option.map f o empty = empty*
by (*rule ext*) *simp*

lemma *option-map-o-map-upd* [*simp*]:
Option.map f o m(*a*|−>*b*) = (*Option.map f o m*)(*a*|−>*f b*)
by (*rule ext*) *simp*

43.5 *map-comp* related

lemma *map-comp-empty* [*simp*]:
m ∘$_m$ *empty = empty*
empty ∘$_m$ *m = empty*

by (*auto simp add*: *map-comp-def intro*: *ext split*: *option.splits*)

lemma *map-comp-simps* [*simp*]:
m2 k = *None* ⟹ (*m1* ∘$_m$ *m2*) *k* = *None*
m2 k = *Some k'* ⟹ (*m1* ∘$_m$ *m2*) *k* = *m1 k'*
by (*auto simp add*: *map-comp-def*)

lemma *map-comp-Some-iff*:
((*m1* ∘$_m$ *m2*) *k* = *Some v*) = (∃ *k'*. *m2 k* = *Some k'* ∧ *m1 k'* = *Some v*)
by (*auto simp add*: *map-comp-def split*: *option.splits*)

lemma *map-comp-None-iff*:
((*m1* ∘$_m$ *m2*) *k* = *None*) = (*m2 k* = *None* ∨ (∃ *k'*. *m2 k* = *Some k'* ∧ *m1 k'* = *None*))
by (*auto simp add*: *map-comp-def split*: *option.splits*)

43.6 ++

lemma *map-add-empty*[*simp*]: *m* ++ *empty* = *m*
by(*simp add*: *map-add-def*)

lemma *empty-map-add*[*simp*]: *empty* ++ *m* = *m*
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-assoc*[*simp*]: *m1* ++ (*m2* ++ *m3*) = (*m1* ++ *m2*) ++ *m3*
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-Some-iff*:
((*m* ++ *n*) *k* = *Some x*) = (*n k* = *Some x* | *n k* = *None* & *m k* = *Some x*)
by (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-SomeD* [*dest*!]:
(*m* ++ *n*) *k* = *Some x* ⟹ *n k* = *Some x* ∨ *n k* = *None* ∧ *m k* = *Some x*
by (*rule map-add-Some-iff* [*THEN iffD1*])

lemma *map-add-find-right* [*simp*]: !!*xx*. *n k* = *Some xx* ==> (*m* ++ *n*) *k* = *Some xx*
by (*subst map-add-Some-iff*) *fast*

lemma *map-add-None* [*iff*]: ((*m* ++ *n*) *k* = *None*) = (*n k* = *None* & *m k* = *None*)
by (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-upd*[*simp*]: *f* ++ *g*(*x*|−>*y*) = (*f* ++ *g*)(*x*|−>*y*)
by (*rule ext*) (*simp add*: *map-add-def*)

lemma *map-add-upds*[*simp*]: *m1* ++ (*m2*(*xs*[↦]*ys*)) = (*m1*++*m2*)(*xs*[↦]*ys*)
by (*simp add*: *map-upds-def*)

lemma *map-of-append*[*simp*]: *map-of* (*xs* @ *ys*) = *map-of ys* ++ *map-of xs*
unfolding *map-add-def*
apply (*induct xs*)
apply *simp*
apply (*rule ext*)
apply (*simp split add*: *option.split*)
done

lemma *finite-range-map-of-map-add*:
finite (*range f*) ==> *finite* (*range* (*f* ++ *map-of l*))
apply (*induct l*)
apply (*auto simp del*: *fun-upd-apply*)
apply (*erule finite-range-updI*)
done

lemma *inj-on-map-add-dom* [*iff*]:
inj-on (*m* ++ *m'*) (*dom m'*) = *inj-on m'* (*dom m'*)
by (*fastsimp simp*: *map-add-def dom-def inj-on-def split*: *option.splits*)

43.7 *restrict-map*

lemma *restrict-map-to-empty* [*simp*]: *m*|`{} = *empty*
by (*simp add*: *restrict-map-def*)

lemma *restrict-map-empty* [*simp*]: *empty*|`*D* = *empty*
by (*simp add*: *restrict-map-def*)

lemma *restrict-in* [*simp*]: *x* ∈ *A* ⟹ (*m*|`*A*) *x* = *m x*
by (*simp add*: *restrict-map-def*)

lemma *restrict-out* [*simp*]: *x* ∉ *A* ⟹ (*m*|`*A*) *x* = *None*
by (*simp add*: *restrict-map-def*)

lemma *ran-restrictD*: *y* ∈ *ran* (*m*|`*A*) ⟹ ∃ *x*∈*A*. *m x* = *Some y*
by (*auto simp*: *restrict-map-def ran-def split*: *split-if-asm*)

lemma *dom-restrict* [*simp*]: *dom* (*m*|`*A*) = *dom m* ∩ *A*
by (*auto simp*: *restrict-map-def dom-def split*: *split-if-asm*)

lemma *restrict-upd-same* [*simp*]: *m*(*x*↦*y*)|`(−{*x*}) = *m*|`(−{*x*})
by (*rule ext*) (*auto simp*: *restrict-map-def*)

lemma *restrict-restrict* [*simp*]: *m*|`*A*|`*B* = *m*|`(*A*∩*B*)
by (*rule ext*) (*auto simp*: *restrict-map-def*)

lemma *restrict-fun-upd* [*simp*]:
m(*x* := *y*)|`*D* = (*if x* ∈ *D then* (*m*|`(*D*−{*x*}))(*x* := *y*) *else m*|`*D*)
by (*simp add*: *restrict-map-def expand-fun-eq*)

lemma *fun-upd-None-restrict* [*simp*]:
  $(m|`D)(x := None) = (if\ x{:}D\ then\ m|`(D - \{x\})\ else\ m|`D)$
by (*simp add*: *restrict-map-def expand-fun-eq*)

lemma *fun-upd-restrict*: $(m|`D)(x := y) = (m|`(D-\{x\}))(x := y)$
by (*simp add*: *restrict-map-def expand-fun-eq*)

lemma *fun-upd-restrict-conv* [*simp*]:
  $x \in D \Longrightarrow (m|`D)(x := y) = (m|`(D-\{x\}))(x := y)$
by (*simp add*: *restrict-map-def expand-fun-eq*)

43.8 *map-upds*

lemma *map-upds-Nil1* [*simp*]: $m([]\ [|->]\ bs) = m$
by (*simp add*: *map-upds-def*)

lemma *map-upds-Nil2* [*simp*]: $m(as\ [|->]\ []) = m$
by (*simp add*:*map-upds-def*)

lemma *map-upds-Cons* [*simp*]: $m(a\#as\ [|->]\ b\#bs) = (m(a|->b))(as[|->]bs)$
by (*simp add*:*map-upds-def*)

lemma *map-upds-append1* [*simp*]: $\bigwedge ys\ m.\ size\ xs < size\ ys \Longrightarrow$
  $m(xs@[x]\ [\mapsto]\ ys) = m(xs\ [\mapsto]\ ys)(x \mapsto ys!size\ xs)$
apply(*induct xs*)
 apply (*clarsimp simp add*: *neq-Nil-conv*)
apply (*case-tac ys*)
 apply *simp*
apply *simp*
done

lemma *map-upds-list-update2-drop* [*simp*]:
  $[\![size\ xs \leq i;\ i < size\ ys]\!]$
    $\Longrightarrow m(xs[\mapsto]ys[i:=y]) = m(xs[\mapsto]ys)$
apply (*induct xs arbitrary*: *m ys i*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp*
apply (*simp split*: *nat.split*)
done

lemma *map-upd-upds-conv-if*:
  $(f(x|->y))(xs\ [|->]\ ys) =$
   $(if\ x : set(take\ (length\ ys)\ xs)\ then\ f(xs\ [|->]\ ys)$
                                $else\ (f(xs\ [|->]\ ys))(x|->y))$
apply (*induct xs arbitrary*: *x y ys f*)
 apply *simp*
apply (*case-tac ys*)
 apply (*auto split*: *split-if simp*: *fun-upd-twist*)

done

lemma *map-upds-twist* [*simp*]:
a ~: *set as* ==> *m*(*a*|−>*b*)(*as*[|−>]*bs*) = *m*(*as*[|−>]*bs*)(*a*|−>*b*)
using *set-take-subset* **by** (*fastsimp simp add*: *map-upd-upds-conv-if*)

lemma *map-upds-apply-nontin* [*simp*]:
x ~: *set xs* ==> (*f*(*xs*[|−>]*ys*)) *x* = *f x*
apply (*induct xs arbitrary*: *ys*)
apply *simp*
apply (*case-tac ys*)
apply (*auto simp*: *map-upd-upds-conv-if*)
done

lemma *fun-upds-append-drop* [*simp*]:
size xs = *size ys* ⟹ *m*(*xs*@*zs*[↦]*ys*) = *m*(*xs*[↦]*ys*)
apply (*induct xs arbitrary*: *m ys*)
apply *simp*
apply (*case-tac ys*)
apply *simp-all*
done

lemma *fun-upds-append2-drop* [*simp*]:
size xs = *size ys* ⟹ *m*(*xs*[↦]*ys*@*zs*) = *m*(*xs*[↦]*ys*)
apply (*induct xs arbitrary*: *m ys*)
apply *simp*
apply (*case-tac ys*)
apply *simp-all*
done

lemma *restrict-map-upds*[*simp*]:
⟦ *length xs* = *length ys*; *set xs* ⊆ *D* ⟧
⟹ *m*(*xs* [↦] *ys*)|‘*D* = (*m*|‘(*D* − *set xs*))(*xs* [↦] *ys*)
apply (*induct xs arbitrary*: *m ys*)
apply *simp*
apply (*case-tac ys*)
apply *simp*
apply (*simp add*: *Diff-insert* [*symmetric*] *insert-absorb*)
apply (*simp add*: *map-upd-upds-conv-if*)
done

43.9 *dom*

lemma *domI*: *m a* = *Some b* ==> *a* : *dom m*
by(*simp add*:*dom-def*)

lemma *domD*: *a* : *dom m* ==> ∃ *b*. *m a* = *Some b*

by (*cases m a*) (*auto simp add*: *dom-def*)

lemma *domIff* [*iff*, *simp del*]: (*a* : *dom m*) = (*m a* ~= *None*)
by(*simp add*:*dom-def*)

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

lemma *dom-fun-upd* [*simp*]:
dom(*f*(*x* := *y*)) = (*if y*=*None then dom f* − {*x*} *else insert x* (*dom f*))
by(*auto simp add*:*dom-def*)

lemma *dom-map-of*: *dom*(*map-of xys*) = {*x*. ∃ *y*. (*x*,*y*) : *set xys*}
by (*induct xys*) (*auto simp del*: *fun-upd-apply*)

lemma *dom-map-of-conv-image-fst*:
dom(*map-of xys*) = *fst* ' (*set xys*)
by(*force simp*: *dom-map-of*)

lemma *dom-map-of-zip* [*simp*]: [| *length xs* = *length ys*; *distinct xs* |] ==>
dom(*map-of*(*zip xs ys*)) = *set xs*
by (*induct rule*: *list-induct2*) *simp-all*

lemma *finite-dom-map-of*: *finite* (*dom* (*map-of l*))
by (*induct l*) (*auto simp add*: *dom-def insert-Collect* [*symmetric*])

lemma *dom-map-upds* [*simp*]:
dom(*m*(*xs*[|−>]*ys*)) = *set*(*take* (*length ys*) *xs*) *Un dom m*
apply (*induct xs arbitrary*: *m ys*)
apply *simp*
apply (*case-tac ys*)
apply *auto*
done

lemma *dom-map-add* [*simp*]: *dom*(*m*++*n*) = *dom n Un dom m*
by(*auto simp*:*dom-def*)

lemma *dom-override-on* [*simp*]:
dom(*override-on f g A*) =
(*dom f* − {*a*. *a* : *A* − *dom g*}) *Un* {*a*. *a* : *A Int dom g*}
by(*auto simp*: *dom-def override-on-def*)

lemma *map-add-comm*: *dom m1* ∩ *dom m2* = {} ⟹ *m1*++*m2* = *m2*++*m1*
by (*rule ext*) (*force simp*: *map-add-def dom-def split*: *option.split*)

lemma *dom-const* [*simp*]:
dom (λ*x*. *Some y*) = *UNIV*
by *auto*

lemma *dom-if*:
dom (λ*x*. *if P x then f x else g x*) = *dom f* ∩ {*x*. *P x*} ∪ *dom g* ∩ {*x*. ¬ *P x*}
by (*auto split*: *if-splits*)

lemma *finite-map-freshness*:
finite (*dom* (*f* :: '*a* ⇀ '*b*)) ⟹ ¬ *finite* (*UNIV* :: '*a set*) ⟹
∃ *x*. *f x* = *None*
by(*bestsimp dest*:*ex-new-if-finite*)

lemma *dom-minus*:
f x = *None* ⟹ *dom f* − *insert x A* = *dom f* − *A*
unfolding *dom-def* **by** *simp*

lemma *insert-dom*:
f x = *Some y* ⟹ *insert x* (*dom f*) = *dom f*
unfolding *dom-def* **by** *auto*

43.10 *ran*

lemma *ranI*: *m a* = *Some b* ==> *b* : *ran m*
by(*auto simp*: *ran-def*)

lemma *ran-empty* [*simp*]: *ran empty* = {}
by(*auto simp*: *ran-def*)

lemma *ran-map-upd* [*simp*]: *m a* = *None* ==> *ran*(*m*(*a*|−>*b*)) = *insert b* (*ran m*)
unfolding *ran-def*
apply *auto*
apply (*subgoal-tac aa* ~= *a*)
apply *auto*
done

43.11 *map-le*

lemma *map-le-empty* [*simp*]: *empty* ⊆$_m$ *g*
by (*simp add*: *map-le-def*)

lemma *upd-None-map-le* [*simp*]: *f*(*x* := *None*) ⊆$_m$ *f*
by (*force simp add*: *map-le-def*)

lemma *map-le-upd*[*simp*]: *f* ⊆$_m$ *g* ==> *f*(*a* := *b*) ⊆$_m$ *g*(*a* := *b*)
by (*fastsimp simp add*: *map-le-def*)

lemma *map-le-imp-upd-le* [*simp*]: *m1* ⊆$_m$ *m2* ⟹ *m1*(*x* := *None*) ⊆$_m$ *m2*(*x* ↦ *y*)
by (*force simp add*: *map-le-def*)

lemma *map-le-upds* [*simp*]:
  $f \subseteq_m g$ ==> $f(as\ [|->]\ bs) \subseteq_m g(as\ [|->]\ bs)$
apply (*induct as arbitrary*: *f g bs*)
 apply *simp*
apply (*case-tac bs*)
 apply *auto*
done

lemma *map-le-implies-dom-le*: $(f \subseteq_m g) \Longrightarrow (dom\ f \subseteq dom\ g)$
by (*fastsimp simp add*: *map-le-def dom-def*)

lemma *map-le-refl* [*simp*]: $f \subseteq_m f$
by (*simp add*: *map-le-def*)

lemma *map-le-trans*[*trans*]: $[\![\ m1 \subseteq_m m2;\ m2 \subseteq_m m3\ ]\!] \Longrightarrow m1 \subseteq_m m3$
by (*auto simp add*: *map-le-def dom-def*)

lemma *map-le-antisym*: $[\![\ f \subseteq_m g;\ g \subseteq_m f\ ]\!] \Longrightarrow f = g$
unfolding *map-le-def*
apply (*rule ext*)
apply (*case-tac* $x \in dom\ f$, *simp*)
apply (*case-tac* $x \in dom\ g$, *simp*, *fastsimp*)
done

lemma *map-le-map-add* [*simp*]: $f \subseteq_m (g\ ++\ f)$
by (*fastsimp simp add*: *map-le-def*)

lemma *map-le-iff-map-add-commute*: $(f \subseteq_m f\ ++\ g) = (f++g = g++f)$
by(*fastsimp simp*: *map-add-def map-le-def expand-fun-eq split*: *option.splits*)

lemma *map-add-le-mapE*: $f++g \subseteq_m h \Longrightarrow g \subseteq_m h$
by (*fastsimp simp add*: *map-le-def map-add-def dom-def*)

lemma *map-add-le-mapI*: $[\![\ f \subseteq_m h;\ g \subseteq_m h;\ f \subseteq_m f++g\ ]\!] \Longrightarrow f++g \subseteq_m h$
by (*clarsimp simp add*: *map-le-def map-add-def dom-def split*: *option.splits*)

end

44 Refute: Refute

theory *Refute*
imports *Hilbert-Choice List Record*
uses
  Tools/prop-logic.ML
  Tools/sat-solver.ML
  Tools/refute.ML
  Tools/refute-isar.ML

begin

setup *Refute.setup*

```
(* ------------------------------------------------------------------------ *)
(* REFUTE                                                                   *)
(*                                                                          *)
(* We use a SAT solver to search for a (finite) model that refutes a given  *)
(* HOL formula.                                                             *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* NOTE                                                                     *)
(*                                                                          *)
(* I strongly recommend that you install a stand-alone SAT solver if you    *)
(* want to use 'refute'.  For details see 'HOL/Tools/sat_solver.ML'.  If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME' *)
(* in 'etc/settings'.                                                       *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* USAGE                                                                    *)
(*                                                                          *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples.  The supported   *)
(* parameters are explained below.                                          *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* CURRENT LIMITATIONS                                                      *)
(*                                                                          *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and          *)
(* inductively defined sets.  Constants are unfolded automatically, and sort *)
(* axioms are added as well.  Other, user-asserted axioms however are       *)
(* ignored.  Inductive datatypes and recursive functions are supported, but *)
(* may lead to spurious countermodels.                                      *)
(*                                                                          *)
(* The (space) complexity of the algorithm is non-elementary.               *)
(*                                                                          *)
(* Schematic type variables are not supported.                              *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* PARAMETERS                                                               *)
(*                                                                          *)
(* The following global parameters are currently supported (and required):  *)
(*                                                                          *)
(* Name          Type    Description                                        *)
```

```
(*                                                                          *)
(* "minsize"     int     Only search for models with size at least          *)
(*                       'minsize'.                                         *)
(* "maxsize"     int     If >0, only search for models with size at most    *)
(*                       'maxsize'.                                         *)
(* "maxvars"     int     If >0, use at most 'maxvars' boolean variables     *)
(*                       when transforming the term into a propositional    *)
(*                       formula.                                           *)
(* "maxtime"     int     If >0, terminate after at most 'maxtime' seconds.  *)
(*                       This value is ignored under some ML compilers.     *)
(* "satsolver"   string  Name of the SAT solver to be used.                 *)
(*                                                                          *)
(* See 'HOL/SAT.thy' for default values.                                    *)
(*                                                                          *)
(* The size of particular types can be specified in the form type=size      *)
(* (where 'type' is a string, and 'size' is an int).  Examples:             *)
(* "'a"=1                                                                   *)
(* "List.list"=2                                                            *)
(* ------------------------------------------------------------------------ *)

(* ------------------------------------------------------------------------ *)
(* FILES                                                                    *)
(*                                                                          *)
(* HOL/Tools/prop_logic.ML    Propositional logic                           *)
(* HOL/Tools/sat_solver.ML    SAT solvers                                   *)
(* HOL/Tools/refute.ML        Translation HOL -> propositional logic and    *)
(*                            Boolean assignment -> HOL model               *)
(* HOL/Tools/refute_isar.ML   Adds 'refute'/'refute_params' to Isabelle's   *)
(*                            syntax                                        *)
(* HOL/Refute.thy             This file: loads the ML files, basic setup,   *)
(*                            documentation                                 *)
(* HOL/SAT.thy                Sets default parameters                       *)
(* HOL/ex/RefuteExamples.thy  Examples                                      *)
(* ------------------------------------------------------------------------ *)
```

end

45 SAT: Reconstructing external resolution proofs for propositional logic

theory *SAT*
imports *Refute*
uses
Tools/cnf-funcs.ML
Tools/sat-funcs.ML
begin

Late package setup: default values for refute, see also theory *Refute*.

refute-params
[*itself=1*,
minsize=1,
maxsize=8,
maxvars=10000,
maxtime=60,
satsolver=auto]

ML ⟪ *structure sat = SATFunc(structure cnf = cnf);* ⟫

method-setup *sat* = ⟪ *Scan.succeed (K (SIMPLE-METHOD' sat.sat-tac))* ⟫
SAT solver

method-setup *satx* = ⟪ *Scan.succeed (K (SIMPLE-METHOD' sat.satx-tac))* ⟫
SAT solver (with definitional CNF)

end

46 Main: Main HOL

theory *Main*
imports *Plain Code-Eval Map Recdef SAT*
begin

Classical Higher-order Logic – only "Main", excluding real and complex numbers etc.

See further [1]

end

47 Lubs: Definitions of Upper Bounds and Least Upper Bounds

theory *Lubs*
imports *Main*
begin

Thanks to suggestions by James Margetson

definition
setle :: ['*a set*, '*a*::*ord*] => *bool* (**infixl** *<= *70*) **where**
S *<= *x* = (*ALL y*: *S*. *y* <= *x*)

definition
setge :: ['*a*::*ord*, '*a set*] => *bool* (**infixl** <=* *70*) **where**

$x <=* S = (ALL\ y{:}\ S.\ x <= y)$

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

definition
$isUb :: ['a\ set,\ 'a\ set,\ 'a::ord] => bool$ **where**
$isUb\ R\ S\ x = (S\ *<=\ x\ \&\ x{:}\ R)$

definition
$isLub :: ['a\ set,\ 'a\ set,\ 'a::ord] => bool$ **where**
$isLub\ R\ S\ x = leastP\ (isUb\ R\ S)\ x$

definition
$ubs :: ['a\ set,\ 'a::ord\ set] => 'a\ set$ **where**
$ubs\ R\ S = Collect\ (isUb\ R\ S)$

47.1 Rules for the Relations *<= and <=*

lemma *setleI*: $ALL\ y{:}\ S.\ y <= x ==> S\ *<=\ x$
by (*simp add*: *setle-def*)

lemma *setleD*: $[|\ S\ *<=\ x;\ y{:}\ S\ |] ==> y <= x$
by (*simp add*: *setle-def*)

lemma *setgeI*: $ALL\ y{:}\ S.\ x<=\ y ==> x <=*\ S$
by (*simp add*: *setge-def*)

lemma *setgeD*: $[|\ x <=*\ S;\ y{:}\ S\ |] ==> x <= y$
by (*simp add*: *setge-def*)

47.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: $leastP\ P\ x ==> P\ x$
by (*simp add*: *leastP-def*)

lemma *leastPD2*: $leastP\ P\ x ==> x <=*\ Collect\ P$
by (*simp add*: *leastP-def*)

lemma *leastPD3*: $[|\ leastP\ P\ x;\ y{:}\ Collect\ P\ |] ==> x <= y$
by (*blast dest*!: *leastPD2 setgeD*)

lemma *isLubD1*: $isLub\ R\ S\ x ==> S\ *<=\ x$
by (*simp add*: *isLub-def isUb-def leastP-def*)

lemma *isLubD1a*: $isLub\ R\ S\ x ==> x{:}\ R$
by (*simp add*: *isLub-def isUb-def leastP-def*)

lemma *isLub-isUb*: $isLub\ R\ S\ x ==> isUb\ R\ S\ x$

apply (*simp add*: *isUb-def*)
apply (*blast dest*: *isLubD1 isLubD1a*)
done

lemma *isLubD2*: [| *isLub R S x*; *y* : *S* |] ==> *y* <= *x*
by (*blast dest*!: *isLubD1 setleD*)

lemma *isLubD3*: *isLub R S x* ==> *leastP*(*isUb R S*) *x*
by (*simp add*: *isLub-def*)

lemma *isLubI1*: *leastP*(*isUb R S*) *x* ==> *isLub R S x*
by (*simp add*: *isLub-def*)

lemma *isLubI2*: [| *isUb R S x*; *x* <=* *Collect* (*isUb R S*) |] ==> *isLub R S x*
by (*simp add*: *isLub-def leastP-def*)

lemma *isUbD*: [| *isUb R S x*; *y* : *S* |] ==> *y* <= *x*
by (*simp add*: *isUb-def setle-def*)

lemma *isUbD2*: *isUb R S x* ==> *S* *<= *x*
by (*simp add*: *isUb-def*)

lemma *isUbD2a*: *isUb R S x* ==> *x*: *R*
by (*simp add*: *isUb-def*)

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

lemma *isLub-le-isUb*: [| *isLub R S x*; *isUb R S y* |] ==> *x* <= *y*
apply (*simp add*: *isLub-def*)
apply (*blast intro*!: *leastPD3*)
done

lemma *isLub-ubs*: *isLub R S x* ==> *x* <=* *ubs R S*
apply (*simp add*: *ubs-def isLub-def*)
apply (*erule leastPD2*)
done

end

48 GCD: The Greatest Common Divisor

theory *GCD*
imports *Main*
begin

See [**?**].

48.1 Specification of GCD on nats

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

Uniqueness

lemma *is-gcd-unique*: *is-gcd a b m* ⟹ *is-gcd a b n* ⟹ *m* = *n*
by (*simp add*: *is-gcd-def*) (*blast intro*: *dvd-anti-sym*)

Connection to divides relation

lemma *is-gcd-dvd*: *is-gcd a b m* ⟹ *k dvd a* ⟹ *k dvd b* ⟹ *k dvd m*
by (*auto simp add*: *is-gcd-def*)

Commutativity

lemma *is-gcd-commute*: *is-gcd m n k* = *is-gcd n m k*
by (*auto simp add*: *is-gcd-def*)

48.2 GCD on nat by Euclid's algorithm

fun
gcd :: *nat* => *nat* => *nat*
where
gcd m n = (*if n* = *0 then m else gcd n* (*m mod n*))
lemma *gcd-induct* [*case-names 0 rec*]:
fixes *m n* :: *nat*
assumes ⋀*m*. *P m 0*
and ⋀*m n*. *0* < *n* ⟹ *P n* (*m mod n*) ⟹ *P m n*
shows *P m n*
proof (*induct m n rule*: *gcd.induct*)
case (*1 m n*) **with** *assms* **show** *?case* **by** (*cases n* = *0*) *simp-all*
qed

lemma *gcd-0* [*simp*, *algebra*]: *gcd m 0* = *m*
by *simp*

lemma *gcd-0-left* [*simp*,*algebra*]: *gcd 0 m* = *m*
by *simp*

lemma *gcd-non-0*: *n* > *0* ⟹ *gcd m n* = *gcd n* (*m mod n*)
by *simp*

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

lemma *nat-gcd-1-right* [*simp*, *algebra*]: *gcd m 1* = *1*
unfolding *One-nat-def* **by** (*rule gcd-1*)

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

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

lemma *gcd-dvd1* [*iff*, *algebra*]: *gcd m n dvd m*
and *gcd-dvd2* [*iff*, *algebra*]: *gcd m n dvd n*
apply (*induct m n rule*: *gcd-induct*)
apply (*simp-all add*: *gcd-non-0*)
apply (*blast dest*: *dvd-mod-imp-dvd*)
done

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

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

Function gcd yields the Greatest Common Divisor.

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

48.3 Derived laws for GCD

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

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

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

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

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

lemma *nat-gcd-1-left* [*simp*, *algebra*]: *gcd 1 m* = *1*
unfolding *One-nat-def* **by** (*rule gcd-1-left*)

Multiplication laws

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

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

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

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

lemma *relprime-dvd-mult-iff*: $gcd\ k\ n = 1 \Longrightarrow (k\ dvd\ m * n) = (k\ dvd\ m)$
by (*auto intro*: *relprime-dvd-mult dvd-mult2*)

lemma *gcd-mult-cancel*: $gcd\ k\ n = 1 \Longrightarrow gcd\ (k * m)\ n = gcd\ m\ n$
apply (*rule dvd-anti-sym*)
apply (*rule gcd-greatest*)
apply (*rule-tac n = k* **in** *relprime-dvd-mult*)
apply (*simp add*: *gcd-assoc*)
apply (*simp add*: *gcd-commute*)
apply (*simp-all add*: *mult-commute*)
done

Addition laws

lemma *gcd-add1* [*simp*, *algebra*]: $gcd\ (m + n)\ n = gcd\ m\ n$
by (*cases n = 0*) (*auto simp add*: *gcd-non-0*)

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

lemma *gcd-add2'* [*simp*, *algebra*]: $gcd\ m\ (n + m) = gcd\ m\ n$

apply (*subst add-commute*)
apply (*rule gcd-add2*)
done

lemma *gcd-add-mult*[*algebra*]: *gcd m* (*k* ∗ *m* + *n*) = *gcd m n*
by (*induct k*) (*simp-all add*: *add-assoc*)

lemma *gcd-dvd-prod*: *gcd m n dvd m* ∗ *n*
using *mult-dvd-mono* [*of 1*] **by** *auto*

Division by gcd yields rrelatively primes.

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

lemma *gcd-unique*: *d dvd a*∧*d dvd b* ∧ (∀ *e*. *e dvd a* ∧ *e dvd b* ⟶ *e dvd d*) ⟷ *d* = *gcd a b*
proof(*auto*)
assume *H*: *d dvd a d dvd b* ∀ *e*. *e dvd a* ∧ *e dvd b* ⟶ *e dvd d*
from *H*(*3*)[*rule-format*] *gcd-dvd1*[*of a b*] *gcd-dvd2*[*of a b*]
have *th*: *gcd a b dvd d* **by** *blast*
from *dvd-anti-sym*[*OF th gcd-greatest*[*OF H*(*1*,*2*)]] **show** *d* = *gcd a b* **by** *blast*
qed

lemma *gcd-eq*: **assumes** *H*: ∀ *d*. *d dvd x* ∧ *d dvd y* ⟷ *d dvd u* ∧ *d dvd v*
shows *gcd x y* = *gcd u v*
proof−
from *H* **have** ∀ *d*. *d dvd x* ∧ *d dvd y* ⟷ *d dvd gcd u v* **by** *simp*
with *gcd-unique*[*of gcd u v x y*] **show** *?thesis* **by** *auto*

qed

lemma *ind-euclid*:
assumes *c*: $\forall a\ b.\ P\ (a::nat)\ b \longleftrightarrow P\ b\ a$ **and** *z*: $\forall a.\ P\ a\ 0$
and *add*: $\forall a\ b.\ P\ a\ b \longrightarrow P\ a\ (a + b)$
shows $P\ a\ b$
proof(*induct* $n \equiv a+b$ *arbitrary*: *a b rule*: *nat-less-induct*)
fix *n a b*
assume *H*: $\forall m < n.\ \forall a\ b.\ m = a + b \longrightarrow P\ a\ b\ n = a + b$
have $a = b \vee a < b \vee b < a$ **by** *arith*
moreover {**assume** *eq*: $a = b$
from *add*[*rule-format*, *OF z*[*rule-format*, *of a*]] **have** $P\ a\ b$ **using** *eq* **by** *simp*}
moreover
{**assume** *lt*: $a < b$
hence $a + b - a < n \vee a = 0$ **using** *H*(*2*) **by** *arith*
moreover
{**assume** $a = 0$ **with** *z c* **have** $P\ a\ b$ **by** *blast* }
moreover
{**assume** *ab*: $a + b - a < n$
have *th0*: $a + b - a = a + (b - a)$ **using** *lt* **by** *arith*
from *add*[*rule-format*, *OF H*(*1*)[*rule-format*, *OF ab th0*]]
have $P\ a\ b$ **by** (*simp add*: *th0*[*symmetric*])}
ultimately have $P\ a\ b$ **by** *blast*}
moreover
{**assume** *lt*: $a > b$
hence $b + a - b < n \vee b = 0$ **using** *H*(*2*) **by** *arith*
moreover
{**assume** $b = 0$ **with** *z c* **have** $P\ a\ b$ **by** *blast* }
moreover
{**assume** *ab*: $b + a - b < n$
have *th0*: $b + a - b = b + (a - b)$ **using** *lt* **by** *arith*
from *add*[*rule-format*, *OF H*(*1*)[*rule-format*, *OF ab th0*]]
have $P\ b\ a$ **by** (*simp add*: *th0*[*symmetric*])
hence $P\ a\ b$ **using** *c* **by** *blast* }
ultimately have $P\ a\ b$ **by** *blast*}
ultimately show $P\ a\ b$ **by** *blast*
qed

lemma *bezout-lemma*:
assumes *ex*: $\exists (d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x = b * y + d \vee b * x = a * y + d)$
shows $\exists d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ a + b \wedge (a * x = (a + b) * y + d \vee (a + b) * x = a * y + d)$
using *ex*
apply *clarsimp*
apply (*rule-tac* $x=d$ **in** *exI*, *simp add*: *dvd-add*)
apply (*case-tac* $a * x = b * y + d$, *simp-all*)
apply (*rule-tac* $x=x + y$ **in** *exI*)
apply (*rule-tac* $x=y$ **in** *exI*)

apply *algebra*
apply (*rule-tac x=x* **in** *exI*)
apply (*rule-tac x=x + y* **in** *exI*)
apply *algebra*
done

lemma *bezout-add*: $\exists(d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x = b * y + d \vee b * x = a * y + d)$
apply(*induct a b rule*: *ind-euclid*)
apply *blast*
apply *clarify*
apply (*rule-tac x=a* **in** *exI*, *simp add*: *dvd-add*)
apply *clarsimp*
apply (*rule-tac x=d* **in** *exI*)
apply (*case-tac* $a * x = b * y + d$, *simp-all add*: *dvd-add*)
apply (*rule-tac x=x+y* **in** *exI*)
apply (*rule-tac x=y* **in** *exI*)
apply *algebra*
apply (*rule-tac x=x* **in** *exI*)
apply (*rule-tac x=x+y* **in** *exI*)
apply *algebra*
done

lemma *bezout*: $\exists(d::nat)\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge (a * x - b * y = d \vee b * x - a * y = d)$
using *bezout-add*[*of a b*]
apply *clarsimp*
apply (*rule-tac x=d* **in** *exI*, *simp*)
apply (*rule-tac x=x* **in** *exI*)
apply (*rule-tac x=y* **in** *exI*)
apply *auto*
done

We can get a stronger version with a nonzeroness assumption.

lemma *divides-le*: *m dvd n* ==> $m <= n \vee n = (0::nat)$ **by** (*auto simp add*: *dvd-def*)

lemma *bezout-add-strong*: **assumes** *nz*: $a \neq (0::nat)$
shows $\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge a * x = b * y + d$
proof−
from *nz* **have** *ap*: $a > 0$ **by** *simp*
from *bezout-add*[*of a b*]
have $(\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge a * x = b * y + d) \vee (\exists\, d\ x\ y.\ d\ dvd\ a \wedge d\ dvd\ b \wedge b * x = a * y + d)$ **by** *blast*
moreover
{fix *d x y* **assume** *H*: *d dvd a d dvd b a* $* x = b * y + d$
from *H* **have** *?thesis* **by** *blast* **}**
moreover
{fix *d x y* **assume** *H*: *d dvd a d dvd b b* $* x = a * y + d$

{assume *b0*: $b = 0$ **with** *H* **have** *?thesis* **by** *simp*}
moreover
{assume *b*: $b \neq 0$ **hence** *bp*: $b > 0$ **by** *simp*
from *divides-le*[*OF H*(*2*)] *b* **have** $d < b \vee d = b$ **using** *le-less* **by** *blast*
moreover
{assume *db*: *d*=*b*
from *prems* **have** *?thesis* **apply** *simp*
apply (*rule exI*[**where** $x = b$], *simp*)
apply (*rule exI*[**where** $x = b$])
by (*rule exI*[**where** $x = a - 1$], *simp add*: *diff-mult-distrib2*)}
moreover
{assume *db*: $d < b$
{assume *x*=*0* **hence** *?thesis* **using** *prems* **by** *simp* }
moreover
{assume *x0*: $x \neq 0$ **hence** *xp*: $x > 0$ **by** *simp*

from *db* **have** $d \leq b - 1$ **by** *simp*
hence $d*b \leq b*(b - 1)$ **by** *simp*
with *xp mult-mono*[*of 1 x d*∗*b b*∗(*b* − *1*)]
have *dble*: $d*b \leq x*b*(b - 1)$ **using** *bp* **by** *simp*
from *H* (*3*) **have** $a * ((b - 1) * y) + d * (b - 1 + 1) = d + x*b*(b - 1)$ **by** *algebra*
hence $a * ((b - 1) * y) = d + x*b*(b - 1) - d*b$ **using** *bp* **by** *simp*
hence $a * ((b - 1) * y) = d + (x*b*(b - 1) - d*b)$
by (*simp only*: *diff-add-assoc*[*OF dble*, *of d*, *symmetric*])
hence $a * ((b - 1) * y) = b*(x*(b - 1) - d) + d$
by (*simp only*: *diff-mult-distrib2 add-commute mult-ac*)
hence *?thesis* **using** *H*(*1*,*2*)
apply −
apply (*rule exI*[**where** *x*=*d*], *simp*)
apply (*rule exI*[**where** $x=(b - 1) * y$])
by (*rule exI*[**where** $x=x*(b - 1) - d$], *simp*)}
ultimately have *?thesis* **by** *blast*}
ultimately have *?thesis* **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

lemma *bezout-gcd*: $\exists\, x\ y.\ a * x - b * y = gcd\ a\ b \vee b * x - a * y = gcd\ a\ b$
proof−
let *?g* = *gcd a b*
from *bezout*[*of a b*] **obtain** *d x y* **where** *d*: *d dvd a d dvd b a* ∗ *x* − *b* ∗ *y* = *d* ∨ *b* ∗ *x* − *a* ∗ *y* = *d* **by** *blast*
from *d*(*1*,*2*) **have** *d dvd ?g* **by** *simp*
then obtain *k* **where** *k*: *?g* = *d*∗*k* **unfolding** *dvd-def* **by** *blast*
from *d*(*3*) **have** $(a * x - b * y)*k = d*k \vee (b * x - a * y)*k = d*k$ **by** *blast*
hence $a * x * k - b * y*k = d*k \vee b * x * k - a * y*k = d*k$
by (*algebra add*: *diff-mult-distrib*)

hence $a * (x * k) - b * (y*k) = ?g \vee b * (x * k) - a * (y*k) = ?g$
by (*simp add*: *k mult-assoc*)
thus *?thesis* **by** *blast*
qed

lemma *bezout-gcd-strong*: **assumes** *a*: $a \neq 0$
shows $\exists x\ y.\ a * x = b * y + gcd\ a\ b$
proof−
let *?g* = *gcd a b*
from *bezout-add-strong*[*OF a*, *of b*]
obtain *d x y* **where** *d*: *d dvd a d dvd b a* $* x = b * y + d$ **by** *blast*
from *d*(*1*,*2*) **have** *d dvd ?g* **by** *simp*
then obtain *k* **where** *k*: *?g* = *d*∗*k* **unfolding** *dvd-def* **by** *blast*
from *d*(*3*) **have** $a * x * k = (b * y + d) * k$ **by** *algebra*
hence $a * (x * k) = b * (y*k) + ?g$ **by** (*algebra add*: *k*)
thus *?thesis* **by** *blast*
qed

lemma *gcd-mult-distrib*: *gcd*(*a* ∗ *c*) (*b* ∗ *c*) = *c* ∗ *gcd a b*
by(*simp add*: *gcd-mult-distrib2 mult-commute*)

lemma *gcd-bezout*: $(\exists x\ y.\ a * x - b * y = d \vee b * x - a * y = d) \longleftrightarrow$ *gcd a b dvd d*
(**is** *?lhs* ⟷ *?rhs*)
proof−
let *?g* = *gcd a b*
{assume *H*: *?rhs* **then obtain** *k* **where** *k*: *d* = *?g*∗*k* **unfolding** *dvd-def* **by** *blast*
from *bezout-gcd*[*of a b*] **obtain** *x y* **where** *xy*: $a * x - b * y = ?g \vee b * x - a * y = ?g$
by *blast*
hence $(a * x - b * y)*k = ?g*k \vee (b * x - a * y)*k = ?g*k$ **by** *auto*
hence $a * x*k - b * y*k = ?g*k \vee b * x * k - a * y*k = ?g*k$
by (*simp only*: *diff-mult-distrib*)
hence $a * (x*k) - b * (y*k) = d \vee b * (x * k) - a * (y*k) = d$
by (*simp add*: *k*[*symmetric*] *mult-assoc*)
hence *?lhs* **by** *blast***}**
moreover
{fix *x y* **assume** *H*: $a * x - b * y = d \vee b * x - a * y = d$
have *dv*: *?g dvd a*∗*x ?g dvd b* ∗ *y ?g dvd b*∗*x ?g dvd a* ∗ *y*
using *dvd-mult2*[*OF gcd-dvd1*[*of a b*]] *dvd-mult2*[*OF gcd-dvd2*[*of a b*]] **by** *simp-all*
from *nat-dvd-diff*[*OF dv*(*1*,*2*)] *nat-dvd-diff*[*OF dv*(*3*,*4*)] *H*
have *?rhs* **by** *auto***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *gcd-bezout-sum*: **assumes** *H*:$a * x + b * y = d$ **shows** *gcd a b dvd d*
proof−

let *?g* = *gcd a b*
have *dv*: *?g dvd a*∗*x ?g dvd b* ∗ *y*
using *dvd-mult2*[*OF gcd-dvd1*[*of a b*]] *dvd-mult2*[*OF gcd-dvd2*[*of a b*]] **by** *simp-all*
from *dvd-add*[*OF dv*] *H*
show *?thesis* **by** *auto*
qed

lemma *gcd-mult'*: *gcd b* (*a* ∗ *b*) = *b*
by (*simp add*: *gcd-mult mult-commute*[*of a b*])

lemma *gcd-add*: *gcd*(*a* + *b*) *b* = *gcd a b*
gcd(*b* + *a*) *b* = *gcd a b gcd a* (*a* + *b*) = *gcd a b gcd a* (*b* + *a*) = *gcd a b*
apply (*simp-all add*: *gcd-add1*)
by (*simp add*: *gcd-commute gcd-add1*)

lemma *gcd-sub*: *b* <= *a* ==> *gcd*(*a* − *b*) *b* = *gcd a b a* <= *b* ==> *gcd a* (*b* − *a*) = *gcd a b*
proof−
{fix *a b* **assume** *H*: *b* ≤ (*a*::*nat*)
hence *th*: *a* − *b* + *b* = *a* **by** *arith*
from *gcd-add*(*1*)[*of a* − *b b*] *th* **have** *gcd*(*a* − *b*) *b* = *gcd a b* **by** *simp***}**
note *th* = *this*
{
assume *ab*: *b* ≤ *a*
from *th*[*OF ab*] **show** *gcd* (*a* − *b*) *b* = *gcd a b* **by** *blast*
next
assume *ab*: *a* ≤ *b*
from *th*[*OF ab*] **show** *gcd a* (*b* − *a*) = *gcd a b*
by (*simp add*: *gcd-commute*)**}**
qed

48.4 LCM defined by GCD

definition
lcm :: *nat* ⇒ *nat* ⇒ *nat*
where
lcm-def: *lcm m n* = *m* ∗ *n div gcd m n*

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

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

lemma *lcm-1* [*simp*]: *lcm m 1* = *m*
unfolding *lcm-def* **by** *simp*

lemma *lcm-0-left* [*simp*]: *lcm 0 n = 0*
unfolding *lcm-def* **by** *simp*

lemma *lcm-1-left* [*simp*]: *lcm 1 m = m*
unfolding *lcm-def* **by** *simp*

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

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

lemma *lcm-dvd1* [*iff*]:
m dvd lcm m n
proof (*cases m*)
case *0* **then show** *?thesis* **by** *simp*
next

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

lemma *lcm-dvd2* [*iff*]:
n dvd lcm m n
proof (*cases n*)
case *0* **then show** *?thesis* **by** *simp*
next
case (*Suc* -)
then have *npos*: $n > 0$ **by** *simp*
show *?thesis*
proof (*cases m*)
case *0* **then show** *?thesis* **by** *simp*
next
case (*Suc* -)
then have *mpos*: $m > 0$ **by** *simp*
have *gcd m n dvd m* **by** *simp*
then obtain *k* **where** $m = gcd\ m\ n * k$ **using** *dvd-def* **by** *auto*
then have $m * n\ div\ gcd\ m\ n = (gcd\ m\ n * k) * n\ div\ gcd\ m\ n$ **by** (*simp add*: *mult-ac*)
also have $\ldots = n * k$ **using** *mpos npos gcd-zero* **by** *simp*
finally show *?thesis* **by** (*simp add*: *lcm-def*)
qed
qed

lemma *gcd-add1-eq*: $gcd\ (m + k)\ k = gcd\ (m + k)\ m$
by (*simp add*: *gcd-commute*)

lemma *gcd-diff2*: $m \leq n \Longrightarrow gcd\ n\ (n - m) = gcd\ n\ m$
apply (*subgoal-tac* $n = m + (n - m)$)
apply (*erule ssubst*, *rule gcd-add1-eq*, *simp*)
done

48.5 GCD and LCM on integers

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

lemma *zgcd-zdvd1* [*iff*,*simp*, *algebra*]: *zgcd i j dvd i*
by (*simp add*: *zgcd-def int-dvd-iff*)

lemma *zgcd-zdvd2* [*iff*,*simp*, *algebra*]: *zgcd i j dvd j*
by (*simp add*: *zgcd-def int-dvd-iff*)

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

lemma *zgcd0* [*simp*,*algebra*]: (*zgcd i j* = *0*) = (*i* = *0* ∧ *j* = *0*)
by (*simp add*: *zgcd-def gcd-zero*)

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

lemma *zgcd-zminus* [*simp*, *algebra*]: *zgcd* (− *i*) *j* = *zgcd i j*
unfolding *zgcd-def* **by** *simp*

lemma *zgcd-zminus2* [*simp*, *algebra*]: *zgcd i* (− *j*) = *zgcd i j*
unfolding *zgcd-def* **by** *simp*

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

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

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

lemma *div-zgcd-relprime*:
assumes *nz*: *a* ≠ *0* ∨ *b* ≠ *0*
shows *zgcd* (*a div* (*zgcd a b*)) (*b div* (*zgcd a b*)) = *1*
proof −
from *nz* **have** *nz′*: *nat* |*a*| ≠ *0* ∨ *nat* |*b*| ≠ *0* **by** *arith*
let *?g* = *zgcd a b*
let *?a′* = *a div ?g*
let *?b′* = *b div ?g*
let *?g′* = *zgcd ?a′ ?b′*
have *dvdg*: *?g dvd a ?g dvd b* **by** (*simp-all add*: *zgcd-zdvd1 zgcd-zdvd2*)
have *dvdg′*: *?g′ dvd ?a′ ?g′ dvd ?b′* **by** (*simp-all add*: *zgcd-zdvd1 zgcd-zdvd2*)
from *dvdg dvdg′* **obtain** *ka kb ka′ kb′* **where**
kab: *a* = *?g*ka b* = *?g*kb ?a′* = *?g′*ka′ ?b′* = *?g′* ∗ *kb′*
unfolding *dvd-def* **by** *blast*
then have *?g*∗ ?a′* = (*?g* ∗ *?g′*) ∗ *ka′ ?g*∗ ?b′* = (*?g* ∗ *?g′*) ∗ *kb′* **by** *simp-all*
then have *dvdgg′*:*?g* ∗ *?g′ dvd a ?g*∗ *?g′ dvd b*
by (*auto simp add*: *zdvd-mult-div-cancel* [*OF dvdg*(*1*)]
zdvd-mult-div-cancel [*OF dvdg*(*2*)] *dvd-def*)
have *?g* ≠ *0* **using** *nz* **by** *simp*
then have *gp*: *?g* ≠ *0* **using** *zgcd-pos*[**where** *i*=*a* **and** *j*=*b*] **by** *arith*
from *zgcd-greatest* [*OF dvdgg′*] **have** *?g* ∗ *?g′ dvd ?g* **.**
with *zdvd-mult-cancel1* [*OF gp*] **have** |*?g′*| = *1* **by** *simp*
with *zgcd-pos* **show** *?g′* = *1* **by** *simp*
qed

lemma *zgcd-0* [*simp*, *algebra*]: *zgcd m 0* = *abs m*
by (*simp add*: *zgcd-def abs-if*)

lemma *zgcd-0-left* [*simp*, *algebra*]: *zgcd 0 m* = *abs m*
by (*simp add*: *zgcd-def abs-if*)

lemma *zgcd-non-0*: *0* < *n* ==> *zgcd m n* = *zgcd n* (*m mod n*)
apply (*frule-tac b* = *n* **and** *a* = *m* **in** *pos-mod-sign*)

apply (*simp del*: *pos-mod-sign add*: *zgcd-def abs-if nat-mod-distrib*)
apply (*auto simp add*: *gcd-non-0 nat-mod-distrib* [*symmetric*] *zmod-zminus1-eq-if*)
apply (*frule-tac a* = *m* **in** *pos-mod-bound*)
apply (*simp del*: *pos-mod-bound add*: *nat-diff-distrib gcd-diff2 nat-le-eq-zle*)
done

lemma *zgcd-eq*: *zgcd m n* = *zgcd n* (*m mod n*)
apply (*case-tac n* = *0*, *simp add*: *DIVISION-BY-ZERO*)
apply (*auto simp add*: *linorder-neq-iff zgcd-non-0*)
apply (*cut-tac m* = −*m* **and** *n* = −*n* **in** *zgcd-non-0*, *auto*)
done

lemma *zgcd-1* [*simp*, *algebra*]: *zgcd m 1* = *1*
by (*simp add*: *zgcd-def abs-if*)

lemma *zgcd-0-1-iff* [*simp*, *algebra*]: *zgcd 0 m* = *1* ⟷ |*m*| = *1*
by (*simp add*: *zgcd-def abs-if*)

lemma *zgcd-greatest-iff* [*algebra*]: *k dvd zgcd m n* = (*k dvd m* ∧ *k dvd n*)
by (*simp add*: *zgcd-def abs-if int-dvd-iff dvd-int-iff nat-dvd-iff*)

lemma *zgcd-1-left* [*simp*, *algebra*]: *zgcd 1 m* = *1*
by (*simp add*: *zgcd-def gcd-1-left*)

lemma *zgcd-assoc*: *zgcd* (*zgcd k m*) *n* = *zgcd k* (*zgcd m n*)
by (*simp add*: *zgcd-def gcd-assoc*)

lemma *zgcd-left-commute*: *zgcd k* (*zgcd m n*) = *zgcd m* (*zgcd k n*)
apply (*rule zgcd-commute* [*THEN trans*])
apply (*rule zgcd-assoc* [*THEN trans*])
apply (*rule zgcd-commute* [*THEN arg-cong*])
done

lemmas *zgcd-ac* = *zgcd-assoc zgcd-commute zgcd-left-commute*
— addition is an AC-operator

lemma *zgcd-zmult-distrib2*: *0* ≤ *k* ==> *k* ∗ *zgcd m n* = *zgcd* (*k* ∗ *m*) (*k* ∗ *n*)
by (*simp del*: *minus-mult-right* [*symmetric*]
add: *minus-mult-right nat-mult-distrib zgcd-def abs-if*
mult-less-0-iff gcd-mult-distrib2 [*symmetric*] *zmult-int* [*symmetric*])

lemma *zgcd-zmult-distrib2-abs*: *zgcd* (*k* ∗ *m*) (*k* ∗ *n*) = *abs k* ∗ *zgcd m n*
by (*simp add*: *abs-if zgcd-zmult-distrib2*)

lemma *zgcd-self* [*simp*]: *0* ≤ *m* ==> *zgcd m m* = *m*
by (*cut-tac k* = *m* **and** *m* = *1* **and** *n* = *1* **in** *zgcd-zmult-distrib2*, *simp-all*)

lemma *zgcd-zmult-eq-self* [*simp*]: *0* ≤ *k* ==> *zgcd k* (*k* ∗ *n*) = *k*
by (*cut-tac k* = *k* **and** *m* = *1* **and** *n* = *n* **in** *zgcd-zmult-distrib2*, *simp-all*)

lemma *zgcd-zmult-eq-self2* [*simp*]: *0* ≤ *k* ==> *zgcd* (*k* ∗ *n*) *k* = *k*
by (*cut-tac k* = *k* **and** *m* = *n* **and** *n* = *1* **in** *zgcd-zmult-distrib2*, *simp-all*)

definition *zlcm i j* = *int* (*lcm*(*nat*(*abs i*)) (*nat*(*abs j*)))

lemma *dvd-zlcm-self1*[*simp*, *algebra*]: *i dvd zlcm i j*
by(*simp add*:*zlcm-def dvd-int-iff*)

lemma *dvd-zlcm-self2*[*simp*, *algebra*]: *j dvd zlcm i j*
by(*simp add*:*zlcm-def dvd-int-iff*)

lemma *dvd-imp-dvd-zlcm1*:
assumes *k dvd i* **shows** *k dvd* (*zlcm i j*)
proof −
have *nat*(*abs k*) *dvd nat*(*abs i*) **using** ‹*k dvd i*›
by(*simp add*:*int-dvd-iff*[*symmetric*] *dvd-int-iff*[*symmetric*])
thus *?thesis* **by**(*simp add*:*zlcm-def dvd-int-iff*)(*blast intro*: *dvd-trans*)
qed

lemma *dvd-imp-dvd-zlcm2*:
assumes *k dvd j* **shows** *k dvd* (*zlcm i j*)
proof −
have *nat*(*abs k*) *dvd nat*(*abs j*) **using** ‹*k dvd j*›
by(*simp add*:*int-dvd-iff*[*symmetric*] *dvd-int-iff*[*symmetric*])
thus *?thesis* **by**(*simp add*:*zlcm-def dvd-int-iff*)(*blast intro*: *dvd-trans*)
qed

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

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

lemma *lcm-pos*:
assumes *mpos*: *m* > *0*
and *npos*: *n*>*0*
shows *lcm m n* > *0*
proof(*rule ccontr*, *simp add*: *lcm-def gcd-zero*)
assume *h*:*m*∗*n div gcd m n* = *0*
from *mpos npos* **have** *gcd m n* ≠ *0* **using** *gcd-zero* **by** *simp*
hence *gcdp*: *gcd m n* > *0* **by** *simp*
with *h*
have *m*∗*n* < *gcd m n*

by (*cases* $m * n < gcd\ m\ n$) (*auto simp add*: *div-if*[*OF gcdp*, **where** $m{=}m{*}n$])
moreover
have *gcd m n dvd m* **by** *simp*
with *mpos dvd-imp-le* **have** *t1*:$gcd\ m\ n \leq m$ **by** *simp*
with *npos* **have** *t1*:$gcd\ m\ n\ *n \leq m{*}n$ **by** *simp*
have $gcd\ m\ n \leq gcd\ m\ n{*}n$ **using** *npos* **by** *simp*
with *t1* **have** $gcd\ m\ n \leq m{*}n$ **by** *arith*
ultimately show *False* **by** *simp*
qed

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

lemma *zgcd-code* [*code*]:
$zgcd\ k\ l = |if\ l = 0\ then\ k\ else\ zgcd\ l\ (|k|\ mod\ |l|)|$
by (*simp add*: *zgcd-def gcd.simps* [*of nat* $|k|$] *nat-mod-distrib*)

end

49 Archimedean-Field: Archimedean Fields, Floor and Ceiling Functions

theory *Archimedean-Field*
imports *Main*
begin

49.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

class *archimedean-field* = *ordered-field* + *number-ring* +
assumes *ex-le-of-int*: $\exists z.\ x \leq of\text{-}int\ z$

lemma *ex-less-of-int*:
fixes $x :: 'a{::}archimedean\text{-}field$ **shows** $\exists z.\ x < of\text{-}int\ z$
proof −
from *ex-le-of-int* **obtain** z **where** $x \leq of\text{-}int\ z$ **..**
then have $x < of\text{-}int\ (z + 1)$ **by** *simp*

then show *?thesis* **..**
qed

lemma *ex-of-int-less*:
fixes x :: *'a*::*archimedean-field* **shows** $\exists z.$ *of-int* $z < x$
proof −
from *ex-less-of-int* **obtain** z **where** $-x <$ *of-int* z **..**
then have *of-int* $(-z) < x$ **by** *simp*
then show *?thesis* **..**
qed

lemma *ex-less-of-nat*:
fixes x :: *'a*::*archimedean-field* **shows** $\exists n.$ $x <$ *of-nat* n
proof −
obtain z **where** $x <$ *of-int* z **using** *ex-less-of-int* **..**
also have $\ldots \leq$ *of-int* (*int* (*nat* z)) **by** *simp*
also have $\ldots =$ *of-nat* (*nat* z) **by** (*simp only*: *of-int-of-nat-eq*)
finally show *?thesis* **..**
qed

lemma *ex-le-of-nat*:
fixes x :: *'a*::*archimedean-field* **shows** $\exists n.$ $x \leq$ *of-nat* n
proof −
obtain n **where** $x <$ *of-nat* n **using** *ex-less-of-nat* **..**
then have $x \leq$ *of-nat* n **by** *simp*
then show *?thesis* **..**
qed

Archimedean fields have no infinitesimal elements.

lemma *ex-inverse-of-nat-Suc-less*:
fixes x :: *'a*::*archimedean-field*
assumes $0 < x$ **shows** $\exists n.$ *inverse* (*of-nat* (*Suc* n)) $< x$
proof −
from ‹$0 < x$› **have** $0 <$ *inverse* x
by (*rule positive-imp-inverse-positive*)
obtain n **where** *inverse* $x <$ *of-nat* n
using *ex-less-of-nat* **..**
then obtain m **where** *inverse* $x <$ *of-nat* (*Suc* m)
using ‹$0 <$ *inverse* x› **by** (*cases* n) (*simp-all del*: *of-nat-Suc*)
then have *inverse* (*of-nat* (*Suc* m)) $<$ *inverse* (*inverse* x)
using ‹$0 <$ *inverse* x› **by** (*rule less-imp-inverse-less*)
then have *inverse* (*of-nat* (*Suc* m)) $< x$
using ‹$0 < x$› **by** (*simp add*: *nonzero-inverse-inverse-eq*)
then show *?thesis* **..**
qed

lemma *ex-inverse-of-nat-less*:
fixes x :: *'a*::*archimedean-field*
assumes $0 < x$ **shows** $\exists n{>}0.$ *inverse* (*of-nat* n) $< x$

using *ex-inverse-of-nat-Suc-less* [*OF* ‹*0 < x*›] **by** *auto*

lemma *ex-less-of-nat-mult*:
fixes *x* :: *'a*::*archimedean-field*
assumes *0 < x* **shows** ∃ *n*. *y < of-nat n * x*
proof −
obtain *n* **where** *y / x < of-nat n* **using** *ex-less-of-nat* **..**
with ‹*0 < x*› **have** *y < of-nat n * x* **by** (*simp add*: *pos-divide-less-eq*)
then show *?thesis* **..**
qed

49.2 Existence and uniqueness of floor function

lemma *exists-least-lemma*:
assumes ¬ *P 0* **and** ∃ *n*. *P n*
shows ∃ *n*. ¬ *P n* ∧ *P* (*Suc n*)
proof −
from ‹∃ *n*. *P n*› **have** *P* (*Least P*) **by** (*rule LeastI-ex*)
with ‹¬ *P 0*› **obtain** *n* **where** *Least P = Suc n*
by (*cases Least P*) *auto*
then have *n < Least P* **by** *simp*
then have ¬ *P n* **by** (*rule not-less-Least*)
then have ¬ *P n* ∧ *P* (*Suc n*)
using ‹*P* (*Least P*)› ‹*Least P = Suc n*› **by** *simp*
then show *?thesis* **..**
qed

lemma *floor-exists*:
fixes *x* :: *'a*::*archimedean-field*
shows ∃ *z*. *of-int z ≤ x* ∧ *x < of-int* (*z + 1*)
proof (*cases*)
assume *0 ≤ x*
then have ¬ *x < of-nat 0* **by** *simp*
then have ∃ *n*. ¬ *x < of-nat n* ∧ *x < of-nat* (*Suc n*)
using *ex-less-of-nat* **by** (*rule exists-least-lemma*)
then obtain *n* **where** ¬ *x < of-nat n* ∧ *x < of-nat* (*Suc n*) **..**
then have *of-int* (*int n*) ≤ *x* ∧ *x < of-int* (*int n + 1*) **by** *simp*
then show *?thesis* **..**
next
assume ¬ *0 ≤ x*
then have ¬ − *x ≤ of-nat 0* **by** *simp*
then have ∃ *n*. ¬ − *x ≤ of-nat n* ∧ − *x ≤ of-nat* (*Suc n*)
using *ex-le-of-nat* **by** (*rule exists-least-lemma*)
then obtain *n* **where** ¬ − *x ≤ of-nat n* ∧ − *x ≤ of-nat* (*Suc n*) **..**
then have *of-int* (− *int n* − *1*) ≤ *x* ∧ *x < of-int* (− *int n* − *1* + *1*) **by** *simp*
then show *?thesis* **..**
qed

lemma *floor-exists1*:

fixes $x :: 'a::archimedean\text{-}field$
shows $\exists!z.\ of\text{-}int\ z \leq x \wedge x < of\text{-}int\ (z + 1)$
proof (*rule ex-ex1I*)
show $\exists z.\ of\text{-}int\ z \leq x \wedge x < of\text{-}int\ (z + 1)$
by (*rule floor-exists*)
next
fix $y\ z$ **assume**
$of\text{-}int\ y \leq x \wedge x < of\text{-}int\ (y + 1)$
$of\text{-}int\ z \leq x \wedge x < of\text{-}int\ (z + 1)$
then have
$of\text{-}int\ y \leq x\ x < of\text{-}int\ (y + 1)$
$of\text{-}int\ z \leq x\ x < of\text{-}int\ (z + 1)$
by *simp-all*
from *le-less-trans* [*OF* ‹*of-int y* ≤ *x*› ‹*x* < *of-int* (*z* + 1)›]
le-less-trans [*OF* ‹*of-int z* ≤ *x*› ‹*x* < *of-int* (*y* + 1)›]
show $y = z$ **by** (*simp del*: *of-int-add*)
qed

49.3 Floor function

definition
$floor :: 'a::archimedean\text{-}field \Rightarrow int$ **where**
[*code del*]: $floor\ x = (THE\ z.\ of\text{-}int\ z \leq x \wedge x < of\text{-}int\ (z + 1))$

notation (*xsymbols*)
$floor\ (\lfloor\text{-}\rfloor)$

notation (*HTML* **output**)
$floor\ (\lfloor\text{-}\rfloor)$

lemma *floor-correct*: $of\text{-}int\ (floor\ x) \leq x \wedge x < of\text{-}int\ (floor\ x + 1)$
unfolding *floor-def* **using** *floor-exists1* **by** (*rule theI'*)

lemma *floor-unique*: $[\![of\text{-}int\ z \leq x;\ x < of\text{-}int\ z + 1]\!] \Longrightarrow floor\ x = z$
using *floor-correct* [*of x*] *floor-exists1* [*of x*] **by** *auto*

lemma *of-int-floor-le*: $of\text{-}int\ (floor\ x) \leq x$
using *floor-correct* **..**

lemma *le-floor-iff*: $z \leq floor\ x \longleftrightarrow of\text{-}int\ z \leq x$
proof
assume $z \leq floor\ x$
then have $(of\text{-}int\ z :: 'a) \leq of\text{-}int\ (floor\ x)$ **by** *simp*
also have $of\text{-}int\ (floor\ x) \leq x$ **by** (*rule of-int-floor-le*)
finally show $of\text{-}int\ z \leq x$ **.**
next
assume $of\text{-}int\ z \leq x$
also have $x < of\text{-}int\ (floor\ x + 1)$ **using** *floor-correct* **..**
finally show $z \leq floor\ x$ **by** (*simp del*: *of-int-add*)

qed

lemma *floor-less-iff*: *floor x* < *z* ⟷ *x* < *of-int z*
by (*simp add*: *not-le* [*symmetric*] *le-floor-iff*)

lemma *less-floor-iff*: *z* < *floor x* ⟷ *of-int z* + *1* ≤ *x*
using *le-floor-iff* [*of z* + *1 x*] **by** *auto*

lemma *floor-le-iff*: *floor x* ≤ *z* ⟷ *x* < *of-int z* + *1*
by (*simp add*: *not-less* [*symmetric*] *less-floor-iff*)

lemma *floor-mono*: **assumes** *x* ≤ *y* **shows** *floor x* ≤ *floor y*
proof −
have *of-int* (*floor x*) ≤ *x* **by** (*rule of-int-floor-le*)
also note ‹*x* ≤ *y*›
finally show *?thesis* **by** (*simp add*: *le-floor-iff*)
qed

lemma *floor-less-cancel*: *floor x* < *floor y* ⟹ *x* < *y*
by (*auto simp add*: *not-le* [*symmetric*] *floor-mono*)

lemma *floor-of-int* [*simp*]: *floor* (*of-int z*) = *z*
by (*rule floor-unique*) *simp-all*

lemma *floor-of-nat* [*simp*]: *floor* (*of-nat n*) = *int n*
using *floor-of-int* [*of of-nat n*] **by** *simp*

Floor with numerals

lemma *floor-zero* [*simp*]: *floor 0* = *0*
using *floor-of-int* [*of 0*] **by** *simp*

lemma *floor-one* [*simp*]: *floor 1* = *1*
using *floor-of-int* [*of 1*] **by** *simp*

lemma *floor-number-of* [*simp*]: *floor* (*number-of v*) = *number-of v*
using *floor-of-int* [*of number-of v*] **by** *simp*

lemma *zero-le-floor* [*simp*]: *0* ≤ *floor x* ⟷ *0* ≤ *x*
by (*simp add*: *le-floor-iff*)

lemma *one-le-floor* [*simp*]: *1* ≤ *floor x* ⟷ *1* ≤ *x*
by (*simp add*: *le-floor-iff*)

lemma *number-of-le-floor* [*simp*]: *number-of v* ≤ *floor x* ⟷ *number-of v* ≤ *x*
by (*simp add*: *le-floor-iff*)

lemma *zero-less-floor* [*simp*]: *0* < *floor x* ⟷ *1* ≤ *x*
by (*simp add*: *less-floor-iff*)

lemma *one-less-floor* [*simp*]: $1 < \mathit{floor}\ x \longleftrightarrow 2 \leq x$
by (*simp add*: *less-floor-iff*)

lemma *number-of-less-floor* [*simp*]:
$\mathit{number\text{-}of}\ v < \mathit{floor}\ x \longleftrightarrow \mathit{number\text{-}of}\ v + 1 \leq x$
by (*simp add*: *less-floor-iff*)

lemma *floor-le-zero* [*simp*]: $\mathit{floor}\ x \leq 0 \longleftrightarrow x < 1$
by (*simp add*: *floor-le-iff*)

lemma *floor-le-one* [*simp*]: $\mathit{floor}\ x \leq 1 \longleftrightarrow x < 2$
by (*simp add*: *floor-le-iff*)

lemma *floor-le-number-of* [*simp*]:
$\mathit{floor}\ x \leq \mathit{number\text{-}of}\ v \longleftrightarrow x < \mathit{number\text{-}of}\ v + 1$
by (*simp add*: *floor-le-iff*)

lemma *floor-less-zero* [*simp*]: $\mathit{floor}\ x < 0 \longleftrightarrow x < 0$
by (*simp add*: *floor-less-iff*)

lemma *floor-less-one* [*simp*]: $\mathit{floor}\ x < 1 \longleftrightarrow x < 1$
by (*simp add*: *floor-less-iff*)

lemma *floor-less-number-of* [*simp*]:
$\mathit{floor}\ x < \mathit{number\text{-}of}\ v \longleftrightarrow x < \mathit{number\text{-}of}\ v$
by (*simp add*: *floor-less-iff*)

Addition and subtraction of integers

lemma *floor-add-of-int* [*simp*]: $\mathit{floor}\ (x + \mathit{of\text{-}int}\ z) = \mathit{floor}\ x + z$
using *floor-correct* [*of x*] **by** (*simp add*: *floor-unique*)

lemma *floor-add-number-of* [*simp*]:
$\mathit{floor}\ (x + \mathit{number\text{-}of}\ v) = \mathit{floor}\ x + \mathit{number\text{-}of}\ v$
using *floor-add-of-int* [*of x number-of v*] **by** *simp*

lemma *floor-add-one* [*simp*]: $\mathit{floor}\ (x + 1) = \mathit{floor}\ x + 1$
using *floor-add-of-int* [*of x 1*] **by** *simp*

lemma *floor-diff-of-int* [*simp*]: $\mathit{floor}\ (x - \mathit{of\text{-}int}\ z) = \mathit{floor}\ x - z$
using *floor-add-of-int* [*of x − z*] **by** (*simp add*: *algebra-simps*)

lemma *floor-diff-number-of* [*simp*]:
$\mathit{floor}\ (x - \mathit{number\text{-}of}\ v) = \mathit{floor}\ x - \mathit{number\text{-}of}\ v$
using *floor-diff-of-int* [*of x number-of v*] **by** *simp*

lemma *floor-diff-one* [*simp*]: $\mathit{floor}\ (x - 1) = \mathit{floor}\ x - 1$
using *floor-diff-of-int* [*of x 1*] **by** *simp*

49.4 Ceiling function

definition
ceiling :: *'a*::*archimedean-field* ⇒ *int* **where**
[*code del*]: *ceiling x* = − *floor* (− *x*)

notation (*xsymbols*)
ceiling (⌈-⌉)

notation (*HTML* **output**)
ceiling (⌈-⌉)

lemma *ceiling-correct*: *of-int* (*ceiling x*) − *1* < *x* ∧ *x* ≤ *of-int* (*ceiling x*)
unfolding *ceiling-def* **using** *floor-correct* [*of* − *x*] **by** *simp*

lemma *ceiling-unique*: ⟦*of-int z* − *1* < *x*; *x* ≤ *of-int z*⟧ ⟹ *ceiling x* = *z*
unfolding *ceiling-def* **using** *floor-unique* [*of* − *z* − *x*] **by** *simp*

lemma *le-of-int-ceiling*: *x* ≤ *of-int* (*ceiling x*)
using *ceiling-correct* **..**

lemma *ceiling-le-iff*: *ceiling x* ≤ *z* ⟷ *x* ≤ *of-int z*
unfolding *ceiling-def* **using** *le-floor-iff* [*of* − *z* − *x*] **by** *auto*

lemma *less-ceiling-iff*: *z* < *ceiling x* ⟷ *of-int z* < *x*
by (*simp add*: *not-le* [*symmetric*] *ceiling-le-iff*)

lemma *ceiling-less-iff*: *ceiling x* < *z* ⟷ *x* ≤ *of-int z* − *1*
using *ceiling-le-iff* [*of x z* − *1*] **by** *simp*

lemma *le-ceiling-iff*: *z* ≤ *ceiling x* ⟷ *of-int z* − *1* < *x*
by (*simp add*: *not-less* [*symmetric*] *ceiling-less-iff*)

lemma *ceiling-mono*: *x* ≥ *y* ⟹ *ceiling x* ≥ *ceiling y*
unfolding *ceiling-def* **by** (*simp add*: *floor-mono*)

lemma *ceiling-less-cancel*: *ceiling x* < *ceiling y* ⟹ *x* < *y*
by (*auto simp add*: *not-le* [*symmetric*] *ceiling-mono*)

lemma *ceiling-of-int* [*simp*]: *ceiling* (*of-int z*) = *z*
by (*rule ceiling-unique*) *simp-all*

lemma *ceiling-of-nat* [*simp*]: *ceiling* (*of-nat n*) = *int n*
using *ceiling-of-int* [*of of-nat n*] **by** *simp*

Ceiling with numerals

lemma *ceiling-zero* [*simp*]: *ceiling 0* = *0*
using *ceiling-of-int* [*of 0*] **by** *simp*

lemma *ceiling-one* [*simp*]: *ceiling 1* = *1*

using *ceiling-of-int* [*of 1*] **by** *simp*

lemma *ceiling-number-of* [*simp*]: *ceiling* (*number-of v*) = *number-of v*
using *ceiling-of-int* [*of number-of v*] **by** *simp*

lemma *ceiling-le-zero* [*simp*]: *ceiling x* ≤ *0* ⟷ *x* ≤ *0*
by (*simp add*: *ceiling-le-iff*)

lemma *ceiling-le-one* [*simp*]: *ceiling x* ≤ *1* ⟷ *x* ≤ *1*
by (*simp add*: *ceiling-le-iff*)

lemma *ceiling-le-number-of* [*simp*]:
ceiling x ≤ *number-of v* ⟷ *x* ≤ *number-of v*
by (*simp add*: *ceiling-le-iff*)

lemma *ceiling-less-zero* [*simp*]: *ceiling x* < *0* ⟷ *x* ≤ −*1*
by (*simp add*: *ceiling-less-iff*)

lemma *ceiling-less-one* [*simp*]: *ceiling x* < *1* ⟷ *x* ≤ *0*
by (*simp add*: *ceiling-less-iff*)

lemma *ceiling-less-number-of* [*simp*]:
ceiling x < *number-of v* ⟷ *x* ≤ *number-of v* − *1*
by (*simp add*: *ceiling-less-iff*)

lemma *zero-le-ceiling* [*simp*]: *0* ≤ *ceiling x* ⟷ −*1* < *x*
by (*simp add*: *le-ceiling-iff*)

lemma *one-le-ceiling* [*simp*]: *1* ≤ *ceiling x* ⟷ *0* < *x*
by (*simp add*: *le-ceiling-iff*)

lemma *number-of-le-ceiling* [*simp*]:
number-of v ≤ *ceiling x*⟷ *number-of v* − *1* < *x*
by (*simp add*: *le-ceiling-iff*)

lemma *zero-less-ceiling* [*simp*]: *0* < *ceiling x* ⟷ *0* < *x*
by (*simp add*: *less-ceiling-iff*)

lemma *one-less-ceiling* [*simp*]: *1* < *ceiling x* ⟷ *1* < *x*
by (*simp add*: *less-ceiling-iff*)

lemma *number-of-less-ceiling* [*simp*]:
number-of v < *ceiling x* ⟷ *number-of v* < *x*
by (*simp add*: *less-ceiling-iff*)

Addition and subtraction of integers

lemma *ceiling-add-of-int* [*simp*]: *ceiling* (*x* + *of-int z*) = *ceiling x* + *z*
using *ceiling-correct* [*of x*] **by** (*simp add*: *ceiling-unique*)

lemma *ceiling-add-number-of* [*simp*]:
  ceiling (*x* + *number-of v*) = *ceiling x* + *number-of v*
 using *ceiling-add-of-int* [*of x number-of v*] **by** *simp*

lemma *ceiling-add-one* [*simp*]: *ceiling* (*x* + *1*) = *ceiling x* + *1*
 using *ceiling-add-of-int* [*of x 1*] **by** *simp*

lemma *ceiling-diff-of-int* [*simp*]: *ceiling* (*x* − *of-int z*) = *ceiling x* − *z*
 using *ceiling-add-of-int* [*of x* − *z*] **by** (*simp add*: *algebra-simps*)

lemma *ceiling-diff-number-of* [*simp*]:
 ceiling (*x* − *number-of v*) = *ceiling x* − *number-of v*
 using *ceiling-diff-of-int* [*of x number-of v*] **by** *simp*

lemma *ceiling-diff-one* [*simp*]: *ceiling* (*x* − *1*) = *ceiling x* − *1*
 using *ceiling-diff-of-int* [*of x 1*] **by** *simp*

49.5 Negation

lemma *floor-minus*: *floor* (− *x*) = − *ceiling x*
 unfolding *ceiling-def* **by** *simp*

lemma *ceiling-minus*: *ceiling* (− *x*) = − *floor x*
 unfolding *ceiling-def* **by** *simp*

end

50 Rational: Rational numbers

theory *Rational*
imports *GCD Archimedean-Field*
uses (*Tools/rat-arith.ML*)
begin

50.1 Rational numbers as quotient

50.1.1 Construction of the type of rational numbers

definition
 ratrel :: ((*int* × *int*) × (*int* × *int*)) *set* **where**
 ratrel = {(*x*, *y*). *snd x* ≠ *0* ∧ *snd y* ≠ *0* ∧ *fst x* ∗ *snd y* = *fst y* ∗ *snd x*}

lemma *ratrel-iff* [*simp*]:
 (*x*, *y*) ∈ *ratrel* ⟷ *snd x* ≠ *0* ∧ *snd y* ≠ *0* ∧ *fst x* ∗ *snd y* = *fst y* ∗ *snd x*
 by (*simp add*: *ratrel-def*)

lemma *refl-on-ratrel*: *refl-on* {*x*. *snd x* ≠ *0*} *ratrel*
 by (*auto simp add*: *refl-on-def ratrel-def*)

lemma *sym-ratrel*: *sym ratrel*
by (*simp add*: *ratrel-def sym-def*)

lemma *trans-ratrel*: *trans ratrel*
proof (*rule transI*, *unfold split-paired-all*)
fix *a b a' b' a'' b''* :: *int*
assume *A*: $((a, b), (a', b')) \in ratrel$
assume *B*: $((a', b'), (a'', b'')) \in ratrel$
have $b' * (a * b'') = b'' * (a * b')$ **by** *simp*
also from *A* **have** $a * b' = a' * b$ **by** *auto*
also have $b'' * (a' * b) = b * (a' * b'')$ **by** *simp*
also from *B* **have** $a' * b'' = a'' * b'$ **by** *auto*
also have $b * (a'' * b') = b' * (a'' * b)$ **by** *simp*
finally have $b' * (a * b'') = b' * (a'' * b)$ **.**
moreover from *B* **have** $b' \neq 0$ **by** *auto*
ultimately have $a * b'' = a'' * b$ **by** *simp*
with *A B* **show** $((a, b), (a'', b'')) \in ratrel$ **by** *auto*
qed

lemma *equiv-ratrel*: *equiv* $\{x.\ snd\ x \neq 0\}$ *ratrel*
by (*rule equiv.intro* [*OF refl-on-ratrel sym-ratrel trans-ratrel*])

lemmas *UN-ratrel* = *UN-equiv-class* [*OF equiv-ratrel*]
lemmas *UN-ratrel2* = *UN-equiv-class2* [*OF equiv-ratrel equiv-ratrel*]

lemma *equiv-ratrel-iff* [*iff*]:
assumes $snd\ x \neq 0$ **and** $snd\ y \neq 0$
shows $ratrel\ ``\ \{x\} = ratrel\ ``\ \{y\} \longleftrightarrow (x, y) \in ratrel$
by (*rule eq-equiv-class-iff*, *rule equiv-ratrel*) (*auto simp add*: *assms*)

typedef (*Rat*) *rat* = $\{x.\ snd\ x \neq 0\}$ // *ratrel*
proof
have (*0*::*int*, *1*::*int*) $\in \{x.\ snd\ x \neq 0\}$ **by** *simp*
then show $ratrel\ ``\ \{(0, 1)\} \in \{x.\ snd\ x \neq 0\}$ // *ratrel* **by** (*rule quotientI*)
qed

lemma *ratrel-in-Rat* [*simp*]: $snd\ x \neq 0 \Longrightarrow ratrel\ ``\ \{x\} \in Rat$
by (*simp add*: *Rat-def quotientI*)

declare *Abs-Rat-inject* [*simp*] *Abs-Rat-inverse* [*simp*]

50.1.2 Representation and basic operations

definition
Fract :: *int* ⇒ *int* ⇒ *rat* **where**
[*code del*]: *Fract a b* = *Abs-Rat* (*ratrel* `` {*if b* = *0 then* (*0*, *1*) *else* (*a*, *b*)})

code-datatype *Fract*

lemma *Rat-cases* [*case-names Fract*, *cases type*: *rat*]:
assumes $\bigwedge a\ b.\ q = Fract\ a\ b \Longrightarrow b \neq 0 \Longrightarrow C$
shows *C*
using *assms* **by** (*cases q*) (*clarsimp simp add*: *Fract-def Rat-def quotient-def*)

lemma *Rat-induct* [*case-names Fract*, *induct type*: *rat*]:
assumes $\bigwedge a\ b.\ b \neq 0 \Longrightarrow P\ (Fract\ a\ b)$
shows *P q*
using *assms* **by** (*cases q*) *simp*

lemma *eq-rat*:
shows $\bigwedge a\ b\ c\ d.\ b \neq 0 \Longrightarrow d \neq 0 \Longrightarrow Fract\ a\ b = Fract\ c\ d \longleftrightarrow a * d = c * b$
and $\bigwedge a.\ Fract\ a\ 0 = Fract\ 0\ 1$
and $\bigwedge a\ c.\ Fract\ 0\ a = Fract\ 0\ c$
by (*simp-all add*: *Fract-def*)

instantiation *rat* :: {*comm-ring-1*, *recpower*}
begin

definition
Zero-rat-def [*code*, *code unfold*]: *0 = Fract 0 1*

definition
One-rat-def [*code*, *code unfold*]: *1 = Fract 1 1*

definition
add-rat-def [*code del*]:
$q + r = Abs\text{-}Rat\ (\bigcup x \in Rep\text{-}Rat\ q.\ \bigcup y \in Rep\text{-}Rat\ r.$
*ratrel `` {(fst x * snd y + fst y * snd x, snd x * snd y)}*)

lemma *add-rat* [*simp*]:
assumes $b \neq 0$ **and** $d \neq 0$
shows $Fract\ a\ b + Fract\ c\ d = Fract\ (a * d + c * b)\ (b * d)$
proof −
have $(\lambda x\ y.\ ratrel``\{(fst\ x * snd\ y + fst\ y * snd\ x, snd\ x * snd\ y)\})$
respects2 ratrel
by (*rule equiv-ratrel* [*THEN congruent2-commuteI*]) (*simp-all add*: *left-distrib*)
with *assms* **show** *?thesis* **by** (*simp add*: *Fract-def add-rat-def UN-ratrel2*)
qed

definition
minus-rat-def [*code del*]:
$- q = Abs\text{-}Rat\ (\bigcup x \in Rep\text{-}Rat\ q.\ ratrel\ ``\ \{(-\ fst\ x, snd\ x)\})$

lemma *minus-rat* [*simp*, *code*]: $- Fract\ a\ b = Fract\ (-\ a)\ b$
proof −
have $(\lambda x.\ ratrel\ ``\ \{(-\ fst\ x, snd\ x)\})\ respects\ ratrel$
by (*simp add*: *congruent-def*)

then show *?thesis* **by** (*simp add*: *Fract-def minus-rat-def UN-ratrel*)
qed

lemma *minus-rat-cancel* [*simp*]: *Fract* (− *a*) (− *b*) = *Fract a b*
by (*cases b* = *0*) (*simp-all add*: *eq-rat*)

definition
diff-rat-def [*code del*]: *q* − *r* = *q* + − (*r*::*rat*)

lemma *diff-rat* [*simp*]:
assumes *b* ≠ *0* **and** *d* ≠ *0*
shows *Fract a b* − *Fract c d* = *Fract* (*a* ∗ *d* − *c* ∗ *b*) (*b* ∗ *d*)
using *assms* **by** (*simp add*: *diff-rat-def*)

definition
mult-rat-def [*code del*]:
q ∗ *r* = *Abs-Rat* (⋃ *x* ∈ *Rep-Rat q*. ⋃ *y* ∈ *Rep-Rat r*.
ratrel``{(*fst x* ∗ *fst y*, *snd x* ∗ *snd y*)})

lemma *mult-rat* [*simp*]: *Fract a b* ∗ *Fract c d* = *Fract* (*a* ∗ *c*) (*b* ∗ *d*)
proof −
have (λ*x y*. *ratrel* `` {(*fst x* ∗ *fst y*, *snd x* ∗ *snd y*)}) *respects2 ratrel*
by (*rule equiv-ratrel* [*THEN congruent2-commuteI*]) *simp-all*
then show *?thesis* **by** (*simp add*: *Fract-def mult-rat-def UN-ratrel2*)
qed

lemma *mult-rat-cancel*:
assumes *c* ≠ *0*
shows *Fract* (*c* ∗ *a*) (*c* ∗ *b*) = *Fract a b*
proof −
from *assms* **have** *Fract c c* = *Fract 1 1* **by** (*simp add*: *Fract-def*)
then show *?thesis* **by** (*simp add*: *mult-rat* [*symmetric*])
qed

primrec *power-rat*
where
q ^ *0* = (*1*::*rat*)
| *q* ^ *Suc n* = (*q*::*rat*) ∗ (*q* ^ *n*)

instance proof
fix *q r s* :: *rat* **show** (*q* ∗ *r*) ∗ *s* = *q* ∗ (*r* ∗ *s*)
by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-rat*)
next
fix *q r* :: *rat* **show** *q* ∗ *r* = *r* ∗ *q*
by (*cases q*, *cases r*) (*simp add*: *eq-rat*)
next
fix *q* :: *rat* **show** *1* ∗ *q* = *q*
by (*cases q*) (*simp add*: *One-rat-def eq-rat*)
next

fix *q r s* :: *rat* **show** (*q* + *r*) + *s* = *q* + (*r* + *s*)
by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-rat algebra-simps*)
next
fix *q r* :: *rat* **show** *q* + *r* = *r* + *q*
by (*cases q*, *cases r*) (*simp add*: *eq-rat*)
next
fix *q* :: *rat* **show** *0* + *q* = *q*
by (*cases q*) (*simp add*: *Zero-rat-def eq-rat*)
next
fix *q* :: *rat* **show** − *q* + *q* = *0*
by (*cases q*) (*simp add*: *Zero-rat-def eq-rat*)
next
fix *q r* :: *rat* **show** *q* − *r* = *q* + − *r*
by (*cases q*, *cases r*) (*simp add*: *eq-rat*)
next
fix *q r s* :: *rat* **show** (*q* + *r*) ∗ *s* = *q* ∗ *s* + *r* ∗ *s*
by (*cases q*, *cases r*, *cases s*) (*simp add*: *eq-rat algebra-simps*)
next
show (*0*::*rat*) ≠ *1* **by** (*simp add*: *Zero-rat-def One-rat-def eq-rat*)
next
fix *q* :: *rat* **show** *q* ∗ *1* = *q*
by (*cases q*) (*simp add*: *One-rat-def eq-rat*)
next
fix *q* :: *rat*
fix *n* :: *nat*
show *q* ^ *0* = *1* **by** *simp*
show *q* ^ (*Suc n*) = *q* ∗ (*q* ^ *n*) **by** *simp*
qed

declare *power-rat.simps* [*simp del*]

end

lemma *of-nat-rat*: *of-nat k* = *Fract* (*of-nat k*) *1*
by (*induct k*) (*simp-all add*: *Zero-rat-def One-rat-def*)

lemma *of-int-rat*: *of-int k* = *Fract k 1*
by (*cases k rule*: *int-diff-cases*) (*simp add*: *of-nat-rat*)

lemma *Fract-of-nat-eq*: *Fract* (*of-nat k*) *1* = *of-nat k*
by (*rule of-nat-rat* [*symmetric*])

lemma *Fract-of-int-eq*: *Fract k 1* = *of-int k*
by (*rule of-int-rat* [*symmetric*])

instantiation *rat* :: *number-ring*
begin

definition

rat-number-of-def [*code del*]: *number-of w* = *Fract w 1*

instance by *intro-classes* (*simp add*: *rat-number-of-def of-int-rat*)

end

lemma *rat-number-collapse* [*code post*]:
Fract 0 k = *0*
Fract 1 1 = *1*
Fract (*number-of k*) *1* = *number-of k*
Fract k 0 = *0*
by (*cases k* = *0*)
(*simp-all add*: *Zero-rat-def One-rat-def number-of-is-id number-of-eq of-int-rat eq-rat Fract-def*)

lemma *rat-number-expand* [*code unfold*]:
0 = *Fract 0 1*
1 = *Fract 1 1*
number-of k = *Fract* (*number-of k*) *1*
by (*simp-all add*: *rat-number-collapse*)

lemma *iszero-rat* [*simp*]:
iszero (*number-of k* :: *rat*) ⟷ *iszero* (*number-of k* :: *int*)
by (*simp add*: *iszero-def rat-number-expand number-of-is-id eq-rat*)

lemma *Rat-cases-nonzero* [*case-names Fract 0*]:
assumes *Fract*: ⋀*a b*. *q* = *Fract a b* ⟹ *b* ≠ *0* ⟹ *a* ≠ *0* ⟹ *C*
assumes *0*: *q* = *0* ⟹ *C*
shows *C*
proof (*cases q* = *0*)
case *True* **then show** *C* **using** *0* **by** *auto*
next
case *False*
then obtain *a b* **where** *q* = *Fract a b* **and** *b* ≠ *0* **by** (*cases q*) *auto*
moreover with *False* **have** *0* ≠ *Fract a b* **by** *simp*
with ‹*b* ≠ *0*› **have** *a* ≠ *0* **by** (*simp add*: *Zero-rat-def eq-rat*)
with *Fract* ‹*q* = *Fract a b*› ‹*b* ≠ *0*› **show** *C* **by** *auto*
qed

50.1.3 The field of rational numbers

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

definition
inverse-rat-def [*code del*]:
inverse q = *Abs-Rat* (⋃*x* ∈ *Rep-Rat q*.
ratrel `` {*if fst x* = *0 then* (*0*, *1*) *else* (*snd x*, *fst x*)})

lemma *inverse-rat* [*simp*]: *inverse* (*Fract a b*) = *Fract b a*
proof −
  have (λ*x*. *ratrel `` {if fst x = 0 then* (*0*, *1*) *else* (*snd x*, *fst x*)}) *respects ratrel*
    by (*auto simp add*: *congruent-def mult-commute*)
  then show *?thesis* **by** (*simp add*: *Fract-def inverse-rat-def UN-ratrel*)
qed

definition
  divide-rat-def [*code del*]: *q* / *r* = *q* ∗ *inverse* (*r*::*rat*)

lemma *divide-rat* [*simp*]: *Fract a b* / *Fract c d* = *Fract* (*a* ∗ *d*) (*b* ∗ *c*)
  by (*simp add*: *divide-rat-def*)

instance proof
  show *inverse 0* = (*0*::*rat*) **by** (*simp add*: *rat-number-expand*)
    (*simp add*: *rat-number-collapse*)
next
  fix *q* :: *rat*
  assume *q* ≠ *0*
  then show *inverse q* ∗ *q* = *1* **by** (*cases q rule*: *Rat-cases-nonzero*)
   (*simp-all add*: *mult-rat inverse-rat rat-number-expand eq-rat*)
next
  fix *q r* :: *rat*
  show *q* / *r* = *q* ∗ *inverse r* **by** (*simp add*: *divide-rat-def*)
qed

end

50.1.4 Various

lemma *Fract-add-one*: *n* ≠ *0* ==> *Fract* (*m* + *n*) *n* = *Fract m n* + *1*
  by (*simp add*: *rat-number-expand*)

lemma *Fract-of-int-quotient*: *Fract k l* = *of-int k* / *of-int l*
  by (*simp add*: *Fract-of-int-eq* [*symmetric*])

lemma *Fract-number-of-quotient* [*code post*]:
  Fract (*number-of k*) (*number-of l*) = *number-of k* / *number-of l*
  unfolding *Fract-of-int-quotient number-of-is-id number-of-eq* **..**

lemma *Fract-1-number-of* [*code post*]:
  Fract 1 (*number-of k*) = *1* / *number-of k*
  unfolding *Fract-of-int-quotient number-of-eq* **by** *simp*

50.1.5 The ordered field of rational numbers

instantiation *rat* :: *linorder*
begin

definition

```
  le-rat-def [code del]:
   q ≤ r ⟷ contents (⋃x ∈ Rep-Rat q. ⋃y ∈ Rep-Rat r.
     {(fst x * snd y) * (snd x * snd y) ≤ (fst y * snd x) * (snd x * snd y)})

lemma le-rat [simp]:
  assumes b ≠ 0 and d ≠ 0
  shows Fract a b ≤ Fract c d ⟷ (a * d) * (b * d) ≤ (c * b) * (b * d)
proof −
  have (λx y. {(fst x * snd y) * (snd x * snd y) ≤ (fst y * snd x) * (snd x * snd
y)})
    respects2 ratrel
  proof (clarsimp simp add: congruent2-def)
    fix a b a' b' c d c' d'::int
    assume neq: b ≠ 0  b' ≠ 0  d ≠ 0  d' ≠ 0
    assume eq1: a * b' = a' * b
    assume eq2: c * d' = c' * d

    let ?le = λa b c d. ((a * d) * (b * d) ≤ (c * b) * (b * d))
    {
      fix a b c d x :: int assume x: x ≠ 0
      have ?le a b c d = ?le (a * x) (b * x) c d
      proof −
        from x have 0 < x * x by (auto simp add: zero-less-mult-iff)
        hence ?le a b c d =
            ((a * d) * (b * d) * (x * x) ≤ (c * b) * (b * d) * (x * x))
          by (simp add: mult-le-cancel-right)
        also have ... = ?le (a * x) (b * x) c d
          by (simp add: mult-ac)
        finally show ?thesis .
      qed
    } note le-factor = this

    let ?D = b * d and ?D' = b' * d'
    from neq have D: ?D ≠ 0 by simp
    from neq have ?D' ≠ 0 by simp
    hence ?le a b c d = ?le (a * ?D') (b * ?D') c d
      by (rule le-factor)
    also have ... = ((a * b') * ?D * ?D' * d * d' ≤ (c * d') * ?D * ?D' * b * b')
      by (simp add: mult-ac)
    also have ... = ((a' * b) * ?D * ?D' * d * d' ≤ (c' * d) * ?D * ?D' * b * b')
      by (simp only: eq1 eq2)
    also have ... = ?le (a' * ?D) (b' * ?D) c' d'
      by (simp add: mult-ac)
    also from D have ... = ?le a' b' c' d'
      by (rule le-factor [symmetric])
    finally show ?le a b c d = ?le a' b' c' d' .
  qed
  with assms show ?thesis by (simp add: Fract-def le-rat-def UN-ratrel2)
qed
```

definition
less-rat-def [*code del*]: $z < (w::rat) \longleftrightarrow z \leq w \wedge z \neq w$

lemma *less-rat* [*simp*]:
assumes $b \neq 0$ **and** $d \neq 0$
shows *Fract a b* $<$ *Fract c d* $\longleftrightarrow$ $(a * d) * (b * d) < (c * b) * (b * d)$
using *assms* **by** (*simp add*: *less-rat-def eq-rat order-less-le*)

instance proof
fix *q r s* :: *rat*
{
assume $q \leq r$ **and** $r \leq s$
show $q \leq s$
proof (*insert prems*, *induct q*, *induct r*, *induct s*)
fix *a b c d e f* :: *int*
assume *neq*: $b \neq 0$ $d \neq 0$ $f \neq 0$
assume *1*: *Fract a b* $\leq$ *Fract c d* **and** *2*: *Fract c d* $\leq$ *Fract e f*
show *Fract a b* $\leq$ *Fract e f*
proof −
from *neq* **obtain** *bb*: $0 < b * b$ **and** *dd*: $0 < d * d$ **and** *ff*: $0 < f * f$
by (*auto simp add*: *zero-less-mult-iff linorder-neq-iff*)
have $(a * d) * (b * d) * (f * f) \leq (c * b) * (b * d) * (f * f)$
proof −
from *neq 1* **have** $(a * d) * (b * d) \leq (c * b) * (b * d)$
by *simp*
with *ff* **show** *?thesis* **by** (*simp add*: *mult-le-cancel-right*)
qed
also have ... $= (c * f) * (d * f) * (b * b)$ **by** *algebra*
also have ... $\leq (e * d) * (d * f) * (b * b)$
proof −
from *neq 2* **have** $(c * f) * (d * f) \leq (e * d) * (d * f)$
by *simp*
with *bb* **show** *?thesis* **by** (*simp add*: *mult-le-cancel-right*)
qed
finally have $(a * f) * (b * f) * (d * d) \leq e * b * (b * f) * (d * d)$
by (*simp only*: *mult-ac*)
with *dd* **have** $(a * f) * (b * f) \leq (e * b) * (b * f)$
by (*simp add*: *mult-le-cancel-right*)
with *neq* **show** *?thesis* **by** *simp*
qed
qed
next
assume $q \leq r$ **and** $r \leq q$
show $q = r$
proof (*insert prems*, *induct q*, *induct r*)
fix *a b c d* :: *int*
assume *neq*: $b \neq 0$ $d \neq 0$
assume *1*: *Fract a b* $\leq$ *Fract c d* **and** *2*: *Fract c d* $\leq$ *Fract a b*

```
      show Fract a b = Fract c d
      proof −
        from neq 1 have (a ∗ d) ∗ (b ∗ d) ≤ (c ∗ b) ∗ (b ∗ d)
          by simp
        also have ... ≤ (a ∗ d) ∗ (b ∗ d)
        proof −
          from neq 2 have (c ∗ b) ∗ (d ∗ b) ≤ (a ∗ d) ∗ (d ∗ b)
            by simp
          thus ?thesis by (simp only: mult-ac)
        qed
        finally have (a ∗ d) ∗ (b ∗ d) = (c ∗ b) ∗ (b ∗ d) .
        moreover from neq have b ∗ d ≠ 0 by simp
        ultimately have a ∗ d = c ∗ b by simp
        with neq show ?thesis by (simp add: eq-rat)
      qed
    qed
  next
    show q ≤ q
      by (induct q) simp
    show (q < r) = (q ≤ r ∧ ¬ r ≤ q)
      by (induct q, induct r) (auto simp add: le-less mult-commute)
    show q ≤ r ∨ r ≤ q
      by (induct q, induct r)
        (simp add: mult-commute, rule linorder-linear)
  }
qed

end

instantiation rat :: {distrib-lattice, abs-if, sgn-if}
begin

definition
  abs-rat-def [code del]: |q| = (if q < 0 then −q else (q::rat))

lemma abs-rat [simp, code]: |Fract a b| = Fract |a| |b|
  by (auto simp add: abs-rat-def zabs-def Zero-rat-def less-rat not-less le-less minus-rat
eq-rat zero-compare-simps)

definition
  sgn-rat-def [code del]: sgn (q::rat) = (if q = 0 then 0 else if 0 < q then 1 else −
1)

lemma sgn-rat [simp, code]: sgn (Fract a b) = of-int (sgn a ∗ sgn b)
  unfolding Fract-of-int-eq
  by (auto simp: zsgn-def sgn-rat-def Zero-rat-def eq-rat)
    (auto simp: rat-number-collapse not-less le-less zero-less-mult-iff )

definition
```

```
  (inf :: rat ⇒ rat ⇒ rat) = min

definition
  (sup :: rat ⇒ rat ⇒ rat) = max

instance by intro-classes
 (auto simp add: abs-rat-def sgn-rat-def min-max.sup-inf-distrib1 inf-rat-def sup-rat-def)

end

instance rat :: ordered-field
proof
  fix q r s :: rat
  show q ≤ r ==> s + q ≤ s + r
  proof (induct q, induct r, induct s)
    fix a b c d e f :: int
    assume neq: b ≠ 0  d ≠ 0  f ≠ 0
    assume le: Fract a b ≤ Fract c d
    show Fract e f + Fract a b ≤ Fract e f + Fract c d
    proof -
      let ?F = f * f from neq have F: 0 < ?F
        by (auto simp add: zero-less-mult-iff)
      from neq le have (a * d) * (b * d) ≤ (c * b) * (b * d)
        by simp
      with F have (a * d) * (b * d) * ?F * ?F ≤ (c * b) * (b * d) * ?F * ?F
        by (simp add: mult-le-cancel-right)
      with neq show ?thesis by (simp add: mult-ac int-distrib)
    qed
  qed
  show q < r ==> 0 < s ==> s * q < s * r
  proof (induct q, induct r, induct s)
    fix a b c d e f :: int
    assume neq: b ≠ 0  d ≠ 0  f ≠ 0
    assume le: Fract a b < Fract c d
    assume gt: 0 < Fract e f
    show Fract e f * Fract a b < Fract e f * Fract c d
    proof -
      let ?E = e * f and ?F = f * f
      from neq gt have 0 < ?E
        by (auto simp add: Zero-rat-def order-less-le eq-rat)
      moreover from neq have 0 < ?F
        by (auto simp add: zero-less-mult-iff)
      moreover from neq le have (a * d) * (b * d) < (c * b) * (b * d)
        by simp
      ultimately have (a * d) * (b * d) * ?E * ?F < (c * b) * (b * d) * ?E * ?F
        by (simp add: mult-less-cancel-right)
      with neq show ?thesis
        by (simp add: mult-ac)
    qed
```

```
  qed
qed auto

lemma Rat-induct-pos [case-names Fract, induct type: rat]:
  assumes step: ⋀a b. 0 < b ⟹ P (Fract a b)
  shows P q
proof (cases q)
  have step': ⋀a b. b < 0 ⟹ P (Fract a b)
  proof −
    fix a::int and b::int
    assume b: b < 0
    hence 0 < −b by simp
    hence P (Fract (−a) (−b)) by (rule step)
    thus P (Fract a b) by (simp add: order-less-imp-not-eq [OF b])
  qed
  case (Fract a b)
  thus P q by (force simp add: linorder-neq-iff step step')
qed

lemma zero-less-Fract-iff:
  0 < b ⟹ 0 < Fract a b ⟷ 0 < a
  by (simp add: Zero-rat-def zero-less-mult-iff)

lemma Fract-less-zero-iff:
  0 < b ⟹ Fract a b < 0 ⟷ a < 0
  by (simp add: Zero-rat-def mult-less-0-iff)

lemma zero-le-Fract-iff:
  0 < b ⟹ 0 ≤ Fract a b ⟷ 0 ≤ a
  by (simp add: Zero-rat-def zero-le-mult-iff)

lemma Fract-le-zero-iff:
  0 < b ⟹ Fract a b ≤ 0 ⟷ a ≤ 0
  by (simp add: Zero-rat-def mult-le-0-iff)

lemma one-less-Fract-iff:
  0 < b ⟹ 1 < Fract a b ⟷ b < a
  by (simp add: One-rat-def mult-less-cancel-right-disj)

lemma Fract-less-one-iff:
  0 < b ⟹ Fract a b < 1 ⟷ a < b
  by (simp add: One-rat-def mult-less-cancel-right-disj)

lemma one-le-Fract-iff:
  0 < b ⟹ 1 ≤ Fract a b ⟷ b ≤ a
  by (simp add: One-rat-def mult-le-cancel-right)

lemma Fract-le-one-iff:
  0 < b ⟹ Fract a b ≤ 1 ⟷ a ≤ b
```

by (*simp add*: *One-rat-def mult-le-cancel-right*)

50.1.6 Rationals are an Archimedean field

lemma *rat-floor-lemma*:
assumes $0 < b$
shows $of\text{-}int\ (a\ div\ b) \leq Fract\ a\ b \wedge Fract\ a\ b < of\text{-}int\ (a\ div\ b + 1)$
proof −
have $Fract\ a\ b = of\text{-}int\ (a\ div\ b) + Fract\ (a\ mod\ b)\ b$
using ‹$0 < b$› **by** (*simp add*: *of-int-rat*)
moreover have $0 \leq Fract\ (a\ mod\ b)\ b \wedge Fract\ (a\ mod\ b)\ b < 1$
using ‹$0 < b$› **by** (*simp add*: *zero-le-Fract-iff Fract-less-one-iff*)
ultimately show *?thesis* **by** *simp*
qed

instance *rat* :: *archimedean-field*
proof
fix r :: *rat*
show $\exists z.\ r \leq of\text{-}int\ z$
proof (*induct r*)
case (*Fract a b*)
then have $Fract\ a\ b \leq of\text{-}int\ (a\ div\ b + 1)$
using *rat-floor-lemma* [*of b a*] **by** *simp*
then show $\exists z.\ Fract\ a\ b \leq of\text{-}int\ z$ **..**
qed
qed

lemma *floor-Fract*:
assumes $0 < b$ **shows** $floor\ (Fract\ a\ b) = a\ div\ b$
using *rat-floor-lemma* [*OF* ‹$0 < b$›, *of a*]
by (*simp add*: *floor-unique*)

50.2 Arithmetic setup

use *Tools/rat-arith.ML*
declaration ⟪ *K rat-arith-setup* ⟫

50.3 Embedding from Rationals to other Fields

class *field-char-0* = *field* + *ring-char-0*

subclass (**in** *ordered-field*) *field-char-0* **..**

context *field-char-0*
begin

definition *of-rat* :: *rat* ⇒ *'a* **where**
[*code del*]: $of\text{-}rat\ q = contents\ (\bigcup(a,b) \in Rep\text{-}Rat\ q.\ \{of\text{-}int\ a\ /\ of\text{-}int\ b\})$

end

lemma *of-rat-congruent*:
(λ(*a*, *b*). {*of-int a* / *of-int b* :: *'a*::*field-char-0*}) *respects ratrel*
apply (*rule congruent.intro*)
apply (*clarsimp simp add*: *nonzero-divide-eq-eq nonzero-eq-divide-eq*)
apply (*simp only*: *of-int-mult* [*symmetric*])
done

lemma *of-rat-rat*: *b* ≠ *0* ⟹ *of-rat* (*Fract a b*) = *of-int a* / *of-int b*
unfolding *Fract-def of-rat-def* **by** (*simp add*: *UN-ratrel of-rat-congruent*)

lemma *of-rat-0* [*simp*]: *of-rat 0* = *0*
by (*simp add*: *Zero-rat-def of-rat-rat*)

lemma *of-rat-1* [*simp*]: *of-rat 1* = *1*
by (*simp add*: *One-rat-def of-rat-rat*)

lemma *of-rat-add*: *of-rat* (*a* + *b*) = *of-rat a* + *of-rat b*
by (*induct a*, *induct b*, *simp add*: *of-rat-rat add-frac-eq*)

lemma *of-rat-minus*: *of-rat* (− *a*) = − *of-rat a*
by (*induct a*, *simp add*: *of-rat-rat*)

lemma *of-rat-diff*: *of-rat* (*a* − *b*) = *of-rat a* − *of-rat b*
by (*simp only*: *diff-minus of-rat-add of-rat-minus*)

lemma *of-rat-mult*: *of-rat* (*a* ∗ *b*) = *of-rat a* ∗ *of-rat b*
apply (*induct a*, *induct b*, *simp add*: *of-rat-rat*)
apply (*simp add*: *divide-inverse nonzero-inverse-mult-distrib mult-ac*)
done

lemma *nonzero-of-rat-inverse*:
a ≠ *0* ⟹ *of-rat* (*inverse a*) = *inverse* (*of-rat a*)
apply (*rule inverse-unique* [*symmetric*])
apply (*simp add*: *of-rat-mult* [*symmetric*])
done

lemma *of-rat-inverse*:
(*of-rat* (*inverse a*)::*'a*::{*field-char-0*,*division-by-zero*}) =
inverse (*of-rat a*)
by (*cases a* = *0*, *simp-all add*: *nonzero-of-rat-inverse*)

lemma *nonzero-of-rat-divide*:
b ≠ *0* ⟹ *of-rat* (*a* / *b*) = *of-rat a* / *of-rat b*
by (*simp add*: *divide-inverse of-rat-mult nonzero-of-rat-inverse*)

lemma *of-rat-divide*:
(*of-rat* (*a* / *b*)::*'a*::{*field-char-0*,*division-by-zero*})
= *of-rat a* / *of-rat b*

by (*cases b = 0*) (*simp-all add*: *nonzero-of-rat-divide*)

lemma *of-rat-power*:
(*of-rat* (*a ^ n*)::*'a*::{*field-char-0*,*recpower*}) = *of-rat a ^ n*
by (*induct n*) (*simp-all add*: *of-rat-mult*)

lemma *of-rat-eq-iff* [*simp*]: (*of-rat a = of-rat b*) = (*a = b*)
apply (*induct a*, *induct b*)
apply (*simp add*: *of-rat-rat eq-rat*)
apply (*simp add*: *nonzero-divide-eq-eq nonzero-eq-divide-eq*)
apply (*simp only*: *of-int-mult* [*symmetric*] *of-int-eq-iff*)
done

lemma *of-rat-less*:
(*of-rat r* :: *'a*::*ordered-field*) < *of-rat s* ⟷ *r < s*
proof (*induct r*, *induct s*)
fix *a b c d* :: *int*
assume *not-zero*: *b > 0 d > 0*
then have *b * d > 0* **by** (*rule mult-pos-pos*)
have *of-int-divide-less-eq*:
(*of-int a* :: *'a*) / *of-int b* < *of-int c* / *of-int d*
⟷ (*of-int a* :: *'a*) * *of-int d* < *of-int c * of-int b*
using *not-zero* **by** (*simp add*: *pos-less-divide-eq pos-divide-less-eq*)
show (*of-rat* (*Fract a b*) :: *'a*::*ordered-field*) < *of-rat* (*Fract c d*)
⟷ *Fract a b < Fract c d*
using *not-zero* ‹*b * d > 0*›
by (*simp add*: *of-rat-rat of-int-divide-less-eq of-int-mult* [*symmetric*] *del*: *of-int-mult*)
qed

lemma *of-rat-less-eq*:
(*of-rat r* :: *'a*::*ordered-field*) ≤ *of-rat s* ⟷ *r ≤ s*
unfolding *le-less* **by** (*auto simp add*: *of-rat-less*)

lemmas *of-rat-eq-0-iff* [*simp*] = *of-rat-eq-iff* [*of - 0*, *simplified*]

lemma *of-rat-eq-id* [*simp*]: *of-rat = id*
proof
fix *a*
show *of-rat a = id a*
by (*induct a*)
(*simp add*: *of-rat-rat Fract-of-int-eq* [*symmetric*])
qed

Collapse nested embeddings

lemma *of-rat-of-nat-eq* [*simp*]: *of-rat* (*of-nat n*) = *of-nat n*
by (*induct n*) (*simp-all add*: *of-rat-add*)

lemma *of-rat-of-int-eq* [*simp*]: *of-rat* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*) (*simp add*: *of-rat-diff*)

lemma *of-rat-number-of-eq* [*simp*]:
of-rat (*number-of w*) = (*number-of w* :: *'a*::{*number-ring*,*field-char-0*})
by (*simp add*: *number-of-eq*)

lemmas *zero-rat* = *Zero-rat-def*
lemmas *one-rat* = *One-rat-def*

abbreviation
rat-of-nat :: *nat* ⇒ *rat*
where
rat-of-nat ≡ *of-nat*

abbreviation
rat-of-int :: *int* ⇒ *rat*
where
rat-of-int ≡ *of-int*

50.4 The Set of Rational Numbers

context *field-char-0*
begin

definition
Rats :: *'a set* **where**
[*code del*]: *Rats* = *range of-rat*

notation (*xsymbols*)
Rats (ℚ)

end

lemma *Rats-of-rat* [*simp*]: *of-rat r* ∈ *Rats*
by (*simp add*: *Rats-def*)

lemma *Rats-of-int* [*simp*]: *of-int z* ∈ *Rats*
by (*subst of-rat-of-int-eq* [*symmetric*], *rule Rats-of-rat*)

lemma *Rats-of-nat* [*simp*]: *of-nat n* ∈ *Rats*
by (*subst of-rat-of-nat-eq* [*symmetric*], *rule Rats-of-rat*)

lemma *Rats-number-of* [*simp*]:
(*number-of w*::*'a*::{*number-ring*,*field-char-0*}) ∈ *Rats*
by (*subst of-rat-number-of-eq* [*symmetric*], *rule Rats-of-rat*)

lemma *Rats-0* [*simp*]: *0* ∈ *Rats*
apply (*unfold Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-0* [*symmetric*])

done

lemma *Rats-1* [*simp*]: *1* ∈ *Rats*
apply (*unfold Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-1* [*symmetric*])
done

lemma *Rats-add* [*simp*]: ⟦*a* ∈ *Rats*; *b* ∈ *Rats*⟧ ⟹ *a* + *b* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-add* [*symmetric*])
done

lemma *Rats-minus* [*simp*]: *a* ∈ *Rats* ⟹ − *a* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-minus* [*symmetric*])
done

lemma *Rats-diff* [*simp*]: ⟦*a* ∈ *Rats*; *b* ∈ *Rats*⟧ ⟹ *a* − *b* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-diff* [*symmetric*])
done

lemma *Rats-mult* [*simp*]: ⟦*a* ∈ *Rats*; *b* ∈ *Rats*⟧ ⟹ *a* ∗ *b* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-mult* [*symmetric*])
done

lemma *nonzero-Rats-inverse*:
fixes *a* :: *'a*::*field-char-0*
shows ⟦*a* ∈ *Rats*; *a* ≠ *0*⟧ ⟹ *inverse a* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*erule nonzero-of-rat-inverse* [*symmetric*])
done

lemma *Rats-inverse* [*simp*]:
fixes *a* :: *'a*::{*field-char-0*,*division-by-zero*}
shows *a* ∈ *Rats* ⟹ *inverse a* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-inverse* [*symmetric*])
done

lemma *nonzero-Rats-divide*:

fixes *a b* :: *'a*::*field-char-0*
shows ⟦*a* ∈ *Rats*; *b* ∈ *Rats*; *b* ≠ *0*⟧ ⟹ *a* / *b* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*erule nonzero-of-rat-divide* [*symmetric*])
done

lemma *Rats-divide* [*simp*]:
fixes *a b* :: *'a*::{*field-char-0*,*division-by-zero*}
shows ⟦*a* ∈ *Rats*; *b* ∈ *Rats*⟧ ⟹ *a* / *b* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-divide* [*symmetric*])
done

lemma *Rats-power* [*simp*]:
fixes *a* :: *'a*::{*field-char-0*,*recpower*}
shows *a* ∈ *Rats* ⟹ *a* ^ *n* ∈ *Rats*
apply (*auto simp add*: *Rats-def*)
apply (*rule range-eqI*)
apply (*rule of-rat-power* [*symmetric*])
done

lemma *Rats-cases* [*cases set*: *Rats*]:
assumes *q* ∈ ℚ
obtains (*of-rat*) *r* **where** *q* = *of-rat r*
unfolding *Rats-def*
proof −
from ‹*q* ∈ ℚ› **have** *q* ∈ *range of-rat* **unfolding** *Rats-def* **.**
then obtain *r* **where** *q* = *of-rat r* **..**
then show *thesis* **..**
qed

lemma *Rats-induct* [*case-names of-rat*, *induct set*: *Rats*]:
q ∈ ℚ ⟹ (⋀*r*. *P* (*of-rat r*)) ⟹ *P q*
by (*rule Rats-cases*) *auto*

50.5 Implementation of rational numbers as pairs of integers

lemma *Fract-norm*: *Fract* (*a div zgcd a b*) (*b div zgcd a b*) = *Fract a b*
proof (*cases a* = *0* ∨ *b* = *0*)
case *True* **then show** *?thesis* **by** (*auto simp add*: *eq-rat*)
next
let *?c* = *zgcd a b*
case *False* **then have** *a* ≠ *0* **and** *b* ≠ *0* **by** *auto*
then have *?c* ≠ *0* **by** *simp*
then have *Fract ?c ?c* = *Fract 1 1* **by** (*simp add*: *eq-rat*)
moreover have *Fract* (*a div ?c* ∗ *?c* + *a mod ?c*) (*b div ?c* ∗ *?c* + *b mod ?c*)
= *Fract a b*

by (*simp add*: *semiring-div-class.mod-div-equality*)
moreover have *a mod ?c = 0* **by** (*simp add*: *dvd-eq-mod-eq-0* [*symmetric*])
moreover have *b mod ?c = 0* **by** (*simp add*: *dvd-eq-mod-eq-0* [*symmetric*])
ultimately show *?thesis*
by (*simp add*: *mult-rat* [*symmetric*])
qed

definition *Fract-norm* :: *int* ⇒ *int* ⇒ *rat* **where**
[*simp*, *code del*]: *Fract-norm a b = Fract a b*

lemma *Fract-norm-code* [*code*]: *Fract-norm a b* = (*if a = 0* ∨ *b = 0 then 0 else let c = zgcd a b in*
if b > 0 then Fract (*a div c*) (*b div c*) *else Fract* (− (*a div c*)) (− (*b div c*)))
by (*simp add*: *eq-rat Zero-rat-def Let-def Fract-norm*)

lemma [*code*]:
of-rat (*Fract a b*) = (*if b ≠ 0 then of-int a / of-int b else 0*)
by (*cases b = 0*) (*simp-all add*: *rat-number-collapse of-rat-rat*)

instantiation *rat* :: *eq*
begin

definition [*code del*]: *eq-class.eq* (*a*::*rat*) *b* ⟷ *a* − *b = 0*

instance by *default* (*simp add*: *eq-rat-def*)

lemma *rat-eq-code* [*code*]:
eq-class.eq (*Fract a b*) (*Fract c d*) ⟷ (*if b = 0*
then c = 0 ∨ *d = 0*
else if d = 0
then a = 0 ∨ *b = 0*
else a ∗ *d = b* ∗ *c*)
by (*auto simp add*: *eq eq-rat*)

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

end

lemma *le-rat'*:
assumes *b ≠ 0*
and *d ≠ 0*
shows *Fract a b* ≤ *Fract c d* ⟷ *a* ∗ |*d*| ∗ *sgn b* ≤ *c* ∗ |*b*| ∗ *sgn d*
proof −
have *abs-sgn*: ⋀*k*::*int*. |*k*| = *k* ∗ *sgn k* **unfolding** *abs-if sgn-if* **by** *simp*
have *a* ∗ *d* ∗ (*b* ∗ *d*) ≤ *c* ∗ *b* ∗ (*b* ∗ *d*) ⟷ *a* ∗ *d* ∗ (*sgn b* ∗ *sgn d*) ≤ *c* ∗ *b* ∗ (*sgn b* ∗ *sgn d*)
proof (*cases b* ∗ *d > 0*)

```
    case True
    moreover from True have sgn b * sgn d = 1
      by (simp add: sgn-times [symmetric] sgn-1-pos)
    ultimately show ?thesis by (simp add: mult-le-cancel-right)
  next
    case False with assms have b * d < 0 by (simp add: less-le)
    moreover from this have sgn b * sgn d = - 1
      by (simp only: sgn-times [symmetric] sgn-1-neg)
    ultimately show ?thesis by (simp add: mult-le-cancel-right)
  qed
  also have ... ⟷ a * |d| * sgn b ≤ c * |b| * sgn d
    by (simp add: abs-sgn mult-ac)
  finally show ?thesis using assms by simp
qed

lemma less-rat':
  assumes b ≠ 0
    and d ≠ 0
  shows Fract a b < Fract c d ⟷ a * |d| * sgn b < c * |b| * sgn d
proof -
  have abs-sgn: ⋀k::int. |k| = k * sgn k unfolding abs-if sgn-if by simp
  have a * d * (b * d) < c * b * (b * d) ⟷ a * d * (sgn b * sgn d) < c * b *
(sgn b * sgn d)
  proof (cases b * d > 0)
    case True
    moreover from True have sgn b * sgn d = 1
      by (simp add: sgn-times [symmetric] sgn-1-pos)
    ultimately show ?thesis by (simp add: mult-less-cancel-right)
  next
    case False with assms have b * d < 0 by (simp add: less-le)
    moreover from this have sgn b * sgn d = - 1
      by (simp only: sgn-times [symmetric] sgn-1-neg)
    ultimately show ?thesis by (simp add: mult-less-cancel-right)
  qed
  also have ... ⟷ a * |d| * sgn b < c * |b| * sgn d
    by (simp add: abs-sgn mult-ac)
  finally show ?thesis using assms by simp
qed

lemma (in ordered-idom) sgn-greater [simp]:
  0 < sgn a ⟷ 0 < a
  unfolding sgn-if by auto

lemma (in ordered-idom) sgn-less [simp]:
  sgn a < 0 ⟷ a < 0
  unfolding sgn-if by auto

lemma rat-le-eq-code [code]:
  Fract a b < Fract c d ⟷ (if b = 0
```

*then sgn c * sgn d > 0*
else if d = 0
*then sgn a * sgn b < 0*
*else a * |d| * sgn b < c * |b| * sgn d)*
by (*auto simp add*: *sgn-times mult-less-0-iff zero-less-mult-iff less-rat' eq-rat simp del*: *less-rat*)

lemma *rat-less-eq-code* [*code*]:
Fract a b ≤ Fract c d ⟷ (if b = 0
*then sgn c * sgn d ≥ 0*
else if d = 0
*then sgn a * sgn b ≤ 0*
*else a * |d| * sgn b ≤ c * |b| * sgn d)*
by (*auto simp add*: *sgn-times mult-le-0-iff zero-le-mult-iff le-rat' eq-rat simp del*: *le-rat*)
(*auto simp add*: *le-less not-less sgn-0-0*)

lemma *rat-plus-code* [*code*]:
Fract a b + Fract c d = (if b = 0
then Fract c d
else if d = 0
then Fract a b
*else Fract-norm (a * d + c * b) (b * d))*
by (*simp add*: *eq-rat*, *simp add*: *Zero-rat-def*)

lemma *rat-times-code* [*code*]:
*Fract a b * Fract c d = Fract-norm (a * c) (b * d)*
by *simp*

lemma *rat-minus-code* [*code*]:
Fract a b − Fract c d = (if b = 0
then Fract (− c) d
else if d = 0
then Fract a b
*else Fract-norm (a * d − c * b) (b * d))*
by (*simp add*: *eq-rat*, *simp add*: *Zero-rat-def*)

lemma *rat-inverse-code* [*code*]:
inverse (Fract a b) = (if b = 0 then Fract 1 0
else if a < 0 then Fract (− b) (− a)
else Fract b a)
by (*simp add*: *eq-rat*)

lemma *rat-divide-code* [*code*]:
*Fract a b / Fract c d = Fract-norm (a * d) (b * c)*
by *simp*

hide (**open**) *const Fract-norm*

Setup for SML code generator

```
types-code
  rat ((int */ int))
attach (term-of) ⟪
fun term-of-rat (p, q) =
  let
    val rT = Type (Rational.rat, [])
  in
    if q = 1 orelse p = 0 then HOLogic.mk-number rT p
    else @{term op / :: rat ⇒ rat ⇒ rat} $
      HOLogic.mk-number rT p $ HOLogic.mk-number rT q
  end;
⟫
attach (test) ⟪
fun gen-rat i =
  let
    val p = random-range 0 i;
    val q = random-range 1 (i + 1);
    val g = Integer.gcd p q;
    val p' = p div g;
    val q' = q div g;
    val r = (if one-of [true, false] then p' else ~ p',
      if p' = 0 then 0 else q')
  in
    (r, fn () => term-of-rat r)
  end;
⟫

consts-code
  Fract ((-,/ -))

consts-code
  of-int :: int ⇒ rat (⟨module⟩rat'-of'-int)
attach ⟪
fun rat-of-int 0 = (0, 0)
  | rat-of-int i = (i, 1);
⟫

end
```

51 PReal: Positive real numbers

```
theory PReal
imports Rational
begin
```

Could be generalized and moved to *Ring-and-Field*

lemma *add-eq-exists*: $\exists\, x.\ a+x = (b::rat)$
by (*rule-tac* $x=b-a$ **in** *exI*, *simp*)

definition
cut :: *rat set* => *bool* **where**
[*code del*]: *cut* $A = (\{\} \subset A\ \&$
$A < \{r.\ 0 < r\}\ \&$
$(\forall\, y \in A.\ ((\forall\, z.\ 0<z\ \&\ z < y \longrightarrow z \in A)\ \&\ (\exists\, u \in A.\ y < u))))$

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

lemma *cut-of-rat*:
assumes *q*: $0 < q$ **shows** *cut* $\{r::rat.\ 0 < r\ \&\ r < q\}$ (**is** *cut ?A*)
proof −
from *q* **have** *pos*: $?A < \{r.\ 0 < r\}$ **by** *force*
have *nonempty*: $\{\} \subset ?A$
proof
show $\{\} \subseteq ?A$ **by** *simp*
show $\{\} \neq ?A$
by (*force simp only*: *q eq-commute* [*of* {}] *interval-empty-iff*)
qed
show *?thesis*
by (*simp add*: *cut-def pos nonempty*,
blast dest: *dense intro*: *order-less-trans*)
qed

typedef *preal* = $\{A.\ cut\ A\}$
by (*blast intro*: *cut-of-rat* [*OF zero-less-one*])

definition
preal-of-rat :: *rat* => *preal* **where**
preal-of-rat q = *Abs-preal* $\{x::rat.\ 0 < x\ \&\ x < q\}$

definition
psup :: *preal set* => *preal* **where**
psup P = *Abs-preal* $(\bigcup X \in P.\ Rep\text{-}preal\ X)$

definition
add-set :: [*rat set*,*rat set*] => *rat set* **where**
add-set A B = $\{w.\ \exists\, x \in A.\ \exists\, y \in B.\ w = x + y\}$

definition
diff-set :: [*rat set*,*rat set*] => *rat set* **where**
[*code del*]: *diff-set A B* = $\{w.\ \exists\, x.\ 0 < w\ \&\ 0 < x\ \&\ x \notin B\ \&\ x + w \in A\}$

definition
mult-set :: [*rat set*,*rat set*] => *rat set* **where**
mult-set A B = {*w*. ∃ *x* ∈ *A*. ∃ *y* ∈ *B*. *w* = *x* ∗ *y*}

definition
inverse-set :: *rat set* => *rat set* **where**
[*code del*]: *inverse-set A* = {*x*. ∃ *y*. *0* < *x* & *x* < *y* & *inverse y* ∉ *A*}

instantiation *preal* :: {*ord*, *plus*, *minus*, *times*, *inverse*, *one*}
begin

definition
preal-less-def [*code del*]:
R < *S* == *Rep-preal R* < *Rep-preal S*

definition
preal-le-def [*code del*]:
R ≤ *S* == *Rep-preal R* ⊆ *Rep-preal S*

definition
preal-add-def:
R + *S* == *Abs-preal* (*add-set* (*Rep-preal R*) (*Rep-preal S*))

definition
preal-diff-def:
R − *S* == *Abs-preal* (*diff-set* (*Rep-preal R*) (*Rep-preal S*))

definition
preal-mult-def:
R ∗ *S* == *Abs-preal* (*mult-set* (*Rep-preal R*) (*Rep-preal S*))

definition
preal-inverse-def:
inverse R == *Abs-preal* (*inverse-set* (*Rep-preal R*))

definition *R* / *S* = *R* ∗ *inverse* (*S*::*preal*)

definition
preal-one-def:
1 == *preal-of-rat 1*

instance ..

end

Reduces equality on abstractions to equality on representatives

declare *Abs-preal-inject* [*simp*]
declare *Abs-preal-inverse* [*simp*]

lemma *rat-mem-preal*: $0 < q \Longrightarrow \{r::rat.\ 0 < r \ \&\ r < q\} \in preal$
by (*simp add*: *preal-def cut-of-rat*)

lemma *preal-nonempty*: $A \in preal \Longrightarrow \exists x \in A.\ 0 < x$
by (*unfold preal-def cut-def*, *blast*)

lemma *preal-Ex-mem*: $A \in preal \Longrightarrow \exists x.\ x \in A$
by (*drule preal-nonempty*, *fast*)

lemma *preal-imp-psubset-positives*: $A \in preal \Longrightarrow A < \{r.\ 0 < r\}$
by (*force simp add*: *preal-def cut-def*)

lemma *preal-exists-bound*: $A \in preal \Longrightarrow \exists x.\ 0 < x \ \&\ x \notin A$
by (*drule preal-imp-psubset-positives*, *auto*)

lemma *preal-exists-greater*: $[|\ A \in preal;\ y \in A\ |] \Longrightarrow \exists u \in A.\ y < u$
by (*unfold preal-def cut-def*, *blast*)

lemma *preal-downwards-closed*: $[|\ A \in preal;\ y \in A;\ 0 < z;\ z < y\ |] \Longrightarrow z \in A$
by (*unfold preal-def cut-def*, *blast*)

Relaxing the final premise

lemma *preal-downwards-closed'*:
$[|\ A \in preal;\ y \in A;\ 0 < z;\ z \leq y\ |] \Longrightarrow z \in A$
apply (*simp add*: *order-le-less*)
apply (*blast intro*: *preal-downwards-closed*)
done

A positive fraction not in a positive real is an upper bound. Gleason p. 122 - Remark (1)

lemma *not-in-preal-ub*:
assumes *A*: $A \in preal$
and *notx*: $x \notin A$
and *y*: $y \in A$
and *pos*: $0 < x$
shows $y < x$
proof (*cases rule*: *linorder-cases*)
assume $x<y$
with *notx* **show** *?thesis*
by (*simp add*: *preal-downwards-closed* [*OF A y*] *pos*)
next
assume $x=y$
with *notx* **and** *y* **show** *?thesis* **by** *simp*
next
assume $y<x$
thus *?thesis* **.**
qed

preal lemmas instantiated to *Rep-preal X*

lemma *mem-Rep-preal-Ex*: $\exists\, x.\ x \in \mathit{Rep\text{-}preal}\ X$
by (*rule preal-Ex-mem* [*OF Rep-preal*])

lemma *Rep-preal-exists-bound*: $\exists\, x{>}0.\ x \notin \mathit{Rep\text{-}preal}\ X$
by (*rule preal-exists-bound* [*OF Rep-preal*])

lemmas *not-in-Rep-preal-ub* = *not-in-preal-ub* [*OF Rep-preal*]

51.1 *preal-of-prat*: the Injection from prat to preal

lemma *rat-less-set-mem-preal*: $0 < y \Longrightarrow \{u{::}\mathit{rat}.\ 0 < u\ \&\ u < y\} \in \mathit{preal}$
by (*simp add*: *preal-def cut-of-rat*)

lemma *rat-subset-imp-le*:
$[|\{u{::}\mathit{rat}.\ 0 < u\ \&\ u < x\} \subseteq \{u.\ 0 < u\ \&\ u < y\};\ 0{<}x|] \Longrightarrow x \le y$
apply (*simp add*: *linorder-not-less* [*symmetric*])
apply (*blast dest*: *dense intro*: *order-less-trans*)
done

lemma *rat-set-eq-imp-eq*:
$[|\{u{::}\mathit{rat}.\ 0 < u\ \&\ u < x\} = \{u.\ 0 < u\ \&\ u < y\};$
$0 < x;\ 0 < y|] \Longrightarrow x = y$
by (*blast intro*: *rat-subset-imp-le order-antisym*)

51.2 Properties of Ordering

instance *preal* :: *order*
proof
fix w :: *preal*
show $w \le w$ **by** (*simp add*: *preal-le-def*)
next
fix $i\ j\ k$ :: *preal*
assume $i \le j$ **and** $j \le k$
then show $i \le k$ **by** (*simp add*: *preal-le-def*)
next
fix $z\ w$ :: *preal*
assume $z \le w$ **and** $w \le z$
then show $z = w$ **by** (*simp add*: *preal-le-def Rep-preal-inject*)
next
fix $z\ w$ :: *preal*
show $z < w \longleftrightarrow z \le w \wedge \neg\ w \le z$
by (*auto simp add*: *preal-le-def preal-less-def Rep-preal-inject*)
qed

lemma *preal-imp-pos*: $[|A \in \mathit{preal};\ r \in A|] \Longrightarrow 0 < r$
by (*insert preal-imp-psubset-positives*, *blast*)

instance *preal* :: *linorder*
proof
fix $x\ y$ :: *preal*

show $x <= y \mid y <= x$
apply (*auto simp add*: *preal-le-def*)
apply (*rule ccontr*)
apply (*blast dest*: *not-in-Rep-preal-ub intro*: *preal-imp-pos* [*OF Rep-preal*] *elim*: *order-less-asym*)
done
qed

instantiation *preal* :: *distrib-lattice*
begin

definition
(*inf* :: *preal* ⇒ *preal* ⇒ *preal*) = *min*

definition
(*sup* :: *preal* ⇒ *preal* ⇒ *preal*) = *max*

instance
by *intro-classes*
(*auto simp add*: *inf-preal-def sup-preal-def min-max.sup-inf-distrib1*)

end

51.3 Properties of Addition

lemma *preal-add-commute*: (*x*::*preal*) + *y* = *y* + *x*
apply (*unfold preal-add-def add-set-def*)
apply (*rule-tac f* = *Abs-preal* **in** *arg-cong*)
apply (*force simp add*: *add-commute*)
done

Lemmas for proving that addition of two positive reals gives a positive real

lemma *empty-psubset-nonempty*: *a* ∈ *A* ==> {} ⊂ *A*
by *blast*

Part 1 of Dedekind sections definition

lemma *add-set-not-empty*:
[|*A* ∈ *preal*; *B* ∈ *preal*|] ==> {} ⊂ *add-set A B*
apply (*drule preal-nonempty*)+
apply (*auto simp add*: *add-set-def*)
done

Part 2 of Dedekind sections definition. A structured version of this proof is *preal-not-mem-mult-set-Ex* below.

lemma *preal-not-mem-add-set-Ex*:
[|*A* ∈ *preal*; *B* ∈ *preal*|] ==> ∃ *q*>*0*. *q* ∉ *add-set A B*
apply (*insert preal-exists-bound* [*of A*] *preal-exists-bound* [*of B*], *auto*)
apply (*rule-tac x* = *x*+*xa* **in** *exI*)
apply (*simp add*: *add-set-def*, *clarify*)

apply (*drule* (*3*) *not-in-preal-ub*)+
apply (*force dest*: *add-strict-mono*)
done

lemma *add-set-not-rat-set*:
  assumes *A*: $A \in preal$
    and *B*: $B \in preal$
  shows *add-set A B* $< \{r.\ 0 < r\}$
proof
 from *preal-imp-pos* [*OF A*] *preal-imp-pos* [*OF B*]
 show *add-set A B* $\subseteq \{r.\ 0 < r\}$ **by** (*force simp add*: *add-set-def*)
next
 show *add-set A B* $\neq \{r.\ 0 < r\}$
  by (*insert preal-not-mem-add-set-Ex* [*OF A B*], *blast*)
qed

Part 3 of Dedekind sections definition

lemma *add-set-lemma3*:
    [|$A \in preal$; $B \in preal$; $u \in$ *add-set A B*; $0 < z$; $z < u$|]
     ==> $z \in$ *add-set A B*
proof (*unfold add-set-def*, *clarify*)
 fix *x*::*rat* **and** *y*::*rat*
 assume *A*: $A \in preal$
  and *B*: $B \in preal$
  and [*simp*]: $0 < z$
  and *zless*: $z < x + y$
  and *x*: $x \in A$
  and *y*: $y \in B$
 have *xpos* [*simp*]: $0<x$ **by** (*rule preal-imp-pos* [*OF A x*])
 have *ypos* [*simp*]: $0<y$ **by** (*rule preal-imp-pos* [*OF B y*])
 have *xypos* [*simp*]: $0 < x+y$ **by** (*simp add*: *pos-add-strict*)
 let *?f* $= z/(x+y)$
 have *fless*: *?f* < 1 **by** (*simp add*: *zless pos-divide-less-eq*)
 show $\exists x' \in A.\ \exists y' \in B.\ z = x' + y'$
 proof (*intro bexI*)
  show $z = x*?f + y*?f$
   by (*simp add*: *left-distrib* [*symmetric*] *divide-inverse mult-ac*
     order-less-imp-not-eq2)
 next
  show $y * ?f \in B$
  proof (*rule preal-downwards-closed* [*OF B y*])
   show $0 < y * ?f$
    by (*simp add*: *divide-inverse zero-less-mult-iff*)
  next
   show $y * ?f < y$
    by (*insert mult-strict-left-mono* [*OF fless ypos*], *simp*)
  qed
 next
  show $x * ?f \in A$

```
    proof (rule preal-downwards-closed [OF A x])
      show 0 < x * ?f
        by (simp add: divide-inverse zero-less-mult-iff)
    next
      show x * ?f < x
        by (insert mult-strict-left-mono [OF fless xpos], simp)
    qed
  qed
qed
```

Part 4 of Dedekind sections definition

```
lemma add-set-lemma4:
    [|A ∈ preal; B ∈ preal; y ∈ add-set A B|] ==> ∃u ∈ add-set A B. y < u
apply (auto simp add: add-set-def)
apply (frule preal-exists-greater [of A], auto)
apply (rule-tac x=u + y in exI)
apply (auto intro: add-strict-left-mono)
done

lemma mem-add-set:
    [|A ∈ preal; B ∈ preal|] ==> add-set A B ∈ preal
apply (simp (no-asm-simp) add: preal-def cut-def)
apply (blast intro!: add-set-not-empty add-set-not-rat-set
                     add-set-lemma3 add-set-lemma4)
done

lemma preal-add-assoc: ((x::preal) + y) + z = x + (y + z)
apply (simp add: preal-add-def mem-add-set Rep-preal)
apply (force simp add: add-set-def add-ac)
done

instance preal :: ab-semigroup-add
proof
  fix a b c :: preal
  show (a + b) + c = a + (b + c) by (rule preal-add-assoc)
  show a + b = b + a by (rule preal-add-commute)
qed

lemma preal-add-left-commute: x + (y + z) = y + ((x + z)::preal)
by (rule add-left-commute)
```

Positive Real addition is an AC operator

```
lemmas preal-add-ac = preal-add-assoc preal-add-commute preal-add-left-commute
```

51.4 Properties of Multiplication

Proofs essentially same as for addition

```
lemma preal-mult-commute: (x::preal) * y = y * x
```

apply (*unfold preal-mult-def mult-set-def*)
apply (*rule-tac f = Abs-preal* **in** *arg-cong*)
apply (*force simp add*: *mult-commute*)
done

Multiplication of two positive reals gives a positive real.

Lemmas for proving positive reals multiplication set in *preal*

Part 1 of Dedekind sections definition

lemma *mult-set-not-empty*:
[|*A* ∈ *preal*; *B* ∈ *preal*|] ==> {} ⊂ *mult-set A B*
apply (*insert preal-nonempty* [*of A*] *preal-nonempty* [*of B*])
apply (*auto simp add*: *mult-set-def*)
done

Part 2 of Dedekind sections definition

lemma *preal-not-mem-mult-set-Ex*:
assumes *A*: *A* ∈ *preal*
and *B*: *B* ∈ *preal*
shows ∃ *q*. *0* < *q* & *q* ∉ *mult-set A B*
proof −
from *preal-exists-bound* [*OF A*]
obtain *x* **where** [*simp*]: *0* < *x x* ∉ *A* **by** *blast*
from *preal-exists-bound* [*OF B*]
obtain *y* **where** [*simp*]: *0* < *y y* ∉ *B* **by** *blast*
show *?thesis*
proof (*intro exI conjI*)
show *0* < *x*∗*y* **by** (*simp add*: *mult-pos-pos*)
show *x* ∗ *y* ∉ *mult-set A B*
proof −
{ fix *u*::*rat* **and** *v*::*rat*
assume *u* ∈ *A* **and** *v* ∈ *B* **and** *x*∗*y* = *u*∗*v*
moreover
with *prems* **have** *u*<*x* **and** *v*<*y* **by** (*blast dest*: *not-in-preal-ub*)+
moreover
with *prems* **have** *0*≤*v*
by (*blast intro*: *preal-imp-pos* [*OF B*] *order-less-imp-le prems*)
moreover
from *calculation*
have *u*∗*v* < *x*∗*y* **by** (*blast intro*: *mult-strict-mono prems*)
ultimately have *False* **by** *force* **}**
thus *?thesis* **by** (*auto simp add*: *mult-set-def*)
qed
qed
qed

lemma *mult-set-not-rat-set*:
assumes *A*: *A* ∈ *preal*

```
    and B: B ∈ preal
  shows mult-set A B < {r. 0 < r}
proof
  show mult-set A B ⊆ {r. 0 < r}
    by (force simp add: mult-set-def
      intro: preal-imp-pos [OF A] preal-imp-pos [OF B] mult-pos-pos)
  show mult-set A B ≠ {r. 0 < r}
    using preal-not-mem-mult-set-Ex [OF A B] by blast
qed
```

Part 3 of Dedekind sections definition

```
lemma mult-set-lemma3:
     [|A ∈ preal; B ∈ preal; u ∈ mult-set A B; 0 < z; z < u|]
      ==> z ∈ mult-set A B
proof (unfold mult-set-def, clarify)
  fix x::rat and y::rat
  assume A: A ∈ preal
    and B: B ∈ preal
    and [simp]: 0 < z
    and zless: z < x * y
    and x:  x ∈ A
    and y:  y ∈ B
  have [simp]: 0<y by (rule preal-imp-pos [OF B y])
  show ∃ x' ∈ A. ∃ y' ∈ B. z = x' * y'
  proof
    show ∃ y'∈B. z = (z/y) * y'
    proof
      show z = (z/y)*y
        by (simp add: divide-inverse mult-commute [of y] mult-assoc
                  order-less-imp-not-eq2)
      show y ∈ B by fact
    qed
  next
    show z/y ∈ A
    proof (rule preal-downwards-closed [OF A x])
      show 0 < z/y
        by (simp add: zero-less-divide-iff)
      show z/y < x by (simp add: pos-divide-less-eq zless)
    qed
  qed
qed
```

Part 4 of Dedekind sections definition

```
lemma mult-set-lemma4:
     [|A ∈ preal; B ∈ preal; y ∈ mult-set A B|] ==> ∃ u ∈ mult-set A B. y < u
apply (auto simp add: mult-set-def)
apply (frule preal-exists-greater [of A], auto)
apply (rule-tac x=u * y in exI)
apply (auto intro: preal-imp-pos [of A] preal-imp-pos [of B]
```

mult-strict-right-mono)
done

lemma *mem-mult-set*:
[|$A \in$ *preal*; $B \in$ *preal*|] ==> *mult-set A B* $\in$ *preal*
apply (*simp* (*no-asm-simp*) *add*: *preal-def cut-def*)
apply (*blast intro*!: *mult-set-not-empty mult-set-not-rat-set*
mult-set-lemma3 mult-set-lemma4)
done

lemma *preal-mult-assoc*: $((x::preal) * y) * z = x * (y * z)$
apply (*simp add*: *preal-mult-def mem-mult-set Rep-preal*)
apply (*force simp add*: *mult-set-def mult-ac*)
done

instance *preal* :: *ab-semigroup-mult*
proof
fix *a b c* :: *preal*
show $(a * b) * c = a * (b * c)$ **by** (*rule preal-mult-assoc*)
show $a * b = b * a$ **by** (*rule preal-mult-commute*)
qed

lemma *preal-mult-left-commute*: $x * (y * z) = y * ((x * z)::preal)$
by (*rule mult-left-commute*)

Positive Real multiplication is an AC operator

lemmas *preal-mult-ac* =
preal-mult-assoc preal-mult-commute preal-mult-left-commute

Positive real 1 is the multiplicative identity element

lemma *preal-mult-1*: $(1::preal) * z = z$
unfolding *preal-one-def*
proof (*induct z*)
fix *A* :: *rat set*
assume *A*: $A \in$ *preal*
have $\{w.\ \exists u.\ 0 < u \wedge u < 1\ \&\ (\exists v \in A.\ w = u * v)\} = A$ (**is** *?lhs* = *A*)
proof
show *?lhs* $\subseteq$ *A*
proof *clarify*
fix *x*::*rat* **and** *u*::*rat* **and** *v*::*rat*
assume *upos*: *0*<*u* **and** *u*<*1* **and** *v*: $v \in A$
have *vpos*: *0*<*v* **by** (*rule preal-imp-pos* [*OF A v*])
hence *u**v < *1**v **by** (*simp only*: *mult-strict-right-mono prems*)
thus $u * v \in A$
by (*force intro*: *preal-downwards-closed* [*OF A v*] *mult-pos-pos*
upos vpos)
qed
next

```
    show A ⊆ ?lhs
    proof clarify
      fix x::rat
      assume x: x ∈ A
      have xpos: 0<x by (rule preal-imp-pos [OF A x])
      from preal-exists-greater [OF A x]
      obtain v where v: v ∈ A and xlessv: x < v ..
      have vpos: 0<v by (rule preal-imp-pos [OF A v])
      show ∃ u. 0 < u ∧ u < 1 ∧ (∃ v∈A. x = u * v)
      proof (intro exI conjI)
        show 0 < x/v
          by (simp add: zero-less-divide-iff xpos vpos)
        show x / v < 1
          by (simp add: pos-divide-less-eq vpos xlessv)
        show ∃ v'∈A. x = (x / v) * v'
        proof
          show x = (x/v)*v
            by (simp add: divide-inverse mult-assoc vpos
                          order-less-imp-not-eq2)
          show v ∈ A by fact
        qed
      qed
    qed
  qed
  thus preal-of-rat 1 * Abs-preal A = Abs-preal A
    by (simp add: preal-of-rat-def preal-mult-def mult-set-def
                  rat-mem-preal A)
qed

instance preal :: comm-monoid-mult
by intro-classes (rule preal-mult-1)

lemma preal-mult-1-right: z * (1::preal) = z
by (rule mult-1-right)
```

51.5 Distribution of Multiplication across Addition

```
lemma mem-Rep-preal-add-iff:
      (z ∈ Rep-preal(R+S)) = (∃ x ∈ Rep-preal R. ∃ y ∈ Rep-preal S. z = x + y)
apply (simp add: preal-add-def mem-add-set Rep-preal)
apply (simp add: add-set-def)
done

lemma mem-Rep-preal-mult-iff:
      (z ∈ Rep-preal(R*S)) = (∃ x ∈ Rep-preal R. ∃ y ∈ Rep-preal S. z = x * y)
apply (simp add: preal-mult-def mem-mult-set Rep-preal)
apply (simp add: mult-set-def)
done
```

lemma *distrib-subset1*:
  Rep-preal $(w * (x + y)) \subseteq$ *Rep-preal* $(w * x + w * y)$
apply (*auto simp add*: *Bex-def mem-Rep-preal-add-iff mem-Rep-preal-mult-iff*)
apply (*force simp add*: *right-distrib*)
done

lemma *preal-add-mult-distrib-mean*:
 assumes *a*: $a \in$ *Rep-preal* w
  and *b*: $b \in$ *Rep-preal* w
  and *d*: $d \in$ *Rep-preal* x
  and *e*: $e \in$ *Rep-preal* y
 shows $\exists c \in$ *Rep-preal* $w.\ a * d + b * e = c * (d + e)$
proof
 let *?c* $= (a*d + b*e)/(d+e)$
 have [*simp*]: $0<a\ 0<b\ 0<d\ 0<e\ 0<d+e$
  by (*blast intro*: *preal-imp-pos* [*OF Rep-preal*] *a b d e pos-add-strict*)+
 have *cpos*: $0 <$ *?c*
  by (*simp add*: *zero-less-divide-iff zero-less-mult-iff pos-add-strict*)
 show $a * d + b * e =$ *?c* $* (d + e)$
  by (*simp add*: *divide-inverse mult-assoc order-less-imp-not-eq2*)
 show *?c* $\in$ *Rep-preal* w
 proof (*cases rule*: *linorder-le-cases*)
  assume $a \leq b$
  hence *?c* $\leq b$
   by (*simp add*: *pos-divide-le-eq right-distrib mult-right-mono*
       order-less-imp-le)
  thus *?thesis* **by** (*rule preal-downwards-closed'* [*OF Rep-preal b cpos*])
 next
  assume $b \leq a$
  hence *?c* $\leq a$
   by (*simp add*: *pos-divide-le-eq right-distrib mult-right-mono*
       order-less-imp-le)
  thus *?thesis* **by** (*rule preal-downwards-closed'* [*OF Rep-preal a cpos*])
 qed
qed

lemma *distrib-subset2*:
  Rep-preal $(w * x + w * y) \subseteq$ *Rep-preal* $(w * (x + y))$
apply (*auto simp add*: *Bex-def mem-Rep-preal-add-iff mem-Rep-preal-mult-iff*)
apply (*drule-tac* $w{=}w$ **and** $x{=}x$ **and** $y{=}y$ **in** *preal-add-mult-distrib-mean*, *auto*)
done

lemma *preal-add-mult-distrib2*: $(w * ((x::preal) + y)) = (w * x) + (w * y)$
apply (*rule Rep-preal-inject* [*THEN iffD1*])
apply (*rule equalityI* [*OF distrib-subset1 distrib-subset2*])
done

lemma *preal-add-mult-distrib*: $(((x::preal) + y) * w) = (x * w) + (y * w)$
by (*simp add*: *preal-mult-commute preal-add-mult-distrib2*)

instance *preal* :: *comm-semiring*
by *intro-classes* (*rule preal-add-mult-distrib*)

51.6 Existence of Inverse, a Positive Real

lemma *mem-inv-set-ex*:
assumes *A*: $A \in preal$ **shows** $\exists x\ y.\ 0 < x\ \&\ x < y\ \&\ inverse\ y \notin A$
proof −
from *preal-exists-bound* [*OF A*]
obtain *x* **where** [*simp*]: $0<x\ x \notin A$ **by** *blast*
show *?thesis*
proof (*intro exI conjI*)
show $0 < inverse\ (x+1)$
by (*simp add*: *order-less-trans* [*OF - less-add-one*])
show $inverse(x+1) < inverse\ x$
by (*simp add*: *less-imp-inverse-less less-add-one*)
show $inverse\ (inverse\ x) \notin A$
by (*simp add*: *order-less-imp-not-eq2*)
qed
qed

Part 1 of Dedekind sections definition

lemma *inverse-set-not-empty*:
$A \in preal \Longrightarrow \{\} \subset inverse\text{-}set\ A$
apply (*insert mem-inv-set-ex* [*of A*])
apply (*auto simp add*: *inverse-set-def*)
done

Part 2 of Dedekind sections definition

lemma *preal-not-mem-inverse-set-Ex*:
assumes *A*: $A \in preal$ **shows** $\exists q.\ 0 < q\ \&\ q \notin inverse\text{-}set\ A$
proof −
from *preal-nonempty* [*OF A*]
obtain *x* **where** *x*: $x \in A$ **and** *xpos* [*simp*]: $0<x$ **..**
show *?thesis*
proof (*intro exI conjI*)
show $0 < inverse\ x$ **by** *simp*
show $inverse\ x \notin inverse\text{-}set\ A$
proof −
{ fix *y*::*rat*
assume *ygt*: $inverse\ x < y$
have [*simp*]: $0 < y$ **by** (*simp add*: *order-less-trans* [*OF - ygt*])
have *iyless*: $inverse\ y < x$
by (*simp add*: *inverse-less-imp-less* [*of x*] *ygt*)
have $inverse\ y \in A$
by (*simp add*: *preal-downwards-closed* [*OF A x*] *iyless*)**}**
thus *?thesis* **by** (*auto simp add*: *inverse-set-def*)
qed

qed
qed

lemma *inverse-set-not-rat-set*:
assumes *A*: $A \in preal$ **shows** $inverse\text{-}set\ A < \{r.\ 0 < r\}$
proof
show $inverse\text{-}set\ A \subseteq \{r.\ 0 < r\}$ **by** (*force simp add*: *inverse-set-def*)
next
show $inverse\text{-}set\ A \neq \{r.\ 0 < r\}$
by (*insert preal-not-mem-inverse-set-Ex* [*OF A*], *blast*)
qed

Part 3 of Dedekind sections definition

lemma *inverse-set-lemma3*:
$[|A \in preal;\ u \in inverse\text{-}set\ A;\ 0 < z;\ z < u|] ==> z \in inverse\text{-}set\ A$
apply (*auto simp add*: *inverse-set-def*)
apply (*auto intro*: *order-less-trans*)
done

Part 4 of Dedekind sections definition

lemma *inverse-set-lemma4*:
$[|A \in preal;\ y \in inverse\text{-}set\ A|] ==> \exists u \in inverse\text{-}set\ A.\ y < u$
apply (*auto simp add*: *inverse-set-def*)
apply (*drule dense* [*of y*])
apply (*blast intro*: *order-less-trans*)
done

lemma *mem-inverse-set*:
$A \in preal ==> inverse\text{-}set\ A \in preal$
apply (*simp* (*no-asm-simp*) *add*: *preal-def cut-def*)
apply (*blast intro*!: *inverse-set-not-empty inverse-set-not-rat-set*
inverse-set-lemma3 inverse-set-lemma4)
done

51.7 Gleason's Lemma 9-3.4, page 122

lemma *Gleason9-34-exists*:
assumes *A*: $A \in preal$
and $\forall x \in A.\ x + u \in A$
and $0 \leq z$
shows $\exists b \in A.\ b + (of\text{-}int\ z) * u \in A$
proof (*cases z rule*: *int-cases*)
case (*nonneg n*)
show *?thesis*
proof (*simp add*: *prems*, *induct n*)
case *0*
from *preal-nonempty* [*OF A*]

```
    show ?case by force
  case (Suc k)
    from this obtain b where b ∈ A b + of-nat k * u ∈ A ..
    hence b + of-int (int k)*u + u ∈ A by (simp add: prems)
    thus ?case by (force simp add: algebra-simps prems)
 qed
next
 case (neg n)
 with prems show ?thesis by simp
qed

lemma Gleason9-34-contra:
 assumes A: A ∈ preal
  shows [|∀x∈A. x + u ∈ A; 0 < u; 0 < y; y ∉ A|] ==> False
proof (induct u, induct y)
 fix a::int and b::int
 fix c::int and d::int
 assume bpos [simp]: 0 < b
   and dpos [simp]: 0 < d
   and closed: ∀x∈A. x + (Fract c d) ∈ A
   and upos: 0 < Fract c d
   and ypos: 0 < Fract a b
   and notin: Fract a b ∉ A
 have cpos [simp]: 0 < c
   by (simp add: zero-less-Fract-iff [OF dpos, symmetric] upos)
 have apos [simp]: 0 < a
   by (simp add: zero-less-Fract-iff [OF bpos, symmetric] ypos)
 let ?k = a*d
 have frle: Fract a b ≤ Fract ?k 1 * (Fract c d)
 proof -
   have ?thesis = ((a * d * b * d) ≤ c * b * (a * d * b * d))
     by (simp add: mult-rat le-rat order-less-imp-not-eq2 mult-ac)
   moreover
   have (1 * (a * d * b * d)) ≤ c * b * (a * d * b * d)
     by (rule mult-mono,
         simp-all add: int-one-le-iff-zero-less zero-less-mult-iff
                       order-less-imp-le)
   ultimately
   show ?thesis by simp
 qed
 have k: 0 ≤ ?k by (simp add: order-less-imp-le zero-less-mult-iff)
 from Gleason9-34-exists [OF A closed k]
 obtain z where z: z ∈ A
           and mem: z + of-int ?k * Fract c d ∈ A ..
 have less: z + of-int ?k * Fract c d < Fract a b
   by (rule not-in-preal-ub [OF A notin mem ypos])
 have 0<z by (rule preal-imp-pos [OF A z])
 with frle and less show False by (simp add: Fract-of-int-eq)
qed
```

lemma *Gleason9-34*:
assumes *A*: $A \in preal$
and *upos*: $0 < u$
shows $\exists r \in A.\ r + u \notin A$
proof (*rule ccontr*, *simp*)
assume *closed*: $\forall r \in A.\ r + u \in A$
from *preal-exists-bound* [*OF A*]
obtain *y* **where** *y*: $y \notin A$ **and** *ypos*: $0 < y$ **by** *blast*
show *False*
by (*rule Gleason9-34-contra* [*OF A closed upos ypos y*])
qed

51.8 Gleason's Lemma 9-3.6

lemma *lemma-gleason9-36*:
assumes *A*: $A \in preal$
and *x*: $1 < x$
shows $\exists r \in A.\ r*x \notin A$
proof −
from *preal-nonempty* [*OF A*]
obtain *y* **where** *y*: $y \in A$ **and** *ypos*: $0<y$ **..**
show *?thesis*
proof (*rule classical*)
assume $\sim(\exists r \in A.\ r * x \notin A)$
with *y* **have** *ymem*: $y * x \in A$ **by** *blast*
from *ypos mult-strict-left-mono* [*OF x*]
have *yless*: $y < y*x$ **by** *simp*
let *?d* $= y*x - y$
from *yless* **have** *dpos*: $0 <$ *?d* **and** *eq*: $y +$ *?d* $= y*x$ **by** *auto*
from *Gleason9-34* [*OF A dpos*]
obtain *r* **where** *r*: $r \in A$ **and** *notin*: $r +$ *?d* $\notin A$ **..**
have *rpos*: $0<r$ **by** (*rule preal-imp-pos* [*OF A r*])
with *dpos* **have** *rdpos*: $0 < r +$ *?d* **by** *arith*
have $\sim$ $(r +$ *?d* $\leq y +$ *?d*$)$
proof
assume *le*: $r +$ *?d* $\leq y +$ *?d*
from *ymem* **have** *yd*: $y +$ *?d* $\in A$ **by** (*simp add*: *eq*)
have $r +$ *?d* $\in A$ **by** (*rule preal-downwards-closed'* [*OF A yd rdpos le*])
with *notin* **show** *False* **by** *simp*
qed
hence $y < r$ **by** *simp*
with *ypos* **have** *dless*: *?d* $< (r * ?d)/y$
by (*simp add*: *pos-less-divide-eq mult-commute* [*of ?d*]
mult-strict-right-mono dpos)
have $r +$ *?d* $< r*x$
proof −
have $r +$ *?d* $< r + (r * ?d)/y$ **by** (*simp add*: *dless*)

```
    also with ypos have ... = (r/y) * (y + ?d)
      by (simp only: algebra-simps divide-inverse, simp)
    also have ... = r*x using ypos
      by (simp add: times-divide-eq-left)
    finally show r + ?d < r*x .
  qed
  with r notin rdpos
  show ∃r∈A. r * x ∉ A by (blast dest:  preal-downwards-closed [OF A])
 qed
qed
```

51.9 Existence of Inverse: Part 2

```
lemma mem-Rep-preal-inverse-iff:
     (z ∈ Rep-preal(inverse R)) =
      (0 < z ∧ (∃ y. z < y ∧ inverse y ∉ Rep-preal R))
apply (simp add: preal-inverse-def mem-inverse-set Rep-preal)
apply (simp add: inverse-set-def)
done

lemma Rep-preal-of-rat:
    0 < q ==> Rep-preal (preal-of-rat q) = {x. 0 < x ∧ x < q}
by (simp add: preal-of-rat-def rat-mem-preal)

lemma subset-inverse-mult-lemma:
 assumes xpos: 0 < x and xless: x < 1
 shows ∃r u y. 0 < r & r < y & inverse y ∉ Rep-preal R &
   u ∈ Rep-preal R & x = r * u
proof −
 from xpos and xless have 1 < inverse x by (simp add: one-less-inverse-iff)
 from lemma-gleason9-36 [OF Rep-preal this]
 obtain r where r: r ∈ Rep-preal R
          and notin: r * (inverse x) ∉ Rep-preal R ..
 have rpos: 0<r by (rule preal-imp-pos [OF Rep-preal r])
 from preal-exists-greater [OF Rep-preal r]
 obtain u where u: u ∈ Rep-preal R and rless: r < u ..
 have upos: 0<u by (rule preal-imp-pos [OF Rep-preal u])
 show ?thesis
 proof (intro exI conjI)
   show 0 < x/u using xpos upos
     by (simp add: zero-less-divide-iff)
   show x/u < x/r using xpos upos rpos
     by (simp add: divide-inverse mult-less-cancel-left rless)
   show inverse (x / r) ∉ Rep-preal R using notin
     by (simp add: divide-inverse mult-commute)
   show u ∈ Rep-preal R by (rule u)
   show x = x / u * u using upos
     by (simp add: divide-inverse mult-commute)
 qed
```

qed

lemma *subset-inverse-mult*:
Rep-preal(*preal-of-rat 1*) ⊆ *Rep-preal*(*inverse R* ∗ *R*)
apply (*auto simp add*: *Bex-def Rep-preal-of-rat mem-Rep-preal-inverse-iff mem-Rep-preal-mult-iff*)
apply (*blast dest*: *subset-inverse-mult-lemma*)
done

lemma *inverse-mult-subset-lemma*:
assumes *rpos*: *0* < *r*
and *rless*: *r* < *y*
and *notin*: *inverse y* ∉ *Rep-preal R*
and *q*: *q* ∈ *Rep-preal R*
shows *r*∗*q* < *1*
proof −
have *q* < *inverse y* **using** *rpos rless*
by (*simp add*: *not-in-preal-ub* [*OF Rep-preal notin*] *q*)
hence *r* ∗ *q* < *r*/*y* **using** *rpos*
by (*simp add*: *divide-inverse mult-less-cancel-left*)
also have ... ≤ *1* **using** *rpos rless*
by (*simp add*: *pos-divide-le-eq*)
finally show *?thesis* **.**
qed

lemma *inverse-mult-subset*:
Rep-preal(*inverse R* ∗ *R*) ⊆ *Rep-preal*(*preal-of-rat 1*)
apply (*auto simp add*: *Bex-def Rep-preal-of-rat mem-Rep-preal-inverse-iff mem-Rep-preal-mult-iff*)
apply (*simp add*: *zero-less-mult-iff preal-imp-pos* [*OF Rep-preal*])
apply (*blast intro*: *inverse-mult-subset-lemma*)
done

lemma *preal-mult-inverse*: *inverse R* ∗ *R* = (*1*::*preal*)
unfolding *preal-one-def*
apply (*rule Rep-preal-inject* [*THEN iffD1*])
apply (*rule equalityI* [*OF inverse-mult-subset subset-inverse-mult*])
done

lemma *preal-mult-inverse-right*: *R* ∗ *inverse R* = (*1*::*preal*)
apply (*rule preal-mult-commute* [*THEN subst*])
apply (*rule preal-mult-inverse*)
done

Theorems needing *Gleason9-34*

lemma *Rep-preal-self-subset*: *Rep-preal* (*R*) ⊆ *Rep-preal*(*R* + *S*)
proof
fix *r*
assume *r*: *r* ∈ *Rep-preal R*

have *rpos*: *0<r* **by** (*rule preal-imp-pos* [*OF Rep-preal r*])
from *mem-Rep-preal-Ex*
obtain *y* **where** *y*: *y* ∈ *Rep-preal S* **..**
have *ypos*: *0<y* **by** (*rule preal-imp-pos* [*OF Rep-preal y*])
have *ry*: *r+y* ∈ *Rep-preal*(*R* + *S*) **using** *r y*
by (*auto simp add*: *mem-Rep-preal-add-iff*)
show *r* ∈ *Rep-preal*(*R* + *S*) **using** *r ypos rpos*
by (*simp add*: *preal-downwards-closed* [*OF Rep-preal ry*])
qed

lemma *Rep-preal-sum-not-subset*: ~ *Rep-preal* (*R* + *S*) ⊆ *Rep-preal*(*R*)
proof −
from *mem-Rep-preal-Ex*
obtain *y* **where** *y*: *y* ∈ *Rep-preal S* **..**
have *ypos*: *0<y* **by** (*rule preal-imp-pos* [*OF Rep-preal y*])
from *Gleason9-34* [*OF Rep-preal ypos*]
obtain *r* **where** *r*: *r* ∈ *Rep-preal R* **and** *notin*: *r* + *y* ∉ *Rep-preal R* **..**
have *r* + *y* ∈ *Rep-preal* (*R* + *S*) **using** *r y*
by (*auto simp add*: *mem-Rep-preal-add-iff*)
thus *?thesis* **using** *notin* **by** *blast*
qed

lemma *Rep-preal-sum-not-eq*: *Rep-preal* (*R* + *S*) ≠ *Rep-preal*(*R*)
by (*insert Rep-preal-sum-not-subset*, *blast*)

at last, Gleason prop. 9-3.5(iii) page 123

lemma *preal-self-less-add-left*: (*R*::*preal*) < *R* + *S*
apply (*unfold preal-less-def less-le*)
apply (*simp add*: *Rep-preal-self-subset Rep-preal-sum-not-eq* [*THEN not-sym*])
done

lemma *preal-self-less-add-right*: (*R*::*preal*) < *S* + *R*
by (*simp add*: *preal-add-commute preal-self-less-add-left*)

lemma *preal-not-eq-self*: *x* ≠ *x* + (*y*::*preal*)
by (*insert preal-self-less-add-left* [*of x y*], *auto*)

51.10 Subtraction for Positive Reals

Gleason prop. 9-3.5(iv), page 123: proving $A < B \Longrightarrow \exists D.\ A + D = B$. We define the claimed D and show that it is a positive real

Part 1 of Dedekind sections definition

lemma *diff-set-not-empty*:
R < *S* ==> {} ⊂ *diff-set* (*Rep-preal S*) (*Rep-preal R*)
apply (*auto simp add*: *preal-less-def diff-set-def elim*!: *equalityE*)
apply (*frule-tac x1* = *S* **in** *Rep-preal* [*THEN preal-exists-greater*])
apply (*drule preal-imp-pos* [*OF Rep-preal*], *clarify*)
apply (*cut-tac a*=*x* **and** *b*=*u* **in** *add-eq-exists*, *force*)

done

Part 2 of Dedekind sections definition

lemma *diff-set-nonempty*:
  $\exists q.\ 0 < q\ \&\ q \notin$ *diff-set* (*Rep-preal S*) (*Rep-preal R*)
apply (*cut-tac* $X = S$ **in** *Rep-preal-exists-bound*)
apply (*erule exE*)
apply (*rule-tac* $x = x$ **in** *exI*, *auto*)
apply (*simp add*: *diff-set-def*)
apply (*auto dest*: *Rep-preal* [*THEN preal-downwards-closed*])
done

lemma *diff-set-not-rat-set*:
  diff-set (*Rep-preal S*) (*Rep-preal R*) $<$ $\{r.\ 0 < r\}$ (**is** *?lhs* $<$ *?rhs*)
proof
  show *?lhs* $\subseteq$ *?rhs* **by** (*auto simp add*: *diff-set-def*)
  show *?lhs* $\neq$ *?rhs* **using** *diff-set-nonempty* **by** *blast*
qed

Part 3 of Dedekind sections definition

lemma *diff-set-lemma3*:
  [|$R < S$; $u \in$ *diff-set* (*Rep-preal S*) (*Rep-preal R*); $0 < z$; $z < u$|]
  ==> $z \in$ *diff-set* (*Rep-preal S*) (*Rep-preal R*)
apply (*auto simp add*: *diff-set-def*)
apply (*rule-tac x*=*x* **in** *exI*)
apply (*drule Rep-preal* [*THEN preal-downwards-closed*], *auto*)
done

Part 4 of Dedekind sections definition

lemma *diff-set-lemma4*:
  [|$R < S$; $y \in$ *diff-set* (*Rep-preal S*) (*Rep-preal R*)|]
  ==> $\exists u \in$ *diff-set* (*Rep-preal S*) (*Rep-preal R*). $y < u$
apply (*auto simp add*: *diff-set-def*)
apply (*drule Rep-preal* [*THEN preal-exists-greater*], *clarify*)
apply (*cut-tac a*=*x*+*y* **and** *b*=*u* **in** *add-eq-exists*, *clarify*)
apply (*rule-tac x*=*y*+*xa* **in** *exI*)
apply (*auto simp add*: *add-ac*)
done

lemma *mem-diff-set*:
  $R < S$ ==> *diff-set* (*Rep-preal S*) (*Rep-preal R*) $\in$ *preal*
apply (*unfold preal-def cut-def*)
apply (*blast intro*!: *diff-set-not-empty diff-set-not-rat-set*
                diff-set-lemma3 diff-set-lemma4)
done

lemma *mem-Rep-preal-diff-iff*:
  $R < S$ ==>
  ($z \in$ *Rep-preal*($S-R$)) =

(∃ x. 0 < x & 0 < z & x ∉ Rep-preal R & x + z ∈ Rep-preal S)
apply (*simp add*: *preal-diff-def mem-diff-set Rep-preal*)
apply (*force simp add*: *diff-set-def*)
done

proving that $R + D \leq S$

lemma *less-add-left-lemma*:
assumes *Rless*: $R < S$
and *a*: $a \in$ *Rep-preal R*
and *cb*: $c + b \in$ *Rep-preal S*
and $c \notin$ *Rep-preal R*
and $0 < b$
and $0 < c$
shows $a + b \in$ *Rep-preal S*
proof −
have *0<a* **by** (*rule preal-imp-pos* [*OF Rep-preal a*])
moreover
have $a < c$ **using** *prems*
by (*blast intro*: *not-in-Rep-preal-ub*)
ultimately show *?thesis* **using** *prems*
by (*simp add*: *preal-downwards-closed* [*OF Rep-preal cb*])
qed

lemma *less-add-left-le1*:
$R < (S::preal) ==> R + (S-R) \leq S$
apply (*auto simp add*: *Bex-def preal-le-def mem-Rep-preal-add-iff*
mem-Rep-preal-diff-iff)
apply (*blast intro*: *less-add-left-lemma*)
done

51.11 proving that $S \leq R + D$ — trickier

lemma *lemma-sum-mem-Rep-preal-ex*:
$x \in$ *Rep-preal S* ==> ∃ e. $0 < e$ & $x + e \in$ *Rep-preal S*
apply (*drule Rep-preal* [*THEN preal-exists-greater*], *clarify*)
apply (*cut-tac a=x* **and** *b=u* **in** *add-eq-exists*, *auto*)
done

lemma *less-add-left-lemma2*:
assumes *Rless*: $R < S$
and *x*: $x \in$ *Rep-preal S*
and *xnot*: $x \notin$ *Rep-preal R*
shows ∃ u v z. $0 < v$ & $0 < z$ & $u \in$ *Rep-preal R* & $z \notin$ *Rep-preal R* &
$z + v \in$ *Rep-preal S* & $x = u + v$
proof −
have *xpos*: *0<x* **by** (*rule preal-imp-pos* [*OF Rep-preal x*])
from *lemma-sum-mem-Rep-preal-ex* [*OF x*]
obtain *e* **where** *epos*: $0 < e$ **and** *xe*: $x + e \in$ *Rep-preal S* **by** *blast*
from *Gleason9-34* [*OF Rep-preal epos*]

obtain *r* **where** *r*: $r \in$ *Rep-preal R* **and** *notin*: $r + e \notin$ *Rep-preal R* **..**
with *x xnot xpos* **have** *rless*: $r < x$ **by** (*blast intro*: *not-in-Rep-preal-ub*)
from *add-eq-exists* [*of r x*]
obtain *y* **where** *eq*: $x = r+y$ **by** *auto*
show *?thesis*
proof (*intro exI conjI*)
show $r \in$ *Rep-preal R* **by** (*rule r*)
show $r + e \notin$ *Rep-preal R* **by** (*rule notin*)
show $r + e + y \in$ *Rep-preal S* **using** *xe eq* **by** (*simp add*: *add-ac*)
show $x = r + y$ **by** (*simp add*: *eq*)
show $0 < r + e$ **using** *epos preal-imp-pos* [*OF Rep-preal r*]
by *simp*
show $0 < y$ **using** *rless eq* **by** *arith*
qed
qed

lemma *less-add-left-le2*: $R < (S::preal) \Longrightarrow S \leq R + (S-R)$
apply (*auto simp add*: *preal-le-def*)
apply (*case-tac* $x \in$ *Rep-preal R*)
apply (*cut-tac Rep-preal-self-subset* [*of R*], *force*)
apply (*auto simp add*: *Bex-def mem-Rep-preal-add-iff mem-Rep-preal-diff-iff*)
apply (*blast dest*: *less-add-left-lemma2*)
done

lemma *less-add-left*: $R < (S::preal) \Longrightarrow R + (S-R) = S$
by (*blast intro*: *antisym* [*OF less-add-left-le1 less-add-left-le2*])

lemma *less-add-left-Ex*: $R < (S::preal) \Longrightarrow \exists D.\ R + D = S$
by (*fast dest*: *less-add-left*)

lemma *preal-add-less2-mono1*: $R < (S::preal) \Longrightarrow R + T < S + T$
apply (*auto dest*!: *less-add-left-Ex simp add*: *preal-add-assoc*)
apply (*rule-tac y1* $= D$ **in** *preal-add-commute* [*THEN subst*])
apply (*auto intro*: *preal-self-less-add-left simp add*: *preal-add-assoc* [*symmetric*])
done

lemma *preal-add-less2-mono2*: $R < (S::preal) \Longrightarrow T + R < T + S$
by (*auto intro*: *preal-add-less2-mono1 simp add*: *preal-add-commute* [*of T*])

lemma *preal-add-right-less-cancel*: $R + T < S + T \Longrightarrow R < (S::preal)$
apply (*insert linorder-less-linear* [*of R S*], *auto*)
apply (*drule-tac* $R = S$ **and** $T = T$ **in** *preal-add-less2-mono1*)
apply (*blast dest*: *order-less-trans*)
done

lemma *preal-add-left-less-cancel*: $T + R < T + S \Longrightarrow R < (S::preal)$
by (*auto elim*: *preal-add-right-less-cancel simp add*: *preal-add-commute* [*of T*])

lemma *preal-add-less-cancel-right*: $((R::preal) + T < S + T) = (R < S)$

by (*blast intro*: *preal-add-less2-mono1 preal-add-right-less-cancel*)

lemma *preal-add-less-cancel-left*: $(T + (R::preal) < T + S) = (R < S)$
by (*blast intro*: *preal-add-less2-mono2 preal-add-left-less-cancel*)

lemma *preal-add-le-cancel-right*: $((R::preal) + T \leq S + T) = (R \leq S)$
by (*simp add*: *linorder-not-less* [*symmetric*] *preal-add-less-cancel-right*)

lemma *preal-add-le-cancel-left*: $(T + (R::preal) \leq T + S) = (R \leq S)$
by (*simp add*: *linorder-not-less* [*symmetric*] *preal-add-less-cancel-left*)

lemma *preal-add-less-mono*:
$[\!|\ x1 < y1;\ x2 < y2\ |\!] ==> x1 + x2 < y1 + (y2::preal)$
apply (*auto dest*!: *less-add-left-Ex simp add*: *preal-add-ac*)
apply (*rule preal-add-assoc* [*THEN subst*])
apply (*rule preal-self-less-add-right*)
done

lemma *preal-add-right-cancel*: $(R::preal) + T = S + T ==> R = S$
apply (*insert linorder-less-linear* [*of R S*], *safe*)
apply (*drule-tac* [!] $T = T$ **in** *preal-add-less2-mono1*, *auto*)
done

lemma *preal-add-left-cancel*: $C + A = C + B ==> A = (B::preal)$
by (*auto intro*: *preal-add-right-cancel simp add*: *preal-add-commute*)

lemma *preal-add-left-cancel-iff*: $(C + A = C + B) = ((A::preal) = B)$
by (*fast intro*: *preal-add-left-cancel*)

lemma *preal-add-right-cancel-iff*: $(A + C = B + C) = ((A::preal) = B)$
by (*fast intro*: *preal-add-right-cancel*)

lemmas *preal-cancels* =
preal-add-less-cancel-right preal-add-less-cancel-left
preal-add-le-cancel-right preal-add-le-cancel-left
preal-add-left-cancel-iff preal-add-right-cancel-iff

instance *preal* :: *ordered-cancel-ab-semigroup-add*
proof
fix *a b c* :: *preal*
show $a + b = a + c \Longrightarrow b = c$ **by** (*rule preal-add-left-cancel*)
show $a \leq b \Longrightarrow c + a \leq c + b$ **by** (*simp only*: *preal-add-le-cancel-left*)
qed

51.12 Completeness of type *preal*

Prove that supremum is a cut

Part 1 of Dedekind sections definition

lemma *preal-sup-set-not-empty*:
$P \neq \{\} \Longrightarrow \{\} \subset (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))$
apply *auto*
apply (*cut-tac* $X = x$ **in** *mem-Rep-preal-Ex*, *auto*)
done

Part 2 of Dedekind sections definition

lemma *preal-sup-not-exists*:
$\forall X \in P.\ X \leq Y \Longrightarrow \exists q.\ 0 < q\ \&\ q \notin (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))$
apply (*cut-tac* $X = Y$ **in** *Rep-preal-exists-bound*)
apply (*auto simp add*: *preal-le-def*)
done

lemma *preal-sup-set-not-rat-set*:
$\forall X \in P.\ X \leq Y \Longrightarrow (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X)) < \{r.\ 0 < r\}$
apply (*drule preal-sup-not-exists*)
apply (*blast intro*: *preal-imp-pos* [*OF Rep-preal*])
done

Part 3 of Dedekind sections definition

lemma *preal-sup-set-lemma3*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y; u \in (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X)); 0 < z; z < u|]$
$\Longrightarrow z \in (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))$
by (*auto elim*: *Rep-preal* [*THEN preal-downwards-closed*])

Part 4 of Dedekind sections definition

lemma *preal-sup-set-lemma4*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y; y \in (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X))\ |]$
$\Longrightarrow \exists u \in (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X)).\ y < u$
by (*blast dest*: *Rep-preal* [*THEN preal-exists-greater*])

lemma *preal-sup*:
$[|P \neq \{\}; \forall X \in P.\ X \leq Y|] \Longrightarrow (\bigcup X \in P.\ \mathit{Rep\text{-}preal}(X)) \in \mathit{preal}$
apply (*unfold preal-def cut-def*)
apply (*blast intro*!: *preal-sup-set-not-empty preal-sup-set-not-rat-set*
preal-sup-set-lemma3 preal-sup-set-lemma4)
done

lemma *preal-psup-le*:
$[|\ \forall X \in P.\ X \leq Y;\ \ x \in P\ |] \Longrightarrow x \leq \mathit{psup}\ P$
apply (*simp* (*no-asm-simp*) *add*: *preal-le-def*)
apply (*subgoal-tac* $P \neq \{\}$)
apply (*auto simp add*: *psup-def preal-sup*)
done

lemma *psup-le-ub*: $[|\ P \neq \{\}; \forall X \in P.\ X \leq Y\ |] \Longrightarrow \mathit{psup}\ P \leq Y$
apply (*simp* (*no-asm-simp*) *add*: *preal-le-def*)
apply (*simp add*: *psup-def preal-sup*)
apply (*auto simp add*: *preal-le-def*)

done

Supremum property

lemma *preal-complete*:
[| $P \neq \{\}$; $\forall X \in P.\ X \leq Y$ |] ==> $(\exists X \in P.\ Z < X) = (Z < psup\ P)$
apply (*simp add*: *preal-less-def psup-def preal-sup*)
apply (*auto simp add*: *preal-le-def*)
apply (*rename-tac U*)
apply (*cut-tac* $x = U$ **and** $y = Z$ **in** *linorder-less-linear*)
apply (*auto simp add*: *preal-less-def*)
done

51.13 The Embedding from *rat* into *preal*

lemma *preal-of-rat-add-lemma1*:
[|$x < y + z$; $0 < x$; $0 < y$|] ==> $x * y * inverse\ (y + z) < (y::rat)$
apply (*frule-tac* $c = y * inverse\ (y + z)$ **in** *mult-strict-right-mono*)
apply (*simp add*: *zero-less-mult-iff*)
apply (*simp add*: *mult-ac*)
done

lemma *preal-of-rat-add-lemma2*:
assumes $u < x + y$
and $0 < x$
and $0 < y$
and $0 < u$
shows $\exists v\ w::rat.\ w < y\ \&\ 0 < v\ \&\ v < x\ \&\ 0 < w\ \&\ u = v + w$
proof (*intro exI conjI*)
show $u * x * inverse(x+y) < x$ **using** *prems*
by (*simp add*: *preal-of-rat-add-lemma1*)
show $u * y * inverse(x+y) < y$ **using** *prems*
by (*simp add*: *preal-of-rat-add-lemma1 add-commute* [*of x*])
show $0 < u * x * inverse\ (x + y)$ **using** *prems*
by (*simp add*: *zero-less-mult-iff*)
show $0 < u * y * inverse\ (x + y)$ **using** *prems*
by (*simp add*: *zero-less-mult-iff*)
show $u = u * x * inverse\ (x + y) + u * y * inverse\ (x + y)$ **using** *prems*
by (*simp add*: *left-distrib* [*symmetric*] *right-distrib* [*symmetric*] *mult-ac*)
qed

lemma *preal-of-rat-add*:
[| $0 < x$; $0 < y$|]
==> $preal\text{-}of\text{-}rat\ ((x::rat) + y) = preal\text{-}of\text{-}rat\ x + preal\text{-}of\text{-}rat\ y$
apply (*unfold preal-of-rat-def preal-add-def*)
apply (*simp add*: *rat-mem-preal*)
apply (*rule-tac f* = *Abs-preal* **in** *arg-cong*)
apply (*auto simp add*: *add-set-def*)
apply (*blast dest*: *preal-of-rat-add-lemma2*)
done

lemma *preal-of-rat-mult-lemma1*:
  $[|x < y; 0 < x; 0 < z|] ==> x * z * inverse\ y < (z::rat)$
apply (*frule-tac* $c = z * inverse\ y$ **in** *mult-strict-right-mono*)
apply (*simp add*: *zero-less-mult-iff*)
apply (*subgoal-tac* $y * (z * inverse\ y) = z * (y * inverse\ y)$)
apply (*simp-all add*: *mult-ac*)
done

lemma *preal-of-rat-mult-lemma2*:
  assumes *xless*: $x < y * z$
    and *xpos*: $0 < x$
    and *ypos*: $0 < y$
  shows $x * z * inverse\ y * inverse\ z < (z::rat)$
proof −
  have $0 < y * z$ **using** *prems* **by** *simp*
  hence *zpos*: $0 < z$ **using** *prems* **by** (*simp add*: *zero-less-mult-iff*)
  have $x * z * inverse\ y * inverse\ z = x * inverse\ y * (z * inverse\ z)$
    by (*simp add*: *mult-ac*)
  also have $... = x/y$ **using** *zpos*
    by (*simp add*: *divide-inverse*)
  also from *xless* **have** $... < z$
    by (*simp add*: *pos-divide-less-eq* [*OF ypos*] *mult-commute*)
  finally show *?thesis* **.**
qed

lemma *preal-of-rat-mult-lemma3*:
  assumes *uless*: $u < x * y$
    and $0 < x$
    and $0 < y$
    and $0 < u$
  shows $\exists v\ w::rat.\ v < x\ \&\ w < y\ \&\ 0 < v\ \&\ 0 < w\ \&\ u = v * w$
proof −
  from *dense* [*OF uless*]
  obtain r **where** $u < r\ r < x * y$ **by** *blast*
  thus *?thesis*
  proof (*intro exI conjI*)
  show $u * x * inverse\ r < x$ **using** *prems*
    by (*simp add*: *preal-of-rat-mult-lemma1*)
  show $r * y * inverse\ x * inverse\ y < y$ **using** *prems*
    by (*simp add*: *preal-of-rat-mult-lemma2*)
  show $0 < u * x * inverse\ r$ **using** *prems*
    by (*simp add*: *zero-less-mult-iff*)
  show $0 < r * y * inverse\ x * inverse\ y$ **using** *prems*
    by (*simp add*: *zero-less-mult-iff*)
  have $u * x * inverse\ r * (r * y * inverse\ x * inverse\ y) =$
      $u * (r * inverse\ r) * (x * inverse\ x) * (y * inverse\ y)$
    by (*simp only*: *mult-ac*)
  thus $u = u * x * inverse\ r * (r * y * inverse\ x * inverse\ y)$ **using** *prems*

```
    by simp
  qed
qed

lemma preal-of-rat-mult:
     [| 0 < x; 0 < y|]
      ==> preal-of-rat ((x::rat) * y) = preal-of-rat x * preal-of-rat y
apply (unfold preal-of-rat-def preal-mult-def)
apply (simp add: rat-mem-preal)
apply (rule-tac f = Abs-preal in arg-cong)
apply (auto simp add: zero-less-mult-iff mult-strict-mono mult-set-def)
apply (blast dest: preal-of-rat-mult-lemma3)
done

lemma preal-of-rat-less-iff:
     [| 0 < x; 0 < y|] ==> (preal-of-rat x < preal-of-rat y) = (x < y)
by (force simp add: preal-of-rat-def preal-less-def rat-mem-preal)

lemma preal-of-rat-le-iff:
     [| 0 < x; 0 < y|] ==> (preal-of-rat x ≤ preal-of-rat y) = (x ≤ y)
by (simp add: preal-of-rat-less-iff linorder-not-less [symmetric])

lemma preal-of-rat-eq-iff:
     [| 0 < x; 0 < y|] ==> (preal-of-rat x = preal-of-rat y) = (x = y)
by (simp add: preal-of-rat-le-iff order-eq-iff)

end
```

52 RealDef: Defining the Reals from the Positive Reals

```
theory RealDef
imports PReal
uses (Tools/real-arith.ML)
begin

definition
  realrel   ::  ((preal * preal) * (preal * preal)) set where
  [code del]: realrel = {p. ∃ x1 y1 x2 y2. p = ((x1,y1),(x2,y2)) & x1+y2 = x2+y1}

typedef (Real)  real = UNIV//realrel
  by (auto simp add: quotient-def)

definition

  real-of-preal :: preal => real where
  [code del]: real-of-preal m = Abs-Real (realrel `` {(m + 1, 1)})
```

instantiation *real* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *inverse*, *ord*, *abs*, *sgn*}
begin

definition
real-zero-def [*code del*]: $0 = Abs\text{-}Real(realrel``\{(1, 1)\})$

definition
real-one-def [*code del*]: $1 = Abs\text{-}Real(realrel``\{(1 + 1, 1)\})$

definition
real-add-def [*code del*]: $z + w =$
$contents\ (\bigcup(x,y) \in Rep\text{-}Real(z).\ \bigcup(u,v) \in Rep\text{-}Real(w).$
$\{\ Abs\text{-}Real(realrel``\{(x+u,\ y+v)\})\ \})$

definition
real-minus-def [*code del*]: $-\ r = contents\ (\bigcup(x,y) \in Rep\text{-}Real(r).\ \{\ Abs\text{-}Real(realrel``\{(y,x)\})\ \})$

definition
real-diff-def [*code del*]: $r - (s::real) = r + -\ s$

definition
real-mult-def [*code del*]:
$z * w =$
$contents\ (\bigcup(x,y) \in Rep\text{-}Real(z).\ \bigcup(u,v) \in Rep\text{-}Real(w).$
$\{\ Abs\text{-}Real(realrel``\{(x*u + y*v,\ x*v + y*u)\})\ \})$

definition
real-inverse-def [*code del*]: $inverse\ (R::real) = (THE\ S.\ (R = 0\ \&\ S = 0) \mid S * R = 1)$

definition
real-divide-def [*code del*]: $R\ /\ (S::real) = R * inverse\ S$

definition
real-le-def [*code del*]: $z \leq (w::real) \longleftrightarrow$
$(\exists x\ y\ u\ v.\ x+v \leq u+y\ \&\ (x,y) \in Rep\text{-}Real\ z\ \&\ (u,v) \in Rep\text{-}Real\ w)$

definition
real-less-def [*code del*]: $x < (y::real) \longleftrightarrow x \leq y \wedge x \neq y$

definition
real-abs-def: $abs\ (r::real) = (if\ r < 0\ then\ -\ r\ else\ r)$

definition
real-sgn-def: $sgn\ (x::real) = (if\ x=0\ then\ 0\ else\ if\ 0<x\ then\ 1\ else\ -\ 1)$

instance ..

end

52.1 Equivalence relation over positive reals

lemma *preal-trans-lemma*:
assumes $x + y1 = x1 + y$
and $x + y2 = x2 + y$
shows $x1 + y2 = x2 + (y1::preal)$
proof −
have $(x1 + y2) + x = (x + y2) + x1$ **by** (*simp add*: *add-ac*)
also have $... = (x2 + y) + x1$ **by** (*simp add*: *prems*)
also have $... = x2 + (x1 + y)$ **by** (*simp add*: *add-ac*)
also have $... = x2 + (x + y1)$ **by** (*simp add*: *prems*)
also have $... = (x2 + y1) + x$ **by** (*simp add*: *add-ac*)
finally have $(x1 + y2) + x = (x2 + y1) + x$ **.**
thus *?thesis* **by** (*rule add-right-imp-eq*)
qed

lemma *realrel-iff* [*simp*]: $(((x1,y1),(x2,y2)) \in realrel) = (x1 + y2 = x2 + y1)$
by (*simp add*: *realrel-def*)

lemma *equiv-realrel*: *equiv UNIV realrel*
apply (*auto simp add*: *equiv-def refl-on-def sym-def trans-def realrel-def*)
apply (*blast dest*: *preal-trans-lemma*)
done

Reduces equality of equivalence classes to the *realrel* relation: $(realrel\ ``\ \{x\} = realrel\ ``\ \{y\}) = ((x,\ y) \in realrel)$

lemmas *equiv-realrel-iff* =
eq-equiv-class-iff [*OF equiv-realrel UNIV-I UNIV-I*]

declare *equiv-realrel-iff* [*simp*]

lemma *realrel-in-real* [*simp*]: *realrel``{(x,y)}*: *Real*
by (*simp add*: *Real-def realrel-def quotient-def*, *blast*)

declare *Abs-Real-inject* [*simp*]
declare *Abs-Real-inverse* [*simp*]

Case analysis on the representation of a real number as an equivalence class of pairs of positive reals.

lemma *eq-Abs-Real* [*case-names Abs-Real*, *cases type*: *real*]:
$(!!x\ y.\ z = Abs\text{-}Real(realrel``\{(x,y)\}) ==> P) ==> P$
apply (*rule Rep-Real* [*of z*, *unfolded Real-def*, *THEN quotientE*])
apply (*drule arg-cong* [**where** *f*=*Abs-Real*])

apply (*auto simp add*: *Rep-Real-inverse*)
done

52.2 Addition and Subtraction

lemma *real-add-congruent2-lemma*:
  [|*a* + *ba* = *aa* + *b*; *ab* + *bc* = *ac* + *bb*|]
  ==> *a* + *ab* + (*ba* + *bc*) = *aa* + *ac* + (*b* + (*bb*::*preal*))
apply (*simp add*: *add-assoc*)
apply (*rule add-left-commute* [*of ab*, *THEN ssubst*])
apply (*simp add*: *add-assoc* [*symmetric*])
apply (*simp add*: *add-ac*)
done

lemma *real-add*:
  Abs-Real (*realrel*``{(*x*,*y*)}) + *Abs-Real* (*realrel*``{(*u*,*v*)}) =
  Abs-Real (*realrel*``{(*x*+*u*, *y*+*v*)})
proof −
  have (λ*z w*. (λ(*x*,*y*). (λ(*u*,*v*). {*Abs-Real* (*realrel* `` {(*x*+*u*, *y*+*v*)})}) *w*) *z*)
      respects2 realrel
    by (*simp add*: *congruent2-def*, *blast intro*: *real-add-congruent2-lemma*)
  thus *?thesis*
    by (*simp add*: *real-add-def UN-UN-split-split-eq*
            UN-equiv-class2 [*OF equiv-realrel equiv-realrel*])
qed

lemma *real-minus*: − *Abs-Real*(*realrel*``{(*x*,*y*)}) = *Abs-Real*(*realrel* `` {(*y*,*x*)})
proof −
  have (λ(*x*,*y*). {*Abs-Real* (*realrel*``{(*y*,*x*)})}) *respects realrel*
    by (*simp add*: *congruent-def add-commute*)
  thus *?thesis*
    by (*simp add*: *real-minus-def UN-equiv-class* [*OF equiv-realrel*])
qed

instance *real* :: *ab-group-add*
proof
  fix *x y z* :: *real*
  show (*x* + *y*) + *z* = *x* + (*y* + *z*)
    by (*cases x*, *cases y*, *cases z*, *simp add*: *real-add add-assoc*)
  show *x* + *y* = *y* + *x*
    by (*cases x*, *cases y*, *simp add*: *real-add add-commute*)
  show *0* + *x* = *x*
    by (*cases x*, *simp add*: *real-add real-zero-def add-ac*)
  show − *x* + *x* = *0*
    by (*cases x*, *simp add*: *real-minus real-add real-zero-def add-commute*)
  show *x* − *y* = *x* + − *y*
    by (*simp add*: *real-diff-def*)
qed

52.3 Multiplication

lemma *real-mult-congruent2-lemma*:
!!(*x1*::*preal*). [| *x1* + *y2* = *x2* + *y1* |] ==>
x * *x1* + *y* * *y1* + (*x* * *y2* + *y* * *x2*) =
x * *x2* + *y* * *y2* + (*x* * *y1* + *y* * *x1*)
apply (*simp add*: *add-left-commute add-assoc* [*symmetric*])
apply (*simp add*: *add-assoc right-distrib* [*symmetric*])
apply (*simp add*: *add-commute*)
done

lemma *real-mult-congruent2*:
(%*p1 p2*.
(%(*x1*,*y1*). (%(*x2*,*y2*).
{ *Abs-Real* (*realrel*``{(*x1**x2* + *y1**y2*, *x1**y2*+*y1**x2*)}) }) *p2*) *p1*)
respects2 realrel
apply (*rule congruent2-commuteI* [*OF equiv-realrel*], *clarify*)
apply (*simp add*: *mult-commute add-commute*)
apply (*auto simp add*: *real-mult-congruent2-lemma*)
done

lemma *real-mult*:
Abs-Real((*realrel*``{(*x1*,*y1*)})) * *Abs-Real*((*realrel*``{(*x2*,*y2*)})) =
Abs-Real(*realrel* `` {(*x1**x2*+*y1**y2*,*x1**y2*+*y1**x2*)})
by (*simp add*: *real-mult-def UN-UN-split-split-eq*
UN-equiv-class2 [*OF equiv-realrel equiv-realrel real-mult-congruent2*])

lemma *real-mult-commute*: (*z*::*real*) * *w* = *w* * *z*
by (*cases z*, *cases w*, *simp add*: *real-mult add-ac mult-ac*)

lemma *real-mult-assoc*: ((*z1*::*real*) * *z2*) * *z3* = *z1* * (*z2* * *z3*)
apply (*cases z1*, *cases z2*, *cases z3*)
apply (*simp add*: *real-mult algebra-simps*)
done

lemma *real-mult-1*: (*1*::*real*) * *z* = *z*
apply (*cases z*)
apply (*simp add*: *real-mult real-one-def algebra-simps*)
done

lemma *real-add-mult-distrib*: ((*z1*::*real*) + *z2*) * *w* = (*z1* * *w*) + (*z2* * *w*)
apply (*cases z1*, *cases z2*, *cases w*)
apply (*simp add*: *real-add real-mult algebra-simps*)
done

one and zero are distinct

lemma *real-zero-not-eq-one*: *0* ≠ (*1*::*real*)
proof −
have (*1*::*preal*) < *1* + *1*
by (*simp add*: *preal-self-less-add-left*)

thus *?thesis*
by (*simp add*: *real-zero-def real-one-def*)
qed

instance *real* :: *comm-ring-1*
proof
fix *x y z* :: *real*
show $(x * y) * z = x * (y * z)$ **by** (*rule real-mult-assoc*)
show $x * y = y * x$ **by** (*rule real-mult-commute*)
show $1 * x = x$ **by** (*rule real-mult-1*)
show $(x + y) * z = x * z + y * z$ **by** (*rule real-add-mult-distrib*)
show $0 \neq (1::real)$ **by** (*rule real-zero-not-eq-one*)
qed

52.4 Inverse and Division

lemma *real-zero-iff*: *Abs-Real* (*realrel* `` {(*x*, *x*)}) = *0*
by (*simp add*: *real-zero-def add-commute*)

Instead of using an existential quantifier and constructing the inverse within the proof, we could define the inverse explicitly.

lemma *real-mult-inverse-left-ex*: $x \neq 0 ==> \exists y.\ y*x = (1::real)$
apply (*simp add*: *real-zero-def real-one-def*, *cases x*)
apply (*cut-tac x = xa* **and** *y = y* **in** *linorder-less-linear*)
apply (*auto dest*!: *less-add-left-Ex simp add*: *real-zero-iff*)
apply (*rule-tac*
x = Abs-Real (*realrel*``{(*1*, *inverse* (*D*) + *1*)})
in *exI*)
apply (*rule-tac* [*2*]
x = Abs-Real (*realrel*``{(*inverse* (*D*) + *1*, *1*)})
in *exI*)
apply (*auto simp add*: *real-mult preal-mult-inverse-right algebra-simps*)
done

lemma *real-mult-inverse-left*: $x \neq 0 ==> inverse(x)*x = (1::real)$
apply (*simp add*: *real-inverse-def*)
apply (*drule real-mult-inverse-left-ex*, *safe*)
apply (*rule theI*, *assumption*, *rename-tac z*)
apply (*subgoal-tac* $(z * x) * y = z * (x * y)$)
apply (*simp add*: *mult-commute*)
apply (*rule mult-assoc*)
done

52.5 The Real Numbers form a Field

instance *real* :: *field*
proof
fix *x y z* :: *real*
show $x \neq 0 ==> inverse\ x * x = 1$ **by** (*rule real-mult-inverse-left*)

show x / $y = x * inverse\ y$ **by** (*simp add*: *real-divide-def*)
qed

Inverse of zero! Useful to simplify certain equations

lemma *INVERSE-ZERO*: $inverse\ 0 = (0::real)$
by (*simp add*: *real-inverse-def*)

instance *real* :: *division-by-zero*
proof
show $inverse\ 0 = (0::real)$ **by** (*rule INVERSE-ZERO*)
qed

52.6 The $\leq$ Ordering

lemma *real-le-refl*: $w \leq (w::real)$
by (*cases w*, *force simp add*: *real-le-def*)

The arithmetic decision procedure is not set up for type preal. This lemma is currently unused, but it could simplify the proofs of the following two lemmas.

lemma *preal-eq-le-imp-le*:
assumes *eq*: $a+b = c+d$ **and** *le*: $c \leq a$
shows $b \leq (d::preal)$
proof −
have $c+d \leq a+d$ **by** (*simp add*: *prems*)
hence $a+b \leq a+d$ **by** (*simp add*: *prems*)
thus $b \leq d$ **by** *simp*
qed

lemma *real-le-lemma*:
assumes *l*: $u1 + v2 \leq u2 + v1$
and $x1 + v1 = u1 + y1$
and $x2 + v2 = u2 + y2$
shows $x1 + y2 \leq x2 + (y1::preal)$
proof −
have $(x1+v1) + (u2+y2) = (u1+y1) + (x2+v2)$ **by** (*simp add*: *prems*)
hence $(x1+y2) + (u2+v1) = (x2+y1) + (u1+v2)$ **by** (*simp add*: *add-ac*)
also have ... $\leq (x2+y1) + (u2+v1)$ **by** (*simp add*: *prems*)
finally show *?thesis* **by** *simp*
qed

lemma *real-le*:
$(Abs\text{-}Real(realrel``\{(x1,y1)\}) \leq Abs\text{-}Real(realrel``\{(x2,y2)\})) =$
$(x1 + y2 \leq x2 + y1)$
apply (*simp add*: *real-le-def*)
apply (*auto intro*: *real-le-lemma*)
done

lemma *real-le-anti-sym*: $[|\ z \leq w;\ w \leq z\ |] ==> z = (w::real)$

by (*cases z*, *cases w*, *simp add*: *real-le*)

lemma *real-trans-lemma*:
  assumes $x + v \leq u + y$
    and $u + v' \leq u' + v$
    and $x2 + v2 = u2 + y2$
  shows $x + v' \leq u' + (y::preal)$
proof −
  have $(x+v') + (u+v) = (x+v) + (u+v')$ **by** (*simp add*: *add-ac*)
  also have ... $\leq (u+y) + (u+v')$ **by** (*simp add*: *prems*)
  also have ... $\leq (u+y) + (u'+v)$ **by** (*simp add*: *prems*)
  also have ... $= (u'+y) + (u+v)$ **by** (*simp add*: *add-ac*)
  finally show *?thesis* **by** *simp*
qed

lemma *real-le-trans*: $[\![i \leq j; j \leq k]\!] ==> i \leq (k::real)$
apply (*cases i*, *cases j*, *cases k*)
apply (*simp add*: *real-le*)
apply (*blast intro*: *real-trans-lemma*)
done

instance *real* :: *order*
proof
  fix $u\ v$ :: *real*
  show $u < v \longleftrightarrow u \leq v \wedge \neg v \leq u$
    by (*auto simp add*: *real-less-def intro*: *real-le-anti-sym*)
qed (*assumption* | *rule real-le-refl real-le-trans real-le-anti-sym*)+

lemma *real-le-linear*: $(z::real) \leq w \mid w \leq z$
apply (*cases z*, *cases w*)
apply (*auto simp add*: *real-le real-zero-def add-ac*)
done

instance *real* :: *linorder*
  by (*intro-classes*, *rule real-le-linear*)

lemma *real-le-eq-diff*: $(x \leq y) = (x-y \leq (0::real))$
apply (*cases x*, *cases y*)
apply (*auto simp add*: *real-le real-zero-def real-diff-def real-add real-minus*
                add-ac)
apply (*simp-all add*: *add-assoc* [*symmetric*])
done

lemma *real-add-left-mono*:
  assumes *le*: $x \leq y$ **shows** $z + x \leq z + (y::real)$
proof −
  have $z + x - (z + y) = (z + -z) + (x - y)$

```
    by (simp add: algebra-simps)
  with le show ?thesis
    by (simp add: real-le-eq-diff[of x] real-le-eq-diff[of z+x] diff-minus)
qed

lemma real-sum-gt-zero-less: (0 < S + (-W::real)) ==> (W < S)
by (simp add: linorder-not-le [symmetric] real-le-eq-diff [of S] diff-minus)

lemma real-less-sum-gt-zero: (W < S) ==> (0 < S + (-W::real))
by (simp add: linorder-not-le [symmetric] real-le-eq-diff [of S] diff-minus)

lemma real-mult-order: [| 0 < x; 0 < y |] ==> (0::real) < x * y
apply (cases x, cases y)
apply (simp add: linorder-not-le [where 'a = real, symmetric]
                 linorder-not-le [where 'a = preal]
                  real-zero-def real-le real-mult)
  — Reduce to the (simpler) ≤ relation
apply (auto dest!: less-add-left-Ex
     simp add: algebra-simps preal-self-less-add-left)
done

lemma real-mult-less-mono2: [| (0::real) < z; x < y |] ==> z * x < z * y
apply (rule real-sum-gt-zero-less)
apply (drule real-less-sum-gt-zero [of x y])
apply (drule real-mult-order, assumption)
apply (simp add: right-distrib)
done

instantiation real :: distrib-lattice
begin

definition
  (inf :: real ⇒ real ⇒ real) = min

definition
  (sup :: real ⇒ real ⇒ real) = max

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

end
```

52.7 The Reals Form an Ordered Field

```
instance real :: ordered-field
proof
  fix x y z :: real
  show x ≤ y ==> z + x ≤ z + y by (rule real-add-left-mono)
  show x < y ==> 0 < z ==> z * x < z * y by (rule real-mult-less-mono2)
```

show $|x| = (if\ x < 0\ then\ -x\ else\ x)$ **by** (*simp only*: *real-abs-def*)
show *sgn x* = (*if x=0 then 0 else if 0<x then 1 else − 1*)
by (*simp only*: *real-sgn-def*)
qed

instance *real* :: *lordered-ab-group-add* **..**

The function *real-of-preal* requires many proofs, but it seems to be essential for proving completeness of the reals from that of the positive reals.

lemma *real-of-preal-add*:
real-of-preal ((*x*::*preal*) + *y*) = *real-of-preal x* + *real-of-preal y*
by (*simp add*: *real-of-preal-def real-add algebra-simps*)

lemma *real-of-preal-mult*:
real-of-preal ((*x*::*preal*) ∗ *y*) = *real-of-preal x*∗ *real-of-preal y*
by (*simp add*: *real-of-preal-def real-mult algebra-simps*)

Gleason prop 9-4.4 p 127

lemma *real-of-preal-trichotomy*:
∃ *m*. (*x*::*real*) = *real-of-preal m* | *x* = *0* | *x* = −(*real-of-preal m*)
apply (*simp add*: *real-of-preal-def real-zero-def*, *cases x*)
apply (*auto simp add*: *real-minus add-ac*)
apply (*cut-tac x* = *x* **and** *y* = *y* **in** *linorder-less-linear*)
apply (*auto dest*!: *less-add-left-Ex simp add*: *add-assoc* [*symmetric*])
done

lemma *real-of-preal-leD*:
real-of-preal m1 ≤ *real-of-preal m2* ==> *m1* ≤ *m2*
by (*simp add*: *real-of-preal-def real-le*)

lemma *real-of-preal-lessI*: *m1* < *m2* ==> *real-of-preal m1* < *real-of-preal m2*
by (*auto simp add*: *real-of-preal-leD linorder-not-le* [*symmetric*])

lemma *real-of-preal-lessD*:
real-of-preal m1 < *real-of-preal m2* ==> *m1* < *m2*
by (*simp add*: *real-of-preal-def real-le linorder-not-le* [*symmetric*])

lemma *real-of-preal-less-iff* [*simp*]:
(*real-of-preal m1* < *real-of-preal m2*) = (*m1* < *m2*)
by (*blast intro*: *real-of-preal-lessI real-of-preal-lessD*)

lemma *real-of-preal-le-iff*:
(*real-of-preal m1* ≤ *real-of-preal m2*) = (*m1* ≤ *m2*)
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *real-of-preal-zero-less*: *0* < *real-of-preal m*
apply (*insert preal-self-less-add-left* [*of 1 m*])
apply (*auto simp add*: *real-zero-def real-of-preal-def*
real-less-def real-le-def add-ac)

apply (*rule-tac x=m + 1* **in** *exI*, *rule-tac x=1* **in** *exI*)
apply (*simp add*: *add-ac*)
done

lemma *real-of-preal-minus-less-zero*: $- \text{real-of-preal } m < 0$
by (*simp add*: *real-of-preal-zero-less*)

lemma *real-of-preal-not-minus-gt-zero*: $\sim\ 0 < - \text{real-of-preal } m$
proof −
  from *real-of-preal-minus-less-zero*
  show *?thesis* **by** (*blast dest*: *order-less-trans*)
qed

52.8 Theorems About the Ordering

lemma *real-gt-zero-preal-Ex*: $(0 < x) = (\exists y.\ x = \text{real-of-preal } y)$
apply (*auto simp add*: *real-of-preal-zero-less*)
apply (*cut-tac x = x* **in** *real-of-preal-trichotomy*)
apply (*blast elim*!: *real-of-preal-not-minus-gt-zero* [*THEN notE*])
done

lemma *real-gt-preal-preal-Ex*:
    $\text{real-of-preal } z < x \Longrightarrow \exists y.\ x = \text{real-of-preal } y$
by (*blast dest*!: *real-of-preal-zero-less* [*THEN order-less-trans*]
          intro: *real-gt-zero-preal-Ex* [*THEN iffD1*])

lemma *real-ge-preal-preal-Ex*:
    $\text{real-of-preal } z \leq x \Longrightarrow \exists y.\ x = \text{real-of-preal } y$
by (*blast dest*: *order-le-imp-less-or-eq real-gt-preal-preal-Ex*)

lemma *real-less-all-preal*: $y \leq 0 \Longrightarrow \forall x.\ y < \text{real-of-preal } x$
by (*auto elim*: *order-le-imp-less-or-eq* [*THEN disjE*]
          intro: *real-of-preal-zero-less* [*THEN* [*2*] *order-less-trans*]
          simp add: *real-of-preal-zero-less*)

lemma *real-less-all-real2*: $\sim\ 0 < y \Longrightarrow \forall x.\ y < \text{real-of-preal } x$
by (*blast intro*!: *real-less-all-preal linorder-not-less* [*THEN iffD1*])

52.9 More Lemmas

lemma *real-mult-left-cancel*: $(c::real) \neq 0 \Longrightarrow (c*a=c*b) = (a=b)$
by *auto*

lemma *real-mult-right-cancel*: $(c::real) \neq 0 \Longrightarrow (a*c=b*c) = (a=b)$
by *auto*

lemma *real-mult-less-iff1* [*simp*]: $(0::real) < z \Longrightarrow (x*z < y*z) = (x < y)$
  by (*force elim*: *order-less-asym*
          simp add: *Ring-and-Field.mult-less-cancel-right*)

lemma *real-mult-le-cancel-iff1* [*simp*]: (*0*::*real*) < *z* ==> (*x*∗*z* ≤ *y*∗*z*) = (*x*≤*y*)
apply (*simp add*: *mult-le-cancel-right*)
apply (*blast intro*: *elim*: *order-less-asym*)
done

lemma *real-mult-le-cancel-iff2* [*simp*]: (*0*::*real*) < *z* ==> (*z*∗*x* ≤ *z*∗*y*) = (*x*≤*y*)
by(*simp add*:*mult-commute*)

lemma *real-inverse-gt-one*: [| (*0*::*real*) < *x*; *x* < *1* |] ==> *1* < *inverse x*
by (*simp add*: *one-less-inverse-iff*)

52.10 Embedding numbers into the Reals

abbreviation
real-of-nat :: *nat* ⇒ *real*
where
real-of-nat ≡ *of-nat*

abbreviation
real-of-int :: *int* ⇒ *real*
where
real-of-int ≡ *of-int*

abbreviation
real-of-rat :: *rat* ⇒ *real*
where
real-of-rat ≡ *of-rat*

consts

real :: *'a* => *real*

defs (**overloaded**)
real-of-nat-def [*code unfold*]: *real* == *real-of-nat*
real-of-int-def [*code unfold*]: *real* == *real-of-int*

lemma *real-eq-of-nat*: *real* = *of-nat*
unfolding *real-of-nat-def* **..**

lemma *real-eq-of-int*: *real* = *of-int*
unfolding *real-of-int-def* **..**

lemma *real-of-int-zero* [*simp*]: *real* (*0*::*int*) = *0*
by (*simp add*: *real-of-int-def*)

lemma *real-of-one* [*simp*]: *real* (*1*::*int*) = (*1*::*real*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-add* [*simp*]: *real*(*x* + *y*) = *real* (*x*::*int*) + *real y*

by (*simp add*: *real-of-int-def*)

lemma *real-of-int-minus* [*simp*]: *real*(−*x*) = −*real* (*x*::*int*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-diff* [*simp*]: *real*(*x* − *y*) = *real* (*x*::*int*) − *real y*
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-mult* [*simp*]: *real*(*x* ∗ *y*) = *real* (*x*::*int*) ∗ *real y*
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-setsum* [*simp*]: *real* ((*SUM x*:*A*. *f x*)::*int*) = (*SUM x*:*A*. *real*(*f x*))
  apply (*subst real-eq-of-int*)+
  apply (*rule of-int-setsum*)
done

lemma *real-of-int-setprod* [*simp*]: *real* ((*PROD x*:*A*. *f x*)::*int*) =
    (*PROD x*:*A*. *real*(*f x*))
  apply (*subst real-eq-of-int*)+
  apply (*rule of-int-setprod*)
done

lemma *real-of-int-zero-cancel* [*simp*, *algebra*, *presburger*]: (*real x* = *0*) = (*x* = (*0*::*int*))
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-inject* [*iff*, *algebra*, *presburger*]: (*real* (*x*::*int*) = *real y*) = (*x* = *y*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-less-iff* [*iff*, *presburger*]: (*real* (*x*::*int*) < *real y*) = (*x* < *y*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-le-iff* [*simp*, *presburger*]: (*real* (*x*::*int*) ≤ *real y*) = (*x* ≤ *y*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-gt-zero-cancel-iff* [*simp*, *presburger*]: (*0* < *real* (*n*::*int*)) = (*0* < *n*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-ge-zero-cancel-iff* [*simp*, *presburger*]: (*0* <= *real* (*n*::*int*)) = (*0* <= *n*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-lt-zero-cancel-iff* [*simp*, *presburger*]: (*real* (*n*::*int*) < *0*) = (*n* < *0*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-le-zero-cancel-iff* [*simp*, *presburger*]: (*real* (*n*::*int*) <= *0*) = (*n* <= *0*)
by (*simp add*: *real-of-int-def*)

lemma *real-of-int-abs* [*simp*]: *real* (*abs x*) = *abs*(*real* (*x*::*int*))
by (*auto simp add*: *abs-if*)

lemma *int-less-real-le*: ((*n*::*int*) < *m*) = (*real n* + *1* <= *real m*)
  apply (*subgoal-tac real n* + *1* = *real* (*n* + *1*))
  apply (*simp del*: *real-of-int-add*)
  apply *auto*
done

lemma *int-le-real-less*: ((*n*::*int*) <= *m*) = (*real n* < *real m* + *1*)
  apply (*subgoal-tac real m* + *1* = *real* (*m* + *1*))
  apply (*simp del*: *real-of-int-add*)
  apply *simp*
done

lemma *real-of-int-div-aux*: *d* ~= *0* ==> (*real* (*x*::*int*)) / (*real d*) =
    real (*x div d*) + (*real* (*x mod d*)) / (*real d*)
proof −
  assume *d* ~= *0*
  have *x* = (*x div d*) * *d* + *x mod d*
    by *auto*
  then have *real x* = *real* (*x div d*) * *real d* + *real*(*x mod d*)
    by (*simp only*: *real-of-int-mult* [*THEN sym*] *real-of-int-add* [*THEN sym*])
  then have *real x* / *real d* = ... / *real d*
    by *simp*
  then show *?thesis*
    by (*auto simp add*: *add-divide-distrib algebra-simps prems*)
qed

lemma *real-of-int-div*: (*d*::*int*) ~= *0* ==> *d dvd n* ==>
    real(*n div d*) = *real n* / *real d*
  apply (*frule real-of-int-div-aux* [*of d n*])
  apply *simp*
  apply (*simp add*: *dvd-eq-mod-eq-0*)
done

lemma *real-of-int-div2*:
  0 <= *real* (*n*::*int*) / *real* (*x*) − *real* (*n div x*)
  apply (*case-tac x* = *0*)
  apply *simp*
  apply (*case-tac 0* < *x*)
  apply (*simp add*: *algebra-simps*)
  apply (*subst real-of-int-div-aux*)
  apply *simp*
  apply *simp*

apply (*subst zero-le-divide-iff*)
apply *auto*
apply (*simp add*: *algebra-simps*)
apply (*subst real-of-int-div-aux*)
apply *simp*
apply *simp*
apply (*subst zero-le-divide-iff*)
apply *auto*
done

lemma *real-of-int-div3*:
real (*n*::*int*) / *real* (*x*) − *real* (*n div x*) <= *1*
apply(*case-tac x = 0*)
apply *simp*
apply (*simp add*: *algebra-simps*)
apply (*subst real-of-int-div-aux*)
apply *assumption*
apply *simp*
apply (*subst divide-le-eq*)
apply *clarsimp*
apply (*rule conjI*)
apply (*rule impI*)
apply (*rule order-less-imp-le*)
apply *simp*
apply (*rule impI*)
apply (*rule order-less-imp-le*)
apply *simp*
done

lemma *real-of-int-div4*: *real* (*n div x*) <= *real* (*n*::*int*) / *real x*
by (*insert real-of-int-div2* [*of n x*], *simp*)

52.11 Embedding the Naturals into the Reals

lemma *real-of-nat-zero* [*simp*]: *real* (*0*::*nat*) = *0*
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-1* [*simp*]: *real* (*1*::*nat*) = *1*
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-one* [*simp*]: *real* (*Suc 0*) = (*1*::*real*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-add* [*simp*]: *real* (*m* + *n*) = *real* (*m*::*nat*) + *real n*
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-Suc*: *real* (*Suc n*) = *real n* + (*1*::*real*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-less-iff* [*iff*]:
(*real* (*n*::*nat*) < *real m*) = (*n* < *m*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-le-iff* [*iff*]: (*real* (*n*::*nat*) ≤ *real m*) = (*n* ≤ *m*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-ge-zero* [*iff*]: *0* ≤ *real* (*n*::*nat*)
by (*simp add*: *real-of-nat-def zero-le-imp-of-nat*)

lemma *real-of-nat-Suc-gt-zero*: *0* < *real* (*Suc n*)
by (*simp add*: *real-of-nat-def del*: *of-nat-Suc*)

lemma *real-of-nat-mult* [*simp*]: *real* (*m* * *n*) = *real* (*m*::*nat*) * *real n*
by (*simp add*: *real-of-nat-def of-nat-mult*)

lemma *real-of-nat-setsum* [*simp*]: *real* ((*SUM x*:*A*. *f x*)::*nat*) =
(*SUM x*:*A*. *real*(*f x*))
apply (*subst real-eq-of-nat*)+
apply (*rule of-nat-setsum*)
done

lemma *real-of-nat-setprod* [*simp*]: *real* ((*PROD x*:*A*. *f x*)::*nat*) =
(*PROD x*:*A*. *real*(*f x*))
apply (*subst real-eq-of-nat*)+
apply (*rule of-nat-setprod*)
done

lemma *real-of-card*: *real* (*card A*) = *setsum* (%*x*.*1*) *A*
apply (*subst card-eq-setsum*)
apply (*subst real-of-nat-setsum*)
apply *simp*
done

lemma *real-of-nat-inject* [*iff*]: (*real* (*n*::*nat*) = *real m*) = (*n* = *m*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-zero-iff* [*iff*]: (*real* (*n*::*nat*) = *0*) = (*n* = *0*)
by (*simp add*: *real-of-nat-def*)

lemma *real-of-nat-diff*: *n* ≤ *m* ==> *real* (*m* - *n*) = *real* (*m*::*nat*) - *real n*
by (*simp add*: *add*: *real-of-nat-def of-nat-diff*)

lemma *real-of-nat-gt-zero-cancel-iff* [*simp*]: (*0* < *real* (*n*::*nat*)) = (*0* < *n*)
by (*auto simp*: *real-of-nat-def*)

lemma *real-of-nat-le-zero-cancel-iff* [*simp*]: (*real* (*n*::*nat*) ≤ *0*) = (*n* = *0*)
by (*simp add*: *add*: *real-of-nat-def*)

lemma *not-real-of-nat-less-zero* [*simp*]: ~ *real* (*n*::*nat*) < *0*
by (*simp add*: *add*: *real-of-nat-def*)

lemma *real-of-nat-ge-zero-cancel-iff* [*simp*]: (*0* ≤ *real* (*n*::*nat*))
by (*simp add*: *add*: *real-of-nat-def*)

lemma *nat-less-real-le*: ((*n*::*nat*) < *m*) = (*real n* + *1* <= *real m*)
 apply (*subgoal-tac real n* + *1* = *real* (*Suc n*))
 apply *simp*
 apply (*auto simp add*: *real-of-nat-Suc*)
done

lemma *nat-le-real-less*: ((*n*::*nat*) <= *m*) = (*real n* < *real m* + *1*)
 apply (*subgoal-tac real m* + *1* = *real* (*Suc m*))
 apply (*simp add*: *less-Suc-eq-le*)
 apply (*simp add*: *real-of-nat-Suc*)
done

lemma *real-of-nat-div-aux*: *0* < *d* ==> (*real* (*x*::*nat*)) / (*real d*) =
 real (*x div d*) + (*real* (*x mod d*)) / (*real d*)
proof −
 assume *0* < *d*
 have *x* = (*x div d*) ∗ *d* + *x mod d*
 by *auto*
 then have *real x* = *real* (*x div d*) ∗ *real d* + *real*(*x mod d*)
 by (*simp only*: *real-of-nat-mult* [*THEN sym*] *real-of-nat-add* [*THEN sym*])
 then have *real x* / *real d* = ... / *real d*
 by *simp*
 then show *?thesis*
 by (*auto simp add*: *add-divide-distrib algebra-simps prems*)
qed

lemma *real-of-nat-div*: *0* < (*d*::*nat*) ==> *d dvd n* ==>
 real(*n div d*) = *real n* / *real d*
 apply (*frule real-of-nat-div-aux* [*of d n*])
 apply *simp*
 apply (*subst dvd-eq-mod-eq-0* [*THEN sym*])
 apply *assumption*
done

lemma *real-of-nat-div2*:
 0 <= *real* (*n*::*nat*) / *real* (*x*) − *real* (*n div x*)
apply(*case-tac x* = *0*)
 apply (*simp*)
apply (*simp add*: *algebra-simps*)
apply (*subst real-of-nat-div-aux*)
 apply *simp*
apply *simp*

apply (*subst zero-le-divide-iff*)
apply *simp*
done

lemma *real-of-nat-div3*:
  real (*n*::*nat*) / *real* (*x*) − *real* (*n div x*) <= *1*
apply(*case-tac x = 0*)
apply (*simp*)
apply (*simp add*: *algebra-simps*)
apply (*subst real-of-nat-div-aux*)
 apply *simp*
apply *simp*
done

lemma *real-of-nat-div4*: *real* (*n div x*) <= *real* (*n*::*nat*) / *real x*
by (*insert real-of-nat-div2* [*of n x*], *simp*)

lemma *real-of-int-real-of-nat*: *real* (*int n*) = *real n*
by (*simp add*: *real-of-nat-def real-of-int-def int-eq-of-nat*)

lemma *real-of-int-of-nat-eq* [*simp*]: *real* (*of-nat n* :: *int*) = *real n*
by (*simp add*: *real-of-int-def real-of-nat-def*)

lemma *real-nat-eq-real* [*simp*]: *0* <= *x* ==> *real*(*nat x*) = *real x*
  apply (*subgoal-tac real*(*int*(*nat x*)) = *real*(*nat x*))
  apply *force*
  apply (*simp only*: *real-of-int-real-of-nat*)
done

52.12 Rationals

lemma *Rats-real-nat*[*simp*]: *real*(*n*::*nat*) ∈ ℚ
by (*simp add*: *real-eq-of-nat*)

lemma *Rats-eq-int-div-int*:
  ℚ = { *real*(*i*::*int*)/*real*(*j*::*int*) |*i j*. *j* ≠ *0*} (**is** - = *?S*)
proof
  show ℚ ⊆ *?S*
  proof
    fix *x*::*real* **assume** *x* : ℚ
    then obtain *r* **where** *x* = *of-rat r* **unfolding** *Rats-def* **..**
    have *of-rat r* : *?S*
      by (*cases r*)(*auto simp add*:*of-rat-rat real-eq-of-int*)
    thus *x* : *?S* **using** ‹*x* = *of-rat r*› **by** *simp*
  qed
next
  show *?S* ⊆ ℚ
  proof(*auto simp*:*Rats-def*)

```
    fix i j :: int assume j ≠ 0
    hence real i / real j = of-rat(Fract i j)
      by (simp add:of-rat-rat real-eq-of-int)
    thus real i / real j ∈ range of-rat by blast
  qed
qed

lemma Rats-eq-int-div-nat:
  ℚ = { real(i::int)/real(n::nat) |i n. n ≠ 0}
proof(auto simp:Rats-eq-int-div-int)
  fix i j::int assume j ≠ 0
  show EX (i'::int) (n::nat). real i/real j = real i'/real n ∧ 0<n
  proof cases
    assume j>0
    hence real i/real j = real i/real(nat j) ∧ 0<nat j
      by (simp add: real-eq-of-int real-eq-of-nat of-nat-nat)
    thus ?thesis by blast
  next
    assume ~ j>0
    hence real i/real j = real(−i)/real(nat(−j)) ∧ 0<nat(−j) using ‹j≠0›
      by (simp add: real-eq-of-int real-eq-of-nat of-nat-nat)
    thus ?thesis by blast
  qed
next
  fix i::int and n::nat assume 0 < n
  hence real i/real n = real i/real(int n) ∧ int n ≠ 0 by simp
  thus ∃(i'::int) j::int. real i/real n = real i'/real j ∧ j ≠ 0 by blast
qed

lemma Rats-abs-nat-div-natE:
  assumes x ∈ ℚ
  obtains m n where n ≠ 0 and |x| = real m / real n and gcd m n = 1
proof −
  from ‹x ∈ ℚ› obtain i::int and n::nat where n ≠ 0 and x = real i / real n
    by(auto simp add: Rats-eq-int-div-nat)
  hence |x| = real(nat(abs i)) / real n by simp
  then obtain m :: nat where x-rat: |x| = real m / real n by blast
  let ?gcd = gcd m n
  from ‹n≠0› have gcd: ?gcd ≠ 0 by (simp add: gcd-zero)
  let ?k = m div ?gcd
  let ?l = n div ?gcd
  let ?gcd' = gcd ?k ?l
  have ?gcd dvd m .. then have gcd-k: ?gcd * ?k = m
    by (rule dvd-mult-div-cancel)
  have ?gcd dvd n .. then have gcd-l: ?gcd * ?l = n
    by (rule dvd-mult-div-cancel)
  from ‹n≠0› and gcd-l have ?l ≠ 0 by (auto iff del: neq0-conv)
  moreover
  have |x| = real ?k / real ?l
```

proof −
from *gcd* **have** *real ?k / real ?l* =
*real (?gcd * ?k) / real (?gcd * ?l)* **by** *simp*
also from *gcd-k* **and** *gcd-l* **have** ... = *real m / real n* **by** *simp*
also from *x-rat* **have** ... = |*x*| **..**
finally show *?thesis* **..**
qed
moreover
have *?gcd' = 1*
proof −
have *?gcd * ?gcd' = gcd (?gcd * ?k) (?gcd * ?l)*
by (*rule gcd-mult-distrib2*)
with *gcd-k gcd-l* **have** *?gcd * ?gcd' = ?gcd* **by** *simp*
with *gcd* **show** *?thesis* **by** *simp*
qed
ultimately show *?thesis* **..**
qed

52.13 Numerals and Arithmetic

instantiation *real* :: *number-ring*
begin

definition
real-number-of-def [*code del*]: *number-of w = real-of-int w*

instance
by *intro-classes* (*simp add*: *real-number-of-def*)

end

lemma [*code unfold*, *symmetric*, *code post*]:
number-of k = real-of-int (*number-of k*)
unfolding *number-of-is-id real-number-of-def* **..**

Collapse applications of *real* to *number-of*

lemma *real-number-of* [*simp*]: *real* (*number-of v* :: *int*) = *number-of v*
by (*simp add*: *real-of-int-def of-int-number-of-eq*)

lemma *real-of-nat-number-of* [*simp*]:
real (*number-of v* :: *nat*) =
(*if neg* (*number-of v* :: *int*) *then 0*
else (*number-of v* :: *real*))
by (*simp add*: *real-of-int-real-of-nat* [*symmetric*] *int-nat-number-of*)

use *Tools/real-arith.ML*
declaration ⟪ *K real-arith-setup* ⟫

52.14 Simprules combining x+y and 0: ARE THEY NEEDED?

Needed in this non-standard form by Hyperreal/Transcendental

lemma *real-0-le-divide-iff*:
$((0::real) \leq x/y) = ((x \leq 0 \mid 0 \leq y) \,\&\, (0 \leq x \mid y \leq 0))$
by (*simp add*: *real-divide-def zero-le-mult-iff*, *auto*)

lemma *real-add-minus-iff* [*simp*]: $(x + -a = (0::real)) = (x=a)$
by *arith*

lemma *real-add-eq-0-iff*: $(x+y = (0::real)) = (y = -x)$
by *auto*

lemma *real-add-less-0-iff*: $(x+y < (0::real)) = (y < -x)$
by *auto*

lemma *real-0-less-add-iff*: $((0::real) < x+y) = (-x < y)$
by *auto*

lemma *real-add-le-0-iff*: $(x+y \leq (0::real)) = (y \leq -x)$
by *auto*

lemma *real-0-le-add-iff*: $((0::real) \leq x+y) = (-x \leq y)$
by *auto*

52.14.1 Density of the Reals

lemma *real-lbound-gt-zero*:
$[\!| (0::real) < d1; 0 < d2 |\!] ==> \exists e.\ 0 < e \,\&\, e < d1 \,\&\, e < d2$
apply (*rule-tac x* = $(min\ d1\ d2)\ /2$ **in** *exI*)
apply (*simp add*: *min-def*)
done

Similar results are proved in *Ring-and-Field*

lemma *real-less-half-sum*: $x < y ==> x < (x+y)\ /\ (2::real)$
by *auto*

lemma *real-gt-half-sum*: $x < y ==> (x+y)/(2::real) < y$
by *auto*

52.15 Absolute Value Function for the Reals

lemma *abs-minus-add-cancel*: $abs(x + (-y)) = abs\ (y + (-(x::real)))$
by (*simp add*: *abs-if*)

lemma *abs-le-interval-iff*: $(abs\ x \leq r) = (-r \leq x \,\&\, x \leq (r::real))$
by (*force simp add*: *OrderedGroup.abs-le-iff*)

lemma *abs-add-one-gt-zero* [*simp*]: (*0*::*real*) < *1* + *abs*(*x*)
by (*simp add*: *abs-if*)

lemma *abs-real-of-nat-cancel* [*simp*]: *abs* (*real x*) = *real* (*x*::*nat*)
by (*rule abs-of-nonneg* [*OF real-of-nat-ge-zero*])

lemma *abs-add-one-not-less-self* [*simp*]: ~ *abs*(*x*) + (*1*::*real*) < *x*
by *simp*

lemma *abs-sum-triangle-ineq*: *abs* ((*x*::*real*) + *y* + (−*l* + −*m*)) ≤ *abs*(*x* + −*l*) + *abs*(*y* + −*m*)
by *simp*

instance *real* :: *lordered-ring*
proof
  fix *a*::*real*
  show *abs a* = *sup a* (−*a*)
    by (*auto simp add*: *real-abs-def sup-real-def*)
qed

52.16 Implementation of rational real numbers

definition *Ratreal* :: *rat* ⇒ *real* **where**
  [*simp*]: *Ratreal* = *of-rat*

code-datatype *Ratreal*

lemma *Ratreal-number-collapse* [*code post*]:
  Ratreal 0 = *0*
  Ratreal 1 = *1*
  Ratreal (*number-of k*) = *number-of k*
by *simp-all*

lemma *zero-real-code* [*code*, *code unfold*]:
  0 = *Ratreal 0*
by *simp*

lemma *one-real-code* [*code*, *code unfold*]:
  1 = *Ratreal 1*
by *simp*

lemma *number-of-real-code* [*code unfold*]:
  number-of k = *Ratreal* (*number-of k*)
by *simp*

lemma *Ratreal-number-of-quotient* [*code post*]:
  Ratreal (*number-of r*) / *Ratreal* (*number-of s*) = *number-of r* / *number-of s*
by *simp*

lemma *Ratreal-number-of-quotient2* [*code post*]:
Ratreal (*number-of r* / *number-of s*) = *number-of r* / *number-of s*
unfolding *Ratreal-number-of-quotient* [*symmetric*] *Ratreal-def of-rat-divide* **..**

instantiation *real* :: *eq*
begin

definition *eq-class.eq* (*x*::*real*) *y* ⟷ *x* − *y* = *0*

instance by *default* (*simp add*: *eq-real-def*)

lemma *real-eq-code* [*code*]: *eq-class.eq* (*Ratreal x*) (*Ratreal y*) ⟷ *eq-class.eq x y*
by (*simp add*: *eq-real-def eq*)

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

end

lemma *real-less-eq-code* [*code*]: *Ratreal x* ≤ *Ratreal y* ⟷ *x* ≤ *y*
by (*simp add*: *of-rat-less-eq*)

lemma *real-less-code* [*code*]: *Ratreal x* < *Ratreal y* ⟷ *x* < *y*
by (*simp add*: *of-rat-less*)

lemma *real-plus-code* [*code*]: *Ratreal x* + *Ratreal y* = *Ratreal* (*x* + *y*)
by (*simp add*: *of-rat-add*)

lemma *real-times-code* [*code*]: *Ratreal x* ∗ *Ratreal y* = *Ratreal* (*x* ∗ *y*)
by (*simp add*: *of-rat-mult*)

lemma *real-uminus-code* [*code*]: − *Ratreal x* = *Ratreal* (− *x*)
by (*simp add*: *of-rat-minus*)

lemma *real-minus-code* [*code*]: *Ratreal x* − *Ratreal y* = *Ratreal* (*x* − *y*)
by (*simp add*: *of-rat-diff*)

lemma *real-inverse-code* [*code*]: *inverse* (*Ratreal x*) = *Ratreal* (*inverse x*)
by (*simp add*: *of-rat-inverse*)

lemma *real-divide-code* [*code*]: *Ratreal x* / *Ratreal y* = *Ratreal* (*x* / *y*)
by (*simp add*: *of-rat-divide*)

Setup for SML code generator

types-code
real ((*int* ∗/ *int*))
attach (*term-of*) ⟪
fun term-of-real (*p*, *q*) =

```
  let
    val rT = HOLogic.realT
  in
    if q = 1 orelse p = 0 then HOLogic.mk-number rT p
    else @{term op / :: real ⇒ real ⇒ real} $
      HOLogic.mk-number rT p $ HOLogic.mk-number rT q
  end;
⟩⟩
attach (test) ⟨⟨
fun gen-real i =
  let
    val p = random-range 0 i;
    val q = random-range 1 (i + 1);
    val g = Integer.gcd p q;
    val p' = p div g;
    val q' = q div g;
    val r = (if one-of [true, false] then p' else ~ p',
      if p' = 0 then 0 else q')
  in
    (r, fn () => term-of-real r)
  end;
⟩⟩

consts-code
  Ratreal ((-))

consts-code
  of-int :: int ⇒ real (⟨module⟩real'-of'-int)
attach ⟨⟨
fun real-of-int 0 = (0, 0)
  | real-of-int i = (i, 1);
⟩⟩

end
```

53 RComplete: Completeness of the Reals; Floor and Ceiling Functions

```
theory RComplete
imports Lubs RealDef
begin

lemma real-sum-of-halves: x/2 + x/2 = (x::real)
  by simp
```

53.1 Completeness of Positive Reals

Supremum property for the set of positive reals

Let P be a non-empty set of positive reals, with an upper bound y. Then P has a least upper bound (written S).

FIXME: Can the premise be weakened to $\forall x \in P.\ x \leq y$?

lemma *posreal-complete*:
  assumes *positive-P*: $\forall x \in P.\ (0::real) < x$
    and *not-empty-P*: $\exists x.\ x \in P$
    and *upper-bound-Ex*: $\exists y.\ \forall x \in P.\ x<y$
  shows $\exists S.\ \forall y.\ (\exists x \in P.\ y < x) = (y < S)$
proof (*rule exI*, *rule allI*)
  fix y
  let *?pP* $= \{w.\ real\text{-}of\text{-}preal\ w \in P\}$

  show $(\exists x{\in}P.\ y < x) = (y < real\text{-}of\text{-}preal\ (psup\ ?pP))$
  proof (*cases* $0 < y$)
    assume *neg-y*: $\neg\ 0 < y$
    show *?thesis*
    proof
      assume $\exists x{\in}P.\ y < x$
      have $\forall x.\ y < real\text{-}of\text{-}preal\ x$
        using *neg-y* **by** (*rule real-less-all-real2*)
      thus $y < real\text{-}of\text{-}preal\ (psup\ ?pP)$ **..**
    next
      assume $y < real\text{-}of\text{-}preal\ (psup\ ?pP)$
      obtain x **where** *x-in-P*: $x \in P$ **using** *not-empty-P* **..**
      hence $0 < x$ **using** *positive-P* **by** *simp*
      hence $y < x$ **using** *neg-y* **by** *simp*
      thus $\exists x \in P.\ y < x$ **using** *x-in-P* **..**
    qed
  next
    assume *pos-y*: $0 < y$

    then obtain *py* **where** *y-is-py*: $y = real\text{-}of\text{-}preal\ py$
      by (*auto simp add*: *real-gt-zero-preal-Ex*)

    obtain a **where** $a \in P$ **using** *not-empty-P* **..**
    with *positive-P* **have** *a-pos*: $0 < a$ **..**
    then obtain *pa* **where** $a = real\text{-}of\text{-}preal\ pa$
      by (*auto simp add*: *real-gt-zero-preal-Ex*)
    hence $pa \in ?pP$ **using** ‹$a \in P$› **by** *auto*
    hence *pP-not-empty*: $?pP \neq \{\}$ **by** *auto*

    obtain *sup* **where** *sup*: $\forall x \in P.\ x < sup$
      using *upper-bound-Ex* **..**
    from *this* **and** ‹$a \in P$› **have** $a < sup$ **..**
    hence $0 < sup$ **using** *a-pos* **by** *arith*

```
    then obtain possup where sup = real-of-preal possup
      by (auto simp add: real-gt-zero-preal-Ex)
    hence ∀ X ∈ ?pP. X ≤ possup
      using sup by (auto simp add: real-of-preal-lessI)
    with pP-not-empty have psup: ⋀Z. (∃ X ∈ ?pP. Z < X) = (Z < psup ?pP)
      by (rule preal-complete)

    show ?thesis
    proof
      assume ∃ x ∈ P. y < x
      then obtain x where x-in-P: x ∈ P and y-less-x: y < x ..
      hence 0 < x using pos-y by arith
      then obtain px where x-is-px: x = real-of-preal px
        by (auto simp add: real-gt-zero-preal-Ex)

      have py-less-X: ∃ X ∈ ?pP. py < X
      proof
        show py < px using y-is-py and x-is-px and y-less-x
          by (simp add: real-of-preal-lessI)
        show px ∈ ?pP using x-in-P and x-is-px by simp
      qed

      have (∃ X ∈ ?pP. py < X) ==> (py < psup ?pP)
        using psup by simp
      hence py < psup ?pP using py-less-X by simp
      thus y < real-of-preal (psup {w. real-of-preal w ∈ P})
        using y-is-py and pos-y by (simp add: real-of-preal-lessI)
    next
      assume y-less-psup: y < real-of-preal (psup ?pP)

      hence py < psup ?pP using y-is-py
        by (simp add: real-of-preal-lessI)
      then obtain X where py-less-X: py < X and X-in-pP: X ∈ ?pP
        using psup by auto
      then obtain x where x-is-X: x = real-of-preal X
        by (simp add: real-gt-zero-preal-Ex)
      hence y < x using py-less-X and y-is-py
        by (simp add: real-of-preal-lessI)

      moreover have x ∈ P using x-is-X and X-in-pP by simp

      ultimately show ∃ x ∈ P. y < x ..
    qed
  qed
qed
```

Completeness properties using *isUb*, *isLub* etc.

```
lemma real-isLub-unique: [| isLub R S x; isLub R S y |] ==> x = (y::real)
  apply (frule isLub-isUb)
```

apply (*frule-tac* $x = y$ **in** *isLub-isUb*)
apply (*blast intro*!: *order-antisym dest*!: *isLub-le-isUb*)
done

Completeness theorem for the positive reals (again).

lemma *posreals-complete*:
assumes *positive-S*: $\forall x \in S.\ 0 < x$
and *not-empty-S*: $\exists x.\ x \in S$
and *upper-bound-Ex*: $\exists u.\ isUb\ (UNIV::real\ set)\ S\ u$
shows $\exists t.\ isLub\ (UNIV::real\ set)\ S\ t$
proof
let *?pS* $= \{w.\ real\text{-}of\text{-}preal\ w \in S\}$

obtain u **where** $isUb\ UNIV\ S\ u$ **using** *upper-bound-Ex* **..**
hence *sup*: $\forall x \in S.\ x \leq u$ **by** (*simp add*: *isUb-def setle-def*)

obtain x **where** *x-in-S*: $x \in S$ **using** *not-empty-S* **..**
hence *x-gt-zero*: $0 < x$ **using** *positive-S* **by** *simp*
have $x \leq u$ **using** *sup* **and** *x-in-S* **..**
hence $0 < u$ **using** *x-gt-zero* **by** *arith*

then obtain *pu* **where** *u-is-pu*: $u = real\text{-}of\text{-}preal\ pu$
by (*auto simp add*: *real-gt-zero-preal-Ex*)

have *pS-less-pu*: $\forall pa \in ?pS.\ pa \leq pu$
proof
fix *pa*
assume $pa \in ?pS$
then obtain a **where** $a \in S$ **and** $a = real\text{-}of\text{-}preal\ pa$
by *simp*
moreover hence $a \leq u$ **using** *sup* **by** *simp*
ultimately show $pa \leq pu$
using *sup* **and** *u-is-pu* **by** (*simp add*: *real-of-preal-le-iff*)
qed

have $\forall y \in S.\ y \leq real\text{-}of\text{-}preal\ (psup\ ?pS)$
proof
fix y
assume *y-in-S*: $y \in S$
hence $0 < y$ **using** *positive-S* **by** *simp*
then obtain *py* **where** *y-is-py*: $y = real\text{-}of\text{-}preal\ py$
by (*auto simp add*: *real-gt-zero-preal-Ex*)
hence *py-in-pS*: $py \in ?pS$ **using** *y-in-S* **by** *simp*
with *pS-less-pu* **have** $py \leq psup\ ?pS$
by (*rule preal-psup-le*)
thus $y \leq real\text{-}of\text{-}preal\ (psup\ ?pS)$
using *y-is-py* **by** (*simp add*: *real-of-preal-le-iff*)
qed

```
  moreover {
    fix x
    assume x-ub-S: ∀ y∈S. y ≤ x
    have real-of-preal (psup ?pS) ≤ x
    proof −
      obtain s where s-in-S: s ∈ S using not-empty-S ..
      hence s-pos: 0 < s using positive-S by simp

      hence ∃ ps. s = real-of-preal ps by (simp add: real-gt-zero-preal-Ex)
      then obtain ps where s-is-ps: s = real-of-preal ps ..
      hence ps-in-pS: ps ∈ {w. real-of-preal w ∈ S} using s-in-S by simp

      from x-ub-S have s ≤ x using s-in-S ..
      hence 0 < x using s-pos by simp
      hence ∃ px. x = real-of-preal px by (simp add: real-gt-zero-preal-Ex)
      then obtain px where x-is-px: x = real-of-preal px ..

      have ∀ pe ∈ ?pS. pe ≤ px
      proof
        fix pe
        assume pe ∈ ?pS
        hence real-of-preal pe ∈ S by simp
        hence real-of-preal pe ≤ x using x-ub-S by simp
        thus pe ≤ px using x-is-px by (simp add: real-of-preal-le-iff)
      qed

      moreover have ?pS ≠ {} using ps-in-pS by auto
      ultimately have (psup ?pS) ≤ px by (simp add: psup-le-ub)
    thus real-of-preal (psup ?pS) ≤ x using x-is-px by (simp add: real-of-preal-le-iff)
    qed
  }
  ultimately show isLub UNIV S (real-of-preal (psup ?pS))
    by (simp add: isLub-def leastP-def isUb-def setle-def setge-def)
qed
```

reals Completeness (again!)

```
lemma reals-complete:
  assumes notempty-S: ∃ X. X ∈ S
    and exists-Ub: ∃ Y. isUb (UNIV::real set) S Y
  shows ∃ t. isLub (UNIV :: real set) S t
proof −
  obtain X where X-in-S: X ∈ S using notempty-S ..
  obtain Y where Y-isUb: isUb (UNIV::real set) S Y
    using exists-Ub ..
  let ?SHIFT = {z. ∃ x ∈S. z = x + (−X) + 1} ∩ {x. 0 < x}

  {
    fix x
    assume isUb (UNIV::real set) S x
```

```
  hence S-le-x: ∀ y ∈ S. y <= x
    by (simp add: isUb-def setle-def)
  {
    fix s
    assume s ∈ {z. ∃x∈S. z = x + - X + 1}
    hence ∃ x ∈ S. s = x + -X + 1 ..
    then obtain x1 where x1 ∈ S and s = x1 + (-X) + 1 ..
    moreover hence x1 ≤ x using S-le-x by simp
    ultimately have s ≤ x + - X + 1 by arith
  }
  then have isUb (UNIV::real set) ?SHIFT (x + (-X) + 1)
    by (auto simp add: isUb-def setle-def)
} note S-Ub-is-SHIFT-Ub = this

hence isUb UNIV ?SHIFT (Y + (-X) + 1) using Y-isUb by simp
hence ∃ Z. isUb UNIV ?SHIFT Z ..
moreover have ∀ y ∈ ?SHIFT. 0 < y by auto
moreover have shifted-not-empty: ∃ u. u ∈ ?SHIFT
  using X-in-S and Y-isUb by auto
ultimately obtain t where t-is-Lub: isLub UNIV ?SHIFT t
  using posreals-complete [of ?SHIFT] by blast

show ?thesis
proof
  show isLub UNIV S (t + X + (-1))
  proof (rule isLubI2)
    {
      fix x
      assume isUb (UNIV::real set) S x
      hence isUb (UNIV::real set) (?SHIFT) (x + (-X) + 1)
        using S-Ub-is-SHIFT-Ub by simp
      hence t ≤ (x + (-X) + 1)
        using t-is-Lub by (simp add: isLub-le-isUb)
      hence t + X + -1 ≤ x by arith
    }
    then show (t + X + -1) <=* Collect (isUb UNIV S)
      by (simp add: setgeI)
  next
    show isUb UNIV S (t + X + -1)
    proof -
      {
        fix y
        assume y-in-S: y ∈ S
        have y ≤ t + X + -1
        proof -
          obtain u where u-in-shift: u ∈ ?SHIFT using shifted-not-empty ..
          hence ∃ x ∈ S. u = x + - X + 1 by simp
          then obtain x where x-and-u: u = x + - X + 1 ..
        have u-le-t: u ≤ t using u-in-shift and t-is-Lub by (simp add: isLubD2)
```

```
          show ?thesis
          proof cases
            assume y ≤ x
            moreover have x = u + X + − 1 using x-and-u by arith
            moreover have u + X + − 1 ≤ t + X + −1 using u-le-t by arith
            ultimately show y ≤ t + X + −1 by arith
          next
            assume ~(y ≤ x)
            hence x-less-y: x < y by arith

           have x + (−X) + 1 ∈ ?SHIFT using x-and-u and u-in-shift by simp
            hence 0 < x + (−X) + 1 by simp
            hence 0 < y + (−X) + 1 using x-less-y by arith
            hence y + (−X) + 1 ∈ ?SHIFT using y-in-S by simp
            hence y + (−X) + 1 ≤ t using t-is-Lub by (simp add: isLubD2)
            thus ?thesis by simp
          qed
        qed
       }
       then show ?thesis by (simp add: isUb-def setle-def)
     qed
   qed
  qed
qed
```

53.2 The Archimedean Property of the Reals

```
theorem reals-Archimedean:
  assumes x-pos: 0 < x
  shows ∃ n. inverse (real (Suc n)) < x
proof (rule ccontr)
  assume contr: ¬ ?thesis
  have ∀ n. x * real (Suc n) <= 1
  proof
    fix n
    from contr have x ≤ inverse (real (Suc n))
      by (simp add: linorder-not-less)
    hence x ≤ (1 / (real (Suc n)))
      by (simp add: inverse-eq-divide)
    moreover have 0 ≤ real (Suc n)
      by (rule real-of-nat-ge-zero)
    ultimately have x * real (Suc n) ≤ (1 / real (Suc n)) * real (Suc n)
      by (rule mult-right-mono)
    thus x * real (Suc n) ≤ 1 by simp
  qed
  hence {z. ∃ n. z = x * (real (Suc n))} *<= 1
    by (simp add: setle-def, safe, rule spec)
  hence isUb (UNIV::real set) {z. ∃ n. z = x * (real (Suc n))} 1
```

```
    by (simp add: isUbI)
  hence ∃ Y. isUb (UNIV::real set) {z. ∃ n. z = x* (real (Suc n))} Y ..
  moreover have ∃ X. X ∈ {z. ∃ n. z = x* (real (Suc n))} by auto
  ultimately have ∃ t. isLub UNIV {z. ∃ n. z = x * real (Suc n)} t
    by (simp add: reals-complete)
  then obtain t where
    t-is-Lub: isLub UNIV {z. ∃ n. z = x * real (Suc n)} t ..

  have ∀ n::nat. x * real n ≤ t + - x
  proof
    fix n
    from t-is-Lub have x * real (Suc n) ≤ t
      by (simp add: isLubD2)
    hence  x * (real n) + x ≤ t
      by (simp add: right-distrib real-of-nat-Suc)
    thus  x * (real n) ≤ t + - x by arith
  qed

  hence ∀ m. x * real (Suc m) ≤ t + - x by simp
  hence {z. ∃ n. z = x * (real (Suc n))}  *<= (t + - x)
    by (auto simp add: setle-def)
  hence isUb (UNIV::real set) {z. ∃ n. z = x * (real (Suc n))} (t + (-x))
    by (simp add: isUbI)
  hence t ≤ t + - x
    using t-is-Lub by (simp add: isLub-le-isUb)
  thus False using x-pos by arith
qed
```

There must be other proofs, e.g. *Suc* of the largest integer in the cut representing x.

```
lemma reals-Archimedean2: ∃ n. (x::real) < real (n::nat)
proof cases
  assume x ≤ 0
  hence x < real (1::nat) by simp
  thus ?thesis ..
next
  assume ¬ x ≤ 0
  hence x-greater-zero: 0 < x by simp
  hence 0 < inverse x by simp
  then obtain n where inverse (real (Suc n)) < inverse x
    using reals-Archimedean by blast
  hence inverse (real (Suc n)) * x < inverse x * x
    using x-greater-zero by (rule mult-strict-right-mono)
  hence inverse (real (Suc n)) * x < 1
    using x-greater-zero by simp
  hence real (Suc n) * (inverse (real (Suc n)) * x) < real (Suc n) * 1
    by (rule mult-strict-left-mono) simp
  hence x < real (Suc n)
    by (simp add: algebra-simps)
```

thus $\exists (n::nat).\ x < real\ n$ **..**
qed

instance *real* :: *archimedean-field*
proof
fix r :: *real*
obtain n :: *nat* **where** $r < real\ n$
using *reals-Archimedean2* **..**
then have $r \leq of\text{-}int\ (int\ n)$
unfolding *real-eq-of-nat* **by** *simp*
then show $\exists z.\ r \leq of\text{-}int\ z$ **..**
qed

lemma *reals-Archimedean3*:
assumes *x-greater-zero*: $0 < x$
shows $\forall (y::real).\ \exists (n::nat).\ y < real\ n * x$
unfolding *real-of-nat-def* **using** ‹$0 < x$›
by (*auto intro*: *ex-less-of-nat-mult*)

lemma *reals-Archimedean6*:
$0 \leq r ==> \exists (n::nat).\ real\ (n - 1) \leq r\ \&\ r < real\ (n)$
unfolding *real-of-nat-def*
apply (*rule exI* [**where** x=*nat* (*floor r + 1*)])
apply (*insert floor-correct* [*of r*])
apply (*simp add*: *nat-add-distrib of-nat-nat*)
done

lemma *reals-Archimedean6a*: $0 \leq r ==> \exists n.\ real\ (n) \leq r\ \&\ r < real\ (Suc\ n)$
by (*drule reals-Archimedean6*) *auto*

lemma *reals-Archimedean-6b-int*:
$0 \leq r ==> \exists n::int.\ real\ n \leq r\ \&\ r < real\ (n+1)$
unfolding *real-of-int-def* **by** (*rule floor-exists*)

lemma *reals-Archimedean-6c-int*:
$r < 0 ==> \exists n::int.\ real\ n \leq r\ \&\ r < real\ (n+1)$
unfolding *real-of-int-def* **by** (*rule floor-exists*)

53.3 Density of the Rational Reals in the Reals

This density proof is due to Stefan Richter and was ported by TN. The original source is *Real Analysis* by H.L. Royden. It employs the Archimedean property of the reals.

lemma *Rats-dense-in-nn-real*: **fixes** x::*real*
assumes $0 \leq x$ **and** $x<y$ **shows** $\exists r \in \mathbb{Q}.\ x<r \wedge r<y$
proof −
from ‹$x<y$› **have** $0 < y-x$ **by** *simp*
with *reals-Archimedean* **obtain** q::*nat*
where q: *inverse* (*real q*) $< y-x$ **and** $0 <$ *real q* **by** *auto*

def $p \equiv LEAST\ n.\ \ y \leq real\ (Suc\ n)/real\ q$
from *reals-Archimedean2* **obtain** n::*nat* **where** $y * real\ q < real\ n$ **by** *auto*
with ‹$0 < real\ q$› **have** *ex*: $y \leq real\ n/real\ q$ (**is** *?P n*)
by (*simp add*: *pos-less-divide-eq*[*THEN sym*])
also from *assms* **have** $\neg\ y \leq real\ (0::nat)\ /\ real\ q$ **by** *simp*
ultimately have *main*: $(LEAST\ n.\ y \leq real\ n/real\ q) = Suc\ p$
by (*unfold p-def*) (*rule Least-Suc*)
also from *ex* **have** *?P* $(LEAST\ x.\ ?P\ x)$ **by** (*rule LeastI*)
ultimately have *suc*: $y \leq real\ (Suc\ p)\ /\ real\ q$ **by** *simp*
def $r \equiv real\ p/real\ q$
have $x = y-(y-x)$ **by** *simp*
also from *suc q* **have** $\ldots < real\ (Suc\ p)/real\ q - inverse\ (real\ q)$ **by** *arith*
also have $\ldots = real\ p\ /\ real\ q$
by (*simp only*: *inverse-eq-divide real-diff-def real-of-nat-Suc*
minus-divide-left add-divide-distrib[*THEN sym*]) *simp*
finally have $x<r$ **by** (*unfold r-def*)
have $p<Suc\ p$ **.. also note** *main*[*THEN sym*]
finally have $\neg$ *?P p* **by** (*rule not-less-Least*)
hence $r<y$ **by** (*simp add*: *r-def*)
from *r-def* **have** $r \in \mathbb{Q}$ **by** *simp*
with ‹$x<r$› ‹$r<y$› **show** *?thesis* **by** *fast*
qed

theorem *Rats-dense-in-real*: **fixes** $x\ y :: real$
assumes $x<y$ **shows** $\exists r \in \mathbb{Q}.\ \ x<r \wedge r<y$
proof −
from *reals-Archimedean2* **obtain** n::*nat* **where** $-x < real\ n$ **by** *auto*
hence $0 \leq x + real\ n$ **by** *arith*
also from ‹$x<y$› **have** $x + real\ n < y + real\ n$ **by** *arith*
ultimately have $\exists r \in \mathbb{Q}.\ x + real\ n < r \wedge r < y + real\ n$
by(*rule Rats-dense-in-nn-real*)
then obtain r **where** $r \in \mathbb{Q}$ **and** *r2*: $x + real\ n < r$
and *r3*: $r < y + real\ n$
by *blast*
have $r - real\ n = r + real\ (int\ n)/real\ (-1::int)$ **by** *simp*
also from ‹$r \in \mathbb{Q}$› **have** $r + real\ (int\ n)/real\ (-1::int) \in \mathbb{Q}$ **by** *simp*
also from *r2* **have** $x < r - real\ n$ **by** *arith*
moreover from *r3* **have** $r - real\ n < y$ **by** *arith*
ultimately show *?thesis* **by** *fast*
qed

53.4 Floor and Ceiling Functions from the Reals to the Integers

lemma *number-of-less-real-of-int-iff* [*simp*]:
$((number\text{-}of\ n) < real\ (m::int)) = (number\text{-}of\ n < m)$
apply *auto*
apply (*rule real-of-int-less-iff* [*THEN iffD1*])
apply (*drule-tac* [*2*] *real-of-int-less-iff* [*THEN iffD2*], *auto*)

done

lemma *number-of-less-real-of-int-iff2* [*simp*]:
(*real* (*m*::*int*) < (*number-of n*)) = (*m* < *number-of n*)
apply *auto*
apply (*rule real-of-int-less-iff* [*THEN iffD1*])
apply (*drule-tac* [*2*] *real-of-int-less-iff* [*THEN iffD2*], *auto*)
done

lemma *number-of-le-real-of-int-iff* [*simp*]:
((*number-of n*) ≤ *real* (*m*::*int*)) = (*number-of n* ≤ *m*)
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *number-of-le-real-of-int-iff2* [*simp*]:
(*real* (*m*::*int*) ≤ (*number-of n*)) = (*m* ≤ *number-of n*)
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *floor-real-of-nat-zero*: *floor* (*real* (*0*::*nat*)) = *0*
by *auto*

lemma *floor-real-of-nat* [*simp*]: *floor* (*real* (*n*::*nat*)) = *int n*
unfolding *real-of-nat-def* **by** *simp*

lemma *floor-minus-real-of-nat* [*simp*]: *floor* (− *real* (*n*::*nat*)) = − *int n*
unfolding *real-of-nat-def* **by** (*simp add*: *floor-minus*)

lemma *floor-real-of-int* [*simp*]: *floor* (*real* (*n*::*int*)) = *n*
unfolding *real-of-int-def* **by** *simp*

lemma *floor-minus-real-of-int* [*simp*]: *floor* (− *real* (*n*::*int*)) = − *n*
unfolding *real-of-int-def* **by** (*simp add*: *floor-minus*)

lemma *real-lb-ub-int*: ∃ *n*::*int*. *real n* ≤ *r* & *r* < *real* (*n*+*1*)
unfolding *real-of-int-def* **by** (*rule floor-exists*)

lemma *lemma-floor*:
assumes *a1*: *real m* ≤ *r* **and** *a2*: *r* < *real n* + *1*
shows *m* ≤ (*n*::*int*)
proof −
have *real m* < *real n* + *1* **using** *a1 a2* **by** (*rule order-le-less-trans*)
also have ... = *real* (*n* + *1*) **by** *simp*
finally have *m* < *n* + *1* **by** (*simp only*: *real-of-int-less-iff*)
thus *?thesis* **by** *arith*
qed

lemma *real-of-int-floor-le* [*simp*]: *real* (*floor r*) ≤ *r*
unfolding *real-of-int-def* **by** (*rule of-int-floor-le*)

lemma *lemma-floor2*: *real n* < *real* (*x*::*int*) + *1* ==> *n* ≤ *x*

by (*auto intro*: *lemma-floor*)

lemma *real-of-int-floor-cancel* [*simp*]:
  (*real* (*floor x*) = *x*) = (∃ *n*::*int*. *x* = *real n*)
 using *floor-real-of-int* **by** *metis*

lemma *floor-eq*: [| *real n* < *x*; *x* < *real n* + *1* |] ==> *floor x* = *n*
 unfolding *real-of-int-def* **using** *floor-unique* [*of n x*] **by** *simp*

lemma *floor-eq2*: [| *real n* ≤ *x*; *x* < *real n* + *1* |] ==> *floor x* = *n*
 unfolding *real-of-int-def* **by** (*rule floor-unique*)

lemma *floor-eq3*: [| *real n* < *x*; *x* < *real* (*Suc n*) |] ==> *nat*(*floor x*) = *n*
apply (*rule inj-int* [*THEN injD*])
apply (*simp add*: *real-of-nat-Suc*)
apply (*simp add*: *real-of-nat-Suc floor-eq floor-eq* [**where** *n* = *int n*])
done

lemma *floor-eq4*: [| *real n* ≤ *x*; *x* < *real* (*Suc n*) |] ==> *nat*(*floor x*) = *n*
apply (*drule order-le-imp-less-or-eq*)
apply (*auto intro*: *floor-eq3*)
done

lemma *floor-number-of-eq*:
   floor(*number-of n* :: *real*) = (*number-of n* :: *int*)
 by (*rule floor-number-of*)

lemma *real-of-int-floor-ge-diff-one* [*simp*]: *r* − *1* ≤ *real*(*floor r*)
 unfolding *real-of-int-def* **using** *floor-correct* [*of r*] **by** *simp*

lemma *real-of-int-floor-gt-diff-one* [*simp*]: *r* − *1* < *real*(*floor r*)
 unfolding *real-of-int-def* **using** *floor-correct* [*of r*] **by** *simp*

lemma *real-of-int-floor-add-one-ge* [*simp*]: *r* ≤ *real*(*floor r*) + *1*
 unfolding *real-of-int-def* **using** *floor-correct* [*of r*] **by** *simp*

lemma *real-of-int-floor-add-one-gt* [*simp*]: *r* < *real*(*floor r*) + *1*
 unfolding *real-of-int-def* **using** *floor-correct* [*of r*] **by** *simp*

lemma *le-floor*: *real a* <= *x* ==> *a* <= *floor x*
 unfolding *real-of-int-def* **by** (*simp add*: *le-floor-iff*)

lemma *real-le-floor*: *a* <= *floor x* ==> *real a* <= *x*
 unfolding *real-of-int-def* **by** (*simp add*: *le-floor-iff*)

lemma *le-floor-eq*: (*a* <= *floor x*) = (*real a* <= *x*)
 unfolding *real-of-int-def* **by** (*rule le-floor-iff*)

lemma *le-floor-eq-number-of*:

(*number-of n <= floor x*) = (*number-of n <= x*)
by (*rule number-of-le-floor*)

lemma *le-floor-eq-zero*: (*0 <= floor x*) = (*0 <= x*)
by (*rule zero-le-floor*)

lemma *le-floor-eq-one*: (*1 <= floor x*) = (*1 <= x*)
by (*rule one-le-floor*)

lemma *floor-less-eq*: (*floor x < a*) = (*x < real a*)
unfolding *real-of-int-def* **by** (*rule floor-less-iff*)

lemma *floor-less-eq-number-of*:
(*floor x < number-of n*) = (*x < number-of n*)
by (*rule floor-less-number-of*)

lemma *floor-less-eq-zero*: (*floor x < 0*) = (*x < 0*)
by (*rule floor-less-zero*)

lemma *floor-less-eq-one*: (*floor x < 1*) = (*x < 1*)
by (*rule floor-less-one*)

lemma *less-floor-eq*: (*a < floor x*) = (*real a + 1 <= x*)
unfolding *real-of-int-def* **by** (*rule less-floor-iff*)

lemma *less-floor-eq-number-of*:
(*number-of n < floor x*) = (*number-of n + 1 <= x*)
by (*rule number-of-less-floor*)

lemma *less-floor-eq-zero*: (*0 < floor x*) = (*1 <= x*)
by (*rule zero-less-floor*)

lemma *less-floor-eq-one*: (*1 < floor x*) = (*2 <= x*)
by (*rule one-less-floor*)

lemma *floor-le-eq*: (*floor x <= a*) = (*x < real a + 1*)
unfolding *real-of-int-def* **by** (*rule floor-le-iff*)

lemma *floor-le-eq-number-of*:
(*floor x <= number-of n*) = (*x < number-of n + 1*)
by (*rule floor-le-number-of*)

lemma *floor-le-eq-zero*: (*floor x <= 0*) = (*x < 1*)
by (*rule floor-le-zero*)

lemma *floor-le-eq-one*: (*floor x <= 1*) = (*x < 2*)
by (*rule floor-le-one*)

lemma *floor-add* [*simp*]: *floor* (*x + real a*) = *floor x + a*

unfolding *real-of-int-def* **by** (*rule floor-add-of-int*)

lemma *floor-subtract* [*simp*]: *floor* (*x* − *real a*) = *floor x* − *a*
unfolding *real-of-int-def* **by** (*rule floor-diff-of-int*)

lemma *floor-subtract-number-of*: *floor* (*x* − *number-of n*) =
floor x − *number-of n*
by (*rule floor-diff-number-of*)

lemma *floor-subtract-one*: *floor* (*x* − *1*) = *floor x* − *1*
by (*rule floor-diff-one*)

lemma *ceiling-real-of-nat* [*simp*]: *ceiling* (*real* (*n*::*nat*)) = *int n*
unfolding *real-of-nat-def* **by** *simp*

lemma *ceiling-real-of-nat-zero*: *ceiling* (*real* (*0*::*nat*)) = *0*
by *auto*

lemma *ceiling-floor* [*simp*]: *ceiling* (*real* (*floor r*)) = *floor r*
unfolding *real-of-int-def* **by** *simp*

lemma *floor-ceiling* [*simp*]: *floor* (*real* (*ceiling r*)) = *ceiling r*
unfolding *real-of-int-def* **by** *simp*

lemma *real-of-int-ceiling-ge* [*simp*]: *r* ≤ *real* (*ceiling r*)
unfolding *real-of-int-def* **by** (*rule le-of-int-ceiling*)

lemma *ceiling-real-of-int* [*simp*]: *ceiling* (*real* (*n*::*int*)) = *n*
unfolding *real-of-int-def* **by** *simp*

lemma *real-of-int-ceiling-cancel* [*simp*]:
(*real* (*ceiling x*) = *x*) = (∃ *n*::*int*. *x* = *real n*)
using *ceiling-real-of-int* **by** *metis*

lemma *ceiling-eq*: [| *real n* < *x*; *x* < *real n* + *1* |] ==> *ceiling x* = *n* + *1*
unfolding *real-of-int-def* **using** *ceiling-unique* [*of n* + *1 x*] **by** *simp*

lemma *ceiling-eq2*: [| *real n* < *x*; *x* ≤ *real n* + *1* |] ==> *ceiling x* = *n* + *1*
unfolding *real-of-int-def* **using** *ceiling-unique* [*of n* + *1 x*] **by** *simp*

lemma *ceiling-eq3*: [| *real n* − *1* < *x*; *x* ≤ *real n* |] ==> *ceiling x* = *n*
unfolding *real-of-int-def* **using** *ceiling-unique* [*of n x*] **by** *simp*

lemma *ceiling-number-of-eq*:
ceiling (*number-of n* :: *real*) = (*number-of n*)
by (*rule ceiling-number-of*)

lemma *real-of-int-ceiling-diff-one-le* [*simp*]: *real* (*ceiling r*) − *1* ≤ *r*
unfolding *real-of-int-def* **using** *ceiling-correct* [*of r*] **by** *simp*

lemma *real-of-int-ceiling-le-add-one* [*simp*]: *real* (*ceiling r*) ≤ *r* + *1*
unfolding *real-of-int-def* **using** *ceiling-correct* [*of r*] **by** *simp*

lemma *ceiling-le*: *x* <= *real a* ==> *ceiling x* <= *a*
unfolding *real-of-int-def* **by** (*simp add*: *ceiling-le-iff*)

lemma *ceiling-le-real*: *ceiling x* <= *a* ==> *x* <= *real a*
unfolding *real-of-int-def* **by** (*simp add*: *ceiling-le-iff*)

lemma *ceiling-le-eq*: (*ceiling x* <= *a*) = (*x* <= *real a*)
unfolding *real-of-int-def* **by** (*rule ceiling-le-iff*)

lemma *ceiling-le-eq-number-of*:
(*ceiling x* <= *number-of n*) = (*x* <= *number-of n*)
by (*rule ceiling-le-number-of*)

lemma *ceiling-le-zero-eq*: (*ceiling x* <= *0*) = (*x* <= *0*)
by (*rule ceiling-le-zero*)

lemma *ceiling-le-eq-one*: (*ceiling x* <= *1*) = (*x* <= *1*)
by (*rule ceiling-le-one*)

lemma *less-ceiling-eq*: (*a* < *ceiling x*) = (*real a* < *x*)
unfolding *real-of-int-def* **by** (*rule less-ceiling-iff*)

lemma *less-ceiling-eq-number-of*:
(*number-of n* < *ceiling x*) = (*number-of n* < *x*)
by (*rule number-of-less-ceiling*)

lemma *less-ceiling-eq-zero*: (*0* < *ceiling x*) = (*0* < *x*)
by (*rule zero-less-ceiling*)

lemma *less-ceiling-eq-one*: (*1* < *ceiling x*) = (*1* < *x*)
by (*rule one-less-ceiling*)

lemma *ceiling-less-eq*: (*ceiling x* < *a*) = (*x* <= *real a* − *1*)
unfolding *real-of-int-def* **by** (*rule ceiling-less-iff*)

lemma *ceiling-less-eq-number-of*:
(*ceiling x* < *number-of n*) = (*x* <= *number-of n* − *1*)
by (*rule ceiling-less-number-of*)

lemma *ceiling-less-eq-zero*: (*ceiling x* < *0*) = (*x* <= *−1*)
by (*rule ceiling-less-zero*)

lemma *ceiling-less-eq-one*: (*ceiling x* < *1*) = (*x* <= *0*)
by (*rule ceiling-less-one*)

lemma *le-ceiling-eq*: $(a <= ceiling\ x) = (real\ a - 1 < x)$
unfolding *real-of-int-def* **by** (*rule le-ceiling-iff*)

lemma *le-ceiling-eq-number-of*:
$(number\text{-}of\ n <= ceiling\ x) = (number\text{-}of\ n - 1 < x)$
by (*rule number-of-le-ceiling*)

lemma *le-ceiling-eq-zero*: $(0 <= ceiling\ x) = (-1 < x)$
by (*rule zero-le-ceiling*)

lemma *le-ceiling-eq-one*: $(1 <= ceiling\ x) = (0 < x)$
by (*rule one-le-ceiling*)

lemma *ceiling-add* [*simp*]: $ceiling\ (x + real\ a) = ceiling\ x + a$
unfolding *real-of-int-def* **by** (*rule ceiling-add-of-int*)

lemma *ceiling-subtract* [*simp*]: $ceiling\ (x - real\ a) = ceiling\ x - a$
unfolding *real-of-int-def* **by** (*rule ceiling-diff-of-int*)

lemma *ceiling-subtract-number-of*: $ceiling\ (x - number\text{-}of\ n) =$
$ceiling\ x - number\text{-}of\ n$
by (*rule ceiling-diff-number-of*)

lemma *ceiling-subtract-one*: $ceiling\ (x - 1) = ceiling\ x - 1$
by (*rule ceiling-diff-one*)

53.5 Versions for the natural numbers

definition
$natfloor :: real => nat$ **where**
$natfloor\ x = nat(floor\ x)$

definition
$natceiling :: real => nat$ **where**
$natceiling\ x = nat(ceiling\ x)$

lemma *natfloor-zero* [*simp*]: $natfloor\ 0 = 0$
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-one* [*simp*]: $natfloor\ 1 = 1$
by (*unfold natfloor-def*, *simp*)

lemma *zero-le-natfloor* [*simp*]: $0 <= natfloor\ x$
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-number-of-eq* [*simp*]: $natfloor\ (number\text{-}of\ n) = number\text{-}of\ n$
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-real-of-nat* [*simp*]: $natfloor(real\ n) = n$

by (*unfold natfloor-def*, *simp*)

lemma *real-natfloor-le*: *0* <= *x* ==> *real*(*natfloor x*) <= *x*
by (*unfold natfloor-def*, *simp*)

lemma *natfloor-neg*: *x* <= *0* ==> *natfloor x* = *0*
apply (*unfold natfloor-def*)
apply (*subgoal-tac floor x* <= *floor 0*)
apply *simp*
apply (*erule floor-mono*)
done

lemma *natfloor-mono*: *x* <= *y* ==> *natfloor x* <= *natfloor y*
apply (*case-tac 0* <= *x*)
apply (*subst natfloor-def*)+
apply (*subst nat-le-eq-zle*)
apply *force*
apply (*erule floor-mono*)
apply (*subst natfloor-neg*)
apply *simp*
apply *simp*
done

lemma *le-natfloor*: *real x* <= *a* ==> *x* <= *natfloor a*
apply (*unfold natfloor-def*)
apply (*subst nat-int* [*THEN sym*])
apply (*subst nat-le-eq-zle*)
apply *simp*
apply (*rule le-floor*)
apply *simp*
done

lemma *le-natfloor-eq*: *0* <= *x* ==> (*a* <= *natfloor x*) = (*real a* <= *x*)
apply (*rule iffI*)
apply (*rule order-trans*)
prefer *2*
apply (*erule real-natfloor-le*)
apply (*subst real-of-nat-le-iff*)
apply *assumption*
apply (*erule le-natfloor*)
done

lemma *le-natfloor-eq-number-of* [*simp*]:
~ *neg*((*number-of n*)::*int*) ==> *0* <= *x* ==>
(*number-of n* <= *natfloor x*) = (*number-of n* <= *x*)
apply (*subst le-natfloor-eq*, *assumption*)
apply *simp*
done

lemma *le-natfloor-eq-one* [*simp*]: (*1 <= natfloor x*) = (*1 <= x*)
apply (*case-tac 0 <= x*)
apply (*subst le-natfloor-eq*, *assumption*, *simp*)
apply (*rule iffI*)
apply (*subgoal-tac natfloor x <= natfloor 0*)
apply *simp*
apply (*rule natfloor-mono*)
apply *simp*
apply *simp*
done

lemma *natfloor-eq*: *real n <= x ==> x < real n + 1 ==> natfloor x = n*
apply (*unfold natfloor-def*)
apply (*subst nat-int* [*THEN sym*])**back**
apply (*subst eq-nat-nat-iff*)
apply *simp*
apply *simp*
apply (*rule floor-eq2*)
apply *auto*
done

lemma *real-natfloor-add-one-gt*: *x < real*(*natfloor x*) + *1*
apply (*case-tac 0 <= x*)
apply (*unfold natfloor-def*)
apply *simp*
apply *simp-all*
done

lemma *real-natfloor-gt-diff-one*: *x - 1 < real*(*natfloor x*)
using *real-natfloor-add-one-gt* **by** (*simp add*: *algebra-simps*)

lemma *ge-natfloor-plus-one-imp-gt*: *natfloor z + 1 <= n ==> z < real n*
apply (*subgoal-tac z < real*(*natfloor z*) + *1*)
apply *arith*
apply (*rule real-natfloor-add-one-gt*)
done

lemma *natfloor-add* [*simp*]: *0 <= x ==> natfloor* (*x* + *real a*) = *natfloor x* + *a*
apply (*unfold natfloor-def*)
apply (*subgoal-tac real a = real* (*int a*))
apply (*erule ssubst*)
apply (*simp add*: *nat-add-distrib del*: *real-of-int-of-nat-eq*)
apply *simp*
done

lemma *natfloor-add-number-of* [*simp*]:
~*neg* ((*number-of n*)::*int*) ==> *0 <= x ==>*
natfloor (*x* + *number-of n*) = *natfloor x* + *number-of n*
apply (*subst natfloor-add* [*THEN sym*])

apply *simp-all*
done

lemma *natfloor-add-one*: *0 <= x ==> natfloor(x + 1) = natfloor x + 1*
apply (*subst natfloor-add* [*THEN sym*])
apply *assumption*
apply *simp*
done

lemma *natfloor-subtract* [*simp*]: *real a <= x ==>*
natfloor(x − real a) = natfloor x − a
apply (*unfold natfloor-def*)
apply (*subgoal-tac real a = real* (*int a*))
apply (*erule ssubst*)
apply (*simp del*: *real-of-int-of-nat-eq*)
apply *simp*
done

lemma *natceiling-zero* [*simp*]: *natceiling 0 = 0*
by (*unfold natceiling-def*, *simp*)

lemma *natceiling-one* [*simp*]: *natceiling 1 = 1*
by (*unfold natceiling-def*, *simp*)

lemma *zero-le-natceiling* [*simp*]: *0 <= natceiling x*
by (*unfold natceiling-def*, *simp*)

lemma *natceiling-number-of-eq* [*simp*]: *natceiling* (*number-of n*) = *number-of n*
by (*unfold natceiling-def*, *simp*)

lemma *natceiling-real-of-nat* [*simp*]: *natceiling*(*real n*) = *n*
by (*unfold natceiling-def*, *simp*)

lemma *real-natceiling-ge*: *x <= real*(*natceiling x*)
apply (*unfold natceiling-def*)
apply (*case-tac x < 0*)
apply *simp*
apply (*subst real-nat-eq-real*)
apply (*subgoal-tac ceiling 0 <= ceiling x*)
apply *simp*
apply (*rule ceiling-mono*)
apply *simp*
apply *simp*
done

lemma *natceiling-neg*: *x <= 0 ==> natceiling x = 0*
apply (*unfold natceiling-def*)
apply *simp*
done

lemma *natceiling-mono*: $x <= y ==> natceiling\ x <= natceiling\ y$
apply (*case-tac* $0 <= x$)
apply (*subst natceiling-def*)+
apply (*subst nat-le-eq-zle*)
apply (*rule disjI2*)
apply (*subgoal-tac real* $(0::int) <= real(ceiling\ y)$)
apply *simp*
apply (*rule order-trans*)
apply *simp*
apply (*erule order-trans*)
apply *simp*
apply (*erule ceiling-mono*)
apply (*subst natceiling-neg*)
apply *simp-all*
done

lemma *natceiling-le*: $x <= real\ a ==> natceiling\ x <= a$
apply (*unfold natceiling-def*)
apply (*case-tac* $x < 0$)
apply *simp*
apply (*subst nat-int* [*THEN sym*])**back**
apply (*subst nat-le-eq-zle*)
apply *simp*
apply (*rule ceiling-le*)
apply *simp*
done

lemma *natceiling-le-eq*: $0 <= x ==> (natceiling\ x <= a) = (x <= real\ a)$
apply (*rule iffI*)
apply (*rule order-trans*)
apply (*rule real-natceiling-ge*)
apply (*subst real-of-nat-le-iff*)
apply *assumption*
apply (*erule natceiling-le*)
done

lemma *natceiling-le-eq-number-of* [*simp*]:
$\sim neg((number\text{-}of\ n)::int) ==> 0 <= x ==>$
$(natceiling\ x <= number\text{-}of\ n) = (x <= number\text{-}of\ n)$
apply (*subst natceiling-le-eq*, *assumption*)
apply *simp*
done

lemma *natceiling-le-eq-one*: $(natceiling\ x <= 1) = (x <= 1)$
apply (*case-tac* $0 <= x$)
apply (*subst natceiling-le-eq*)
apply *assumption*
apply *simp*

apply (*subst natceiling-neg*)
apply *simp*
apply *simp*
done

lemma *natceiling-eq*: *real n < x ==> x <= real n + 1 ==> natceiling x = n + 1*
apply (*unfold natceiling-def*)
apply (*simplesubst nat-int* [*THEN sym*]) **back back**
apply (*subgoal-tac nat*(*int n*) + *1* = *nat*(*int n* + *1*))
apply (*erule ssubst*)
apply (*subst eq-nat-nat-iff*)
apply (*subgoal-tac ceiling 0 <= ceiling x*)
apply *simp*
apply (*rule ceiling-mono*)
apply *force*
apply *force*
apply (*rule ceiling-eq2*)
apply (*simp*, *simp*)
apply (*subst nat-add-distrib*)
apply *auto*
done

lemma *natceiling-add* [*simp*]: *0 <= x ==>*
natceiling (*x* + *real a*) = *natceiling x* + *a*
apply (*unfold natceiling-def*)
apply (*subgoal-tac real a = real* (*int a*))
apply (*erule ssubst*)
apply (*simp del*: *real-of-int-of-nat-eq*)
apply (*subst nat-add-distrib*)
apply (*subgoal-tac 0 = ceiling 0*)
apply (*erule ssubst*)
apply (*erule ceiling-mono*)
apply *simp-all*
done

lemma *natceiling-add-number-of* [*simp*]:
~ *neg* ((*number-of n*)::*int*) ==> *0 <= x ==>*
natceiling (*x* + *number-of n*) = *natceiling x* + *number-of n*
apply (*subst natceiling-add* [*THEN sym*])
apply *simp-all*
done

lemma *natceiling-add-one*: *0 <= x ==> natceiling*(*x* + *1*) = *natceiling x* + *1*
apply (*subst natceiling-add* [*THEN sym*])
apply *assumption*
apply *simp*
done

lemma *natceiling-subtract* [*simp*]: *real a <= x ==>*
natceiling(*x − real a*) = *natceiling x − a*
apply (*unfold natceiling-def*)
apply (*subgoal-tac real a = real* (*int a*))
apply (*erule ssubst*)
apply (*simp del*: *real-of-int-of-nat-eq*)
apply *simp*
done

lemma *natfloor-div-nat*: *1 <= x ==> y > 0 ==>*
natfloor (*x / real y*) = *natfloor x div y*
proof −
assume *1 <=* (*x*::*real*) **and** (*y*::*nat*) > *0*
have *natfloor x* = (*natfloor x*) *div y* ∗ *y* + (*natfloor x*) *mod y*
by *simp*
then have *a*: *real*(*natfloor x*) = *real* ((*natfloor x*) *div y*) ∗ *real y* +
real((*natfloor x*) *mod y*)
by (*simp only*: *real-of-nat-add* [*THEN sym*] *real-of-nat-mult* [*THEN sym*])
have *x* = *real*(*natfloor x*) + (*x − real*(*natfloor x*))
by *simp*
then have *x* = *real* ((*natfloor x*) *div y*) ∗ *real y* +
real((*natfloor x*) *mod y*) + (*x − real*(*natfloor x*))
by (*simp add*: *a*)
then have *x / real y* = ... / *real y*
by *simp*
also have ... = *real*((*natfloor x*) *div y*) + *real*((*natfloor x*) *mod y*) /
real y + (*x − real*(*natfloor x*)) / *real y*
by (*auto simp add*: *algebra-simps add-divide-distrib*
diff-divide-distrib prems)
finally have *natfloor* (*x / real y*) = *natfloor*(...) **by** *simp*
also have ... = *natfloor*(*real*((*natfloor x*) *mod y*) /
real y + (*x − real*(*natfloor x*)) / *real y* + *real*((*natfloor x*) *div y*))
by (*simp add*: *add-ac*)
also have ... = *natfloor*(*real*((*natfloor x*) *mod y*) /
real y + (*x − real*(*natfloor x*)) / *real y*) + (*natfloor x*) *div y*
apply (*rule natfloor-add*)
apply (*rule add-nonneg-nonneg*)
apply (*rule divide-nonneg-pos*)
apply *simp*
apply (*simp add*: *prems*)
apply (*rule divide-nonneg-pos*)
apply (*simp add*: *algebra-simps*)
apply (*rule real-natfloor-le*)
apply (*insert prems*, *auto*)
done
also have *natfloor*(*real*((*natfloor x*) *mod y*) /
real y + (*x − real*(*natfloor x*)) / *real y*) = *0*
apply (*rule natfloor-eq*)
apply *simp*

apply (*rule add-nonneg-nonneg*)
apply (*rule divide-nonneg-pos*)
apply *force*
apply (*force simp add*: *prems*)
apply (*rule divide-nonneg-pos*)
apply (*simp add*: *algebra-simps*)
apply (*rule real-natfloor-le*)
apply (*auto simp add*: *prems*)
apply (*insert prems*, *arith*)
apply (*simp add*: *add-divide-distrib* [*THEN sym*])
apply (*subgoal-tac real y = real y − 1 + 1*)
apply (*erule ssubst*)
apply (*rule add-le-less-mono*)
apply (*simp add*: *algebra-simps*)
apply (*subgoal-tac 1 + real*(*natfloor x mod y*) =
real(*natfloor x mod y + 1*))
apply (*erule ssubst*)
apply (*subst real-of-nat-le-iff*)
apply (*subgoal-tac natfloor x mod y < y*)
apply *arith*
apply (*rule mod-less-divisor*)
apply *auto*
using *real-natfloor-add-one-gt*
apply (*simp add*: *algebra-simps*)
done
finally show *?thesis* **by** *simp*
qed

end

54 RealPow: Natural powers theory

theory *RealPow*
imports *RealDef*
uses (*Tools/float-syntax.ML*)
begin

declare *abs-mult-self* [*simp*]

instantiation *real* :: *recpower*
begin

primrec *power-real* **where**
r ^ 0 = (*1*::*real*)
| *r ^ Suc n* = (*r*::*real*) ∗ *r ^ n*

instance proof
fix *z* :: *real*

fix n :: *nat*
show $z\hat{}0 = 1$ **by** *simp*
show $z\hat{}(Suc\ n) = z * (z\hat{}n)$ **by** *simp*
qed

declare *power-real.simps* [*simp del*]

end

lemma *two-realpow-ge-one* [*simp*]: $(1::real) \leq 2\ \hat{}\ n$
by *simp*

lemma *two-realpow-gt* [*simp*]: *real* $(n::nat) < 2\ \hat{}\ n$
apply (*induct n*)
apply (*auto simp add*: *real-of-nat-Suc*)
apply (*subst mult-2*)
apply (*rule add-less-le-mono*)
apply (*auto simp add*: *two-realpow-ge-one*)
done

lemma *realpow-Suc-le-self*: [| $0 \leq r$; $r \leq (1::real)$ |] ==> $r\ \hat{}\ Suc\ n \leq r$
by (*insert power-decreasing* [*of 1 Suc n r*], *simp*)

lemma *realpow-minus-mult* [*rule-format*]:
$0 < n$ −−> $(x::real)\ \hat{}\ (n - 1) * x = x\ \hat{}\ n$
unfolding *One-nat-def*
apply (*simp split add*: *nat-diff-split*)
done

lemma *realpow-two-mult-inverse* [*simp*]:
$r \neq 0$ ==> $r * inverse\ r\ \hat{}Suc\ (Suc\ 0) = inverse\ (r::real)$
by (*simp add*: *real-mult-assoc* [*symmetric*])

lemma *realpow-two-minus* [*simp*]: $(-x)\ \hat{}Suc\ (Suc\ 0) = (x::real)\ \hat{}Suc\ (Suc\ 0)$
by *simp*

lemma *realpow-two-diff*:
$(x::real)\ \hat{}Suc\ (Suc\ 0) - y\ \hat{}Suc\ (Suc\ 0) = (x - y) * (x + y)$
apply (*unfold real-diff-def*)
apply (*simp add*: *algebra-simps*)
done

lemma *realpow-two-disj*:
$((x::real)\ \hat{}Suc\ (Suc\ 0) = y\ \hat{}Suc\ (Suc\ 0)) = (x = y \mid x = -y)$
apply (*cut-tac* $x = x$ **and** $y = y$ **in** *realpow-two-diff*)
apply *auto*
done

lemma *realpow-real-of-nat*: *real* (*m*::*nat*) ^ *n* = *real* (*m* ^ *n*)
apply (*induct n*)
apply (*auto simp add*: *real-of-nat-one real-of-nat-mult*)
done

lemma *realpow-real-of-nat-two-pos* [*simp*] : *0* < *real* (*Suc* (*Suc 0*) ^ *n*)
apply (*induct n*)
apply (*auto simp add*: *real-of-nat-mult zero-less-mult-iff*)
done

lemma *realpow-increasing*:
[|(*0*::*real*) ≤ *x*; *0* ≤ *y*; *x* ^ *Suc n* ≤ *y* ^ *Suc n*|] ==> *x* ≤ *y*
by (*rule power-le-imp-le-base*)

54.1 Literal Arithmetic Involving Powers, Type *real*

lemma *real-of-int-power*: *real* (*x*::*int*) ^ *n* = *real* (*x* ^ *n*)
apply (*induct n*)
apply (*simp-all add*: *nat-mult-distrib*)
done
declare *real-of-int-power* [*symmetric*, *simp*]

lemma *power-real-number-of*:
(*number-of v* :: *real*) ^ *n* = *real* ((*number-of v* :: *int*) ^ *n*)
by (*simp only*: *real-number-of* [*symmetric*] *real-of-int-power*)

declare *power-real-number-of* [*of* - *number-of w*, *standard*, *simp*]

54.2 Properties of Squares

lemma *sum-squares-ge-zero*:
fixes *x y* :: *'a*::*ordered-ring-strict*
shows *0* ≤ *x* * *x* + *y* * *y*
by (*intro add-nonneg-nonneg zero-le-square*)

lemma *not-sum-squares-lt-zero*:
fixes *x y* :: *'a*::*ordered-ring-strict*
shows ¬ *x* * *x* + *y* * *y* < *0*
by (*simp add*: *linorder-not-less sum-squares-ge-zero*)

lemma *sum-nonneg-eq-zero-iff*:
fixes *x y* :: *'a*::*pordered-ab-group-add*
assumes *x*: *0* ≤ *x* **and** *y*: *0* ≤ *y*
shows (*x* + *y* = *0*) = (*x* = *0* ∧ *y* = *0*)
proof (*auto*)
from *y* **have** *x* + *0* ≤ *x* + *y* **by** (*rule add-left-mono*)
also assume *x* + *y* = *0*
finally have *x* ≤ *0* **by** *simp*
thus *x* = *0* **using** *x* **by** (*rule order-antisym*)

next
from x **have** $0 + y \leq x + y$ **by** (*rule add-right-mono*)
also assume $x + y = 0$
finally have $y \leq 0$ **by** *simp*
thus $y = 0$ **using** y **by** (*rule order-antisym*)
qed

lemma *sum-squares-eq-zero-iff*:
fixes $x\ y :: 'a$::*ordered-ring-strict*
shows $(x * x + y * y = 0) = (x = 0 \wedge y = 0)$
by (*simp add*: *sum-nonneg-eq-zero-iff*)

lemma *sum-squares-le-zero-iff*:
fixes $x\ y :: 'a$::*ordered-ring-strict*
shows $(x * x + y * y \leq 0) = (x = 0 \wedge y = 0)$
by (*simp add*: *order-le-less not-sum-squares-lt-zero sum-squares-eq-zero-iff*)

lemma *sum-squares-gt-zero-iff*:
fixes $x\ y :: 'a$::*ordered-ring-strict*
shows $(0 < x * x + y * y) = (x \neq 0 \vee y \neq 0)$
by (*simp add*: *order-less-le sum-squares-ge-zero sum-squares-eq-zero-iff*)

lemma *sum-power2-ge-zero*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $0 \leq x^2 + y^2$
unfolding *power2-eq-square* **by** (*rule sum-squares-ge-zero*)

lemma *not-sum-power2-lt-zero*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $\neg\ x^2 + y^2 < 0$
unfolding *power2-eq-square* **by** (*rule not-sum-squares-lt-zero*)

lemma *sum-power2-eq-zero-iff*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $(x^2 + y^2 = 0) = (x = 0 \wedge y = 0)$
unfolding *power2-eq-square* **by** (*rule sum-squares-eq-zero-iff*)

lemma *sum-power2-le-zero-iff*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $(x^2 + y^2 \leq 0) = (x = 0 \wedge y = 0)$
unfolding *power2-eq-square* **by** (*rule sum-squares-le-zero-iff*)

lemma *sum-power2-gt-zero-iff*:
fixes $x\ y :: 'a$::{*ordered-idom*,*recpower*}
shows $(0 < x^2 + y^2) = (x \neq 0 \vee y \neq 0)$
unfolding *power2-eq-square* **by** (*rule sum-squares-gt-zero-iff*)

54.3 Squares of Reals

lemma *real-two-squares-add-zero-iff* [*simp*]:
$(x * x + y * y = 0) = ((x::real) = 0 \wedge y = 0)$
by (*rule sum-squares-eq-zero-iff*)

lemma *real-sum-squares-cancel*: $x * x + y * y = 0 ==> x = (0::real)$
by *simp*

lemma *real-sum-squares-cancel2*: $x * x + y * y = 0 ==> y = (0::real)$
by *simp*

lemma *real-mult-self-sum-ge-zero*: $(0::real) \leq x*x + y*y$
by (*rule sum-squares-ge-zero*)

lemma *real-sum-squares-cancel-a*: $x * x = -(y * y) ==> x = (0::real)$ & $y=0$
by (*simp add*: *real-add-eq-0-iff* [*symmetric*])

lemma *real-squared-diff-one-factored*: $x*x - (1::real) = (x + 1)*(x - 1)$
by (*simp add*: *left-distrib right-diff-distrib*)

lemma *real-mult-is-one* [*simp*]: $(x*x = (1::real)) = (x = 1 \mid x = -1)$
apply *auto*
apply (*drule right-minus-eq* [*THEN iffD2*])
apply (*auto simp add*: *real-squared-diff-one-factored*)
done

lemma *real-sum-squares-not-zero*: $x \sim= 0 ==> x * x + y * y \sim= (0::real)$
by *simp*

lemma *real-sum-squares-not-zero2*: $y \sim= 0 ==> x * x + y * y \sim= (0::real)$
by *simp*

lemma *realpow-two-sum-zero-iff* [*simp*]:
$(x \hat{} 2 + y \hat{} 2 = (0::real)) = (x = 0 \& y = 0)$
by (*rule sum-power2-eq-zero-iff*)

lemma *realpow-two-le-add-order* [*simp*]: $(0::real) \leq u \hat{} 2 + v \hat{} 2$
by (*rule sum-power2-ge-zero*)

lemma *realpow-two-le-add-order2* [*simp*]: $(0::real) \leq u \hat{} 2 + v \hat{} 2 + w \hat{} 2$
by (*intro add-nonneg-nonneg zero-le-power2*)

lemma *real-sum-square-gt-zero*: $x \sim= 0 ==> (0::real) < x * x + y * y$
by (*simp add*: *sum-squares-gt-zero-iff*)

lemma *real-sum-square-gt-zero2*: $y \sim= 0 ==> (0::real) < x * x + y * y$
by (*simp add*: *sum-squares-gt-zero-iff*)

lemma *real-minus-mult-self-le* [*simp*]: $-(u * u) \leq (x * (x::real))$

by (*rule-tac j = 0* **in** *real-le-trans*, *auto*)

lemma *realpow-square-minus-le* [*simp*]: $-(u \hat{} 2) \leq (x::real) \hat{} 2$
by (*auto simp add*: *power2-eq-square*)

lemma *real-sq-order*:
fixes *x*::*real*
assumes *xgt0*: $0 \leq x$ **and** *ygt0*: $0 \leq y$ **and** *sq*: $x\hat{}2 \leq y\hat{}2$
shows $x \leq y$
proof −
from *sq* **have** $x \hat{} Suc\ (Suc\ 0) \leq y \hat{} Suc\ (Suc\ 0)$
by (*simp only*: *numeral-2-eq-2*)
thus $x \leq y$ **using** *ygt0*
by (*rule power-le-imp-le-base*)
qed

54.4 Various Other Theorems

lemma *real-le-add-half-cancel*: $(x + y/2 \leq (y::real)) = (x \leq y\ /2)$
by *auto*

lemma *real-minus-half-eq* [*simp*]: $(x::real) - x/2 = x/2$
by *auto*

lemma *real-mult-inverse-cancel*:
$[|(0::real) < x;\ 0 < x1;\ x1 * y < x * u\ |]$
$==>$ *inverse x* $*$ *y* $<$ *inverse x1* $*$ *u*
apply (*rule-tac c*=*x* **in** *mult-less-imp-less-left*)
apply (*auto simp add*: *real-mult-assoc* [*symmetric*])
apply (*simp* (*no-asm*) *add*: *mult-ac*)
apply (*rule-tac c*=*x1* **in** *mult-less-imp-less-right*)
apply (*auto simp add*: *mult-ac*)
done

lemma *real-mult-inverse-cancel2*:
$[|(0::real) < x;0 < x1;\ x1 * y < x * u\ |] ==> y *$ *inverse x* $< u *$ *inverse x1*
apply (*auto dest*: *real-mult-inverse-cancel simp add*: *mult-ac*)
done

lemma *inverse-real-of-nat-gt-zero* [*simp*]: $0 <$ *inverse* (*real* (*Suc n*))
by *simp*

lemma *inverse-real-of-nat-ge-zero* [*simp*]: $0 \leq$ *inverse* (*real* (*Suc n*))
by *simp*

lemma *realpow-num-eq-if*: $(m::real) \hat{} n =$ (*if n=0 then 1 else m* $*$ *m* $\hat{}$ $(n - 1)$)
by (*case-tac n*, *auto*)

54.5 Float syntax

syntax -*Float* :: *float-const* ⇒ 'a (-)

use *Tools/float-syntax.ML*
setup *FloatSyntax.setup*

Test:

lemma $123.456 = -111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::real)$
by *simp*

end

55 RealVector: Vector Spaces and Algebras over the Reals

theory *RealVector*
imports *RealPow*
begin

55.1 Locale for additive functions

locale *additive* =
  fixes f :: 'a::*ab-group-add* ⇒ 'b::*ab-group-add*
  assumes *add*: $f\ (x + y) = f\ x + f\ y$
begin

lemma *zero*: $f\ 0 = 0$
proof −
  have $f\ 0 = f\ (0 + 0)$ **by** *simp*
  also have $\ldots = f\ 0 + f\ 0$ **by** (*rule add*)
  finally show $f\ 0 = 0$ **by** *simp*
qed

lemma *minus*: $f\ (-\ x) = -\ f\ x$
proof −
  have $f\ (-\ x) + f\ x = f\ (-\ x + x)$ **by** (*rule add* [*symmetric*])
  also have $\ldots = -\ f\ x + f\ x$ **by** (*simp add*: *zero*)
  finally show $f\ (-\ x) = -\ f\ x$ **by** (*rule add-right-imp-eq*)
qed

lemma *diff*: $f\ (x - y) = f\ x - f\ y$
by (*simp add*: *diff-def add minus*)

lemma *setsum*: $f\ (setsum\ g\ A) = (\sum x{\in}A.\ f\ (g\ x))$
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply (*simp add*: *zero*)

apply (*simp add*: *add*)
apply (*simp add*: *zero*)
done

end

55.2 Vector spaces

locale *vector-space* =
  fixes *scale* :: *'a*::*field* ⇒ *'b*::*ab-group-add* ⇒ *'b*
  assumes *scale-right-distrib* [*algebra-simps*]:
    scale a (*x* + *y*) = *scale a x* + *scale a y*
  and *scale-left-distrib* [*algebra-simps*]:
    scale (*a* + *b*) *x* = *scale a x* + *scale b x*
  and *scale-scale* [*simp*]: *scale a* (*scale b x*) = *scale* (*a* ∗ *b*) *x*
  and *scale-one* [*simp*]: *scale 1 x* = *x*
begin

lemma *scale-left-commute*:
  scale a (*scale b x*) = *scale b* (*scale a x*)
by (*simp add*: *mult-commute*)

lemma *scale-zero-left* [*simp*]: *scale 0 x* = *0*
  and *scale-minus-left* [*simp*]: *scale* (− *a*) *x* = − (*scale a x*)
  and *scale-left-diff-distrib* [*algebra-simps*]:
        scale (*a* − *b*) *x* = *scale a x* − *scale b x*
proof −
  interpret *s*: *additive* λ*a*. *scale a x*
    proof qed (*rule scale-left-distrib*)
  show *scale 0 x* = *0* **by** (*rule s.zero*)
  show *scale* (− *a*) *x* = − (*scale a x*) **by** (*rule s.minus*)
  show *scale* (*a* − *b*) *x* = *scale a x* − *scale b x* **by** (*rule s.diff*)
qed

lemma *scale-zero-right* [*simp*]: *scale a 0* = *0*
  and *scale-minus-right* [*simp*]: *scale a* (− *x*) = − (*scale a x*)
  and *scale-right-diff-distrib* [*algebra-simps*]:
        scale a (*x* − *y*) = *scale a x* − *scale a y*
proof −
  interpret *s*: *additive* λ*x*. *scale a x*
    proof qed (*rule scale-right-distrib*)
  show *scale a 0* = *0* **by** (*rule s.zero*)
  show *scale a* (− *x*) = − (*scale a x*) **by** (*rule s.minus*)
  show *scale a* (*x* − *y*) = *scale a x* − *scale a y* **by** (*rule s.diff*)
qed

lemma *scale-eq-0-iff* [*simp*]:
  scale a x = *0* ⟷ *a* = *0* ∨ *x* = *0*
proof *cases*

assume $a = 0$ **thus** *?thesis* **by** *simp*
next
assume *anz* [*simp*]: $a \neq 0$
{ assume *scale a x* $= 0$
hence *scale* (*inverse a*) (*scale a x*) $= 0$ **by** *simp*
hence $x = 0$ **by** *simp* **}**
thus *?thesis* **by** *force*
qed

lemma *scale-left-imp-eq*:
$[\![a \neq 0; \mathit{scale}\ a\ x = \mathit{scale}\ a\ y]\!] \Longrightarrow x = y$
proof −
assume *nonzero*: $a \neq 0$
assume *scale a x* $=$ *scale a y*
hence *scale a* $(x - y) = 0$
by (*simp add*: *scale-right-diff-distrib*)
hence $x - y = 0$ **by** (*simp add*: *nonzero*)
thus $x = y$ **by** (*simp only*: *right-minus-eq*)
qed

lemma *scale-right-imp-eq*:
$[\![x \neq 0; \mathit{scale}\ a\ x = \mathit{scale}\ b\ x]\!] \Longrightarrow a = b$
proof −
assume *nonzero*: $x \neq 0$
assume *scale a x* $=$ *scale b x*
hence *scale* $(a - b)\ x = 0$
by (*simp add*: *scale-left-diff-distrib*)
hence $a - b = 0$ **by** (*simp add*: *nonzero*)
thus $a = b$ **by** (*simp only*: *right-minus-eq*)
qed

lemma *scale-cancel-left*:
scale a x $=$ *scale a y* $\longleftrightarrow x = y \vee a = 0$
by (*auto intro*: *scale-left-imp-eq*)

lemma *scale-cancel-right*:
scale a x $=$ *scale b x* $\longleftrightarrow a = b \vee x = 0$
by (*auto intro*: *scale-right-imp-eq*)

end

55.3 Real vector spaces

class *scaleR* =
fixes *scaleR* :: *real* $\Rightarrow$ *'a* $\Rightarrow$ *'a* (**infixr** $*_R$ *75*)
begin

abbreviation
divideR :: *'a* $\Rightarrow$ *real* $\Rightarrow$ *'a* (**infixl** $'/_R$ *70*)

where
  $x /_R r == scaleR\ (inverse\ r)\ x$

end

class *real-vector* = *scaleR* + *ab-group-add* +
  assumes *scaleR-right-distrib*: $scaleR\ a\ (x + y) = scaleR\ a\ x + scaleR\ a\ y$
  and *scaleR-left-distrib*: $scaleR\ (a + b)\ x = scaleR\ a\ x + scaleR\ b\ x$
  and *scaleR-scaleR*: $scaleR\ a\ (scaleR\ b\ x) = scaleR\ (a * b)\ x$
  and *scaleR-one*: $scaleR\ 1\ x = x$

interpretation *real-vector*:
  vector-space scaleR :: *real* ⇒ *'a* ⇒ *'a*::*real-vector*
apply *unfold-locales*
apply (*rule scaleR-right-distrib*)
apply (*rule scaleR-left-distrib*)
apply (*rule scaleR-scaleR*)
apply (*rule scaleR-one*)
done

Recover original theorem names

lemmas *scaleR-left-commute* = *real-vector.scale-left-commute*
lemmas *scaleR-zero-left* = *real-vector.scale-zero-left*
lemmas *scaleR-minus-left* = *real-vector.scale-minus-left*
lemmas *scaleR-left-diff-distrib* = *real-vector.scale-left-diff-distrib*
lemmas *scaleR-zero-right* = *real-vector.scale-zero-right*
lemmas *scaleR-minus-right* = *real-vector.scale-minus-right*
lemmas *scaleR-right-diff-distrib* = *real-vector.scale-right-diff-distrib*
lemmas *scaleR-eq-0-iff* = *real-vector.scale-eq-0-iff*
lemmas *scaleR-left-imp-eq* = *real-vector.scale-left-imp-eq*
lemmas *scaleR-right-imp-eq* = *real-vector.scale-right-imp-eq*
lemmas *scaleR-cancel-left* = *real-vector.scale-cancel-left*
lemmas *scaleR-cancel-right* = *real-vector.scale-cancel-right*

class *real-algebra* = *real-vector* + *ring* +
  assumes *mult-scaleR-left* [*simp*]: $scaleR\ a\ x * y = scaleR\ a\ (x * y)$
  and *mult-scaleR-right* [*simp*]: $x * scaleR\ a\ y = scaleR\ a\ (x * y)$

class *real-algebra-1* = *real-algebra* + *ring-1*

class *real-div-algebra* = *real-algebra-1* + *division-ring*

class *real-field* = *real-div-algebra* + *field*

instantiation *real* :: *real-field*
begin

definition
  real-scaleR-def [*simp*]: $scaleR\ a\ x = a * x$

instance proof
qed (*simp-all add*: *algebra-simps*)

end

interpretation *scaleR-left*: *additive* (λ*a*. *scaleR a x*::′*a*::*real-vector*)
proof qed (*rule scaleR-left-distrib*)

interpretation *scaleR-right*: *additive* (λ*x*. *scaleR a x*::′*a*::*real-vector*)
proof qed (*rule scaleR-right-distrib*)

lemma *nonzero-inverse-scaleR-distrib*:
fixes *x* :: ′*a*::*real-div-algebra* **shows**
⟦*a* ≠ *0*; *x* ≠ *0*⟧ ⟹ *inverse* (*scaleR a x*) = *scaleR* (*inverse a*) (*inverse x*)
by (*rule inverse-unique*, *simp*)

lemma *inverse-scaleR-distrib*:
fixes *x* :: ′*a*::{*real-div-algebra*,*division-by-zero*}
shows *inverse* (*scaleR a x*) = *scaleR* (*inverse a*) (*inverse x*)
apply (*case-tac a* = *0*, *simp*)
apply (*case-tac x* = *0*, *simp*)
apply (*erule* (*1*) *nonzero-inverse-scaleR-distrib*)
done

55.4 Embedding of the Reals into any *real-algebra-1*: *of-real*

definition
of-real :: *real* ⇒ ′*a*::*real-algebra-1* **where**
of-real r = *scaleR r 1*

lemma *scaleR-conv-of-real*: *scaleR r x* = *of-real r* ∗ *x*
by (*simp add*: *of-real-def*)

lemma *of-real-0* [*simp*]: *of-real 0* = *0*
by (*simp add*: *of-real-def*)

lemma *of-real-1* [*simp*]: *of-real 1* = *1*
by (*simp add*: *of-real-def*)

lemma *of-real-add* [*simp*]: *of-real* (*x* + *y*) = *of-real x* + *of-real y*
by (*simp add*: *of-real-def scaleR-left-distrib*)

lemma *of-real-minus* [*simp*]: *of-real* (− *x*) = − *of-real x*
by (*simp add*: *of-real-def*)

lemma *of-real-diff* [*simp*]: *of-real* (*x* − *y*) = *of-real x* − *of-real y*
by (*simp add*: *of-real-def scaleR-left-diff-distrib*)

lemma *of-real-mult* [*simp*]: *of-real* (*x* ∗ *y*) = *of-real x* ∗ *of-real y*
by (*simp add*: *of-real-def mult-commute*)

lemma *nonzero-of-real-inverse*:
x ≠ *0* ⟹ *of-real* (*inverse x*) =
inverse (*of-real x* :: *'a*::*real-div-algebra*)
by (*simp add*: *of-real-def nonzero-inverse-scaleR-distrib*)

lemma *of-real-inverse* [*simp*]:
of-real (*inverse x*) =
inverse (*of-real x* :: *'a*::{*real-div-algebra*,*division-by-zero*})
by (*simp add*: *of-real-def inverse-scaleR-distrib*)

lemma *nonzero-of-real-divide*:
y ≠ *0* ⟹ *of-real* (*x* / *y*) =
(*of-real x* / *of-real y* :: *'a*::*real-field*)
by (*simp add*: *divide-inverse nonzero-of-real-inverse*)

lemma *of-real-divide* [*simp*]:
of-real (*x* / *y*) =
(*of-real x* / *of-real y* :: *'a*::{*real-field*,*division-by-zero*})
by (*simp add*: *divide-inverse*)

lemma *of-real-power* [*simp*]:
of-real (*x* ^ *n*) = (*of-real x* :: *'a*::{*real-algebra-1*,*recpower*}) ^ *n*
by (*induct n*) *simp-all*

lemma *of-real-eq-iff* [*simp*]: (*of-real x* = *of-real y*) = (*x* = *y*)
by (*simp add*: *of-real-def scaleR-cancel-right*)

lemmas *of-real-eq-0-iff* [*simp*] = *of-real-eq-iff* [*of - 0*, *simplified*]

lemma *of-real-eq-id* [*simp*]: *of-real* = (*id* :: *real* ⇒ *real*)
proof
fix *r*
show *of-real r* = *id r*
by (*simp add*: *of-real-def*)
qed

Collapse nested embeddings

lemma *of-real-of-nat-eq* [*simp*]: *of-real* (*of-nat n*) = *of-nat n*
by (*induct n*) *auto*

lemma *of-real-of-int-eq* [*simp*]: *of-real* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*, *simp*)

lemma *of-real-number-of-eq*:
of-real (*number-of w*) = (*number-of w* :: *'a*::{*number-ring*,*real-algebra-1*})
by (*simp add*: *number-of-eq*)

Every real algebra has characteristic zero

instance *real-algebra-1* < *ring-char-0*
proof
  fix *m n* :: *nat*
  have (*of-real* (*of-nat m*) = (*of-real* (*of-nat n*)::'*a*)) = (*m* = *n*)
    by (*simp only*: *of-real-eq-iff of-nat-eq-iff*)
  thus (*of-nat m* = (*of-nat n*::'*a*)) = (*m* = *n*)
    by (*simp only*: *of-real-of-nat-eq*)
qed

instance *real-field* < *field-char-0* **..**

55.5 The Set of Real Numbers

definition
  Reals :: '*a*::*real-algebra-1 set* **where**
  [*code del*]: *Reals* = *range of-real*

notation (*xsymbols*)
  Reals (ℝ)

lemma *Reals-of-real* [*simp*]: *of-real r* ∈ *Reals*
by (*simp add*: *Reals-def*)

lemma *Reals-of-int* [*simp*]: *of-int z* ∈ *Reals*
by (*subst of-real-of-int-eq* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-of-nat* [*simp*]: *of-nat n* ∈ *Reals*
by (*subst of-real-of-nat-eq* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-number-of* [*simp*]:
  (*number-of w*::'*a*::{*number-ring*,*real-algebra-1*}) ∈ *Reals*
by (*subst of-real-number-of-eq* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-0* [*simp*]: *0* ∈ *Reals*
apply (*unfold Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-0* [*symmetric*])
done

lemma *Reals-1* [*simp*]: *1* ∈ *Reals*
apply (*unfold Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-1* [*symmetric*])
done

lemma *Reals-add* [*simp*]: ⟦*a* ∈ *Reals*; *b* ∈ *Reals*⟧ ⟹ *a* + *b* ∈ *Reals*
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)

apply (*rule of-real-add* [*symmetric*])
done

lemma *Reals-minus* [*simp*]: $a \in Reals \Longrightarrow - a \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-minus* [*symmetric*])
done

lemma *Reals-diff* [*simp*]: $[\![a \in Reals; b \in Reals]\!] \Longrightarrow a - b \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-diff* [*symmetric*])
done

lemma *Reals-mult* [*simp*]: $[\![a \in Reals; b \in Reals]\!] \Longrightarrow a * b \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-mult* [*symmetric*])
done

lemma *nonzero-Reals-inverse*:
fixes $a :: 'a$::*real-div-algebra*
shows $[\![a \in Reals; a \neq 0]\!] \Longrightarrow inverse\ a \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*erule nonzero-of-real-inverse* [*symmetric*])
done

lemma *Reals-inverse* [*simp*]:
fixes $a :: 'a$::{*real-div-algebra*,*division-by-zero*}
shows $a \in Reals \Longrightarrow inverse\ a \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-inverse* [*symmetric*])
done

lemma *nonzero-Reals-divide*:
fixes $a\ b :: 'a$::*real-field*
shows $[\![a \in Reals; b \in Reals; b \neq 0]\!] \Longrightarrow a\ /\ b \in Reals$
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*erule nonzero-of-real-divide* [*symmetric*])
done

lemma *Reals-divide* [*simp*]:
fixes $a\ b :: 'a$::{*real-field*,*division-by-zero*}
shows $[\![a \in Reals; b \in Reals]\!] \Longrightarrow a\ /\ b \in Reals$
apply (*auto simp add*: *Reals-def*)

apply (*rule range-eqI*)
apply (*rule of-real-divide* [*symmetric*])
done

lemma *Reals-power* [*simp*]:
  fixes *a* :: *'a*::{*real-algebra-1*,*recpower*}
  shows *a* ∈ *Reals* ⟹ *a* ^ *n* ∈ *Reals*
apply (*auto simp add*: *Reals-def*)
apply (*rule range-eqI*)
apply (*rule of-real-power* [*symmetric*])
done

lemma *Reals-cases* [*cases set*: *Reals*]:
  assumes *q* ∈ ℝ
  obtains (*of-real*) *r* **where** *q* = *of-real r*
  unfolding *Reals-def*
proof −
  from ‹*q* ∈ ℝ› **have** *q* ∈ *range of-real* **unfolding** *Reals-def* **.**
  then obtain *r* **where** *q* = *of-real r* **..**
  then show *thesis* **..**
qed

lemma *Reals-induct* [*case-names of-real*, *induct set*: *Reals*]:
  q ∈ ℝ ⟹ (⋀*r*. *P* (*of-real r*)) ⟹ *P q*
  by (*rule Reals-cases*) *auto*

55.6 Real normed vector spaces

class *norm* =
  fixes *norm* :: *'a* ⇒ *real*

class *sgn-div-norm* = *scaleR* + *norm* + *sgn* +
  assumes *sgn-div-norm*: *sgn x* = *x* /$_R$ *norm x*

class *real-normed-vector* = *real-vector* + *sgn-div-norm* +
  assumes *norm-ge-zero* [*simp*]: *0* ≤ *norm x*
  and *norm-eq-zero* [*simp*]: *norm x* = *0* ⟷ *x* = *0*
  and *norm-triangle-ineq*: *norm* (*x* + *y*) ≤ *norm x* + *norm y*
  and *norm-scaleR*: *norm* (*scaleR a x*) = |*a*| ∗ *norm x*

class *real-normed-algebra* = *real-algebra* + *real-normed-vector* +
  assumes *norm-mult-ineq*: *norm* (*x* ∗ *y*) ≤ *norm x* ∗ *norm y*

class *real-normed-algebra-1* = *real-algebra-1* + *real-normed-algebra* +
  assumes *norm-one* [*simp*]: *norm 1* = *1*

class *real-normed-div-algebra* = *real-div-algebra* + *real-normed-vector* +
  assumes *norm-mult*: *norm* (*x* ∗ *y*) = *norm x* ∗ *norm y*

class *real-normed-field* = *real-field* + *real-normed-div-algebra*

instance *real-normed-div-algebra* < *real-normed-algebra-1*
proof
  fix *x y* :: *'a*
  show *norm* (*x* * *y*) ≤ *norm x* * *norm y*
    by (*simp add*: *norm-mult*)
next
  have *norm* (*1* * *1*::*'a*) = *norm* (*1*::*'a*) * *norm* (*1*::*'a*)
    by (*rule norm-mult*)
  thus *norm* (*1*::*'a*) = *1* **by** *simp*
qed

instantiation *real* :: *real-normed-field*
begin

definition
  real-norm-def [*simp*]: *norm r* = |*r*|

instance
apply (*intro-classes*, *unfold real-norm-def real-scaleR-def*)
apply (*simp add*: *real-sgn-def*)
apply (*rule abs-ge-zero*)
apply (*rule abs-eq-0*)
apply (*rule abs-triangle-ineq*)
apply (*rule abs-mult*)
apply (*rule abs-mult*)
done

end

lemma *norm-zero* [*simp*]: *norm* (*0*::*'a*::*real-normed-vector*) = *0*
by *simp*

lemma *zero-less-norm-iff* [*simp*]:
  fixes *x* :: *'a*::*real-normed-vector*
  shows (*0* < *norm x*) = (*x* ≠ *0*)
by (*simp add*: *order-less-le*)

lemma *norm-not-less-zero* [*simp*]:
  fixes *x* :: *'a*::*real-normed-vector*
  shows ¬ *norm x* < *0*
by (*simp add*: *linorder-not-less*)

lemma *norm-le-zero-iff* [*simp*]:
  fixes *x* :: *'a*::*real-normed-vector*
  shows (*norm x* ≤ *0*) = (*x* = *0*)
by (*simp add*: *order-le-less*)

lemma *norm-minus-cancel* [*simp*]:
fixes *x* :: *'a*::*real-normed-vector*
shows *norm* (− *x*) = *norm x*
proof −
have *norm* (− *x*) = *norm* (*scaleR* (− *1*) *x*)
by (*simp only*: *scaleR-minus-left scaleR-one*)
also have ... = |− *1*| ∗ *norm x*
by (*rule norm-scaleR*)
finally show *?thesis* **by** *simp*
qed

lemma *norm-minus-commute*:
fixes *a b* :: *'a*::*real-normed-vector*
shows *norm* (*a* − *b*) = *norm* (*b* − *a*)
proof −
have *norm* (− (*b* − *a*)) = *norm* (*b* − *a*)
by (*rule norm-minus-cancel*)
thus *?thesis* **by** *simp*
qed

lemma *norm-triangle-ineq2*:
fixes *a b* :: *'a*::*real-normed-vector*
shows *norm a* − *norm b* ≤ *norm* (*a* − *b*)
proof −
have *norm* (*a* − *b* + *b*) ≤ *norm* (*a* − *b*) + *norm b*
by (*rule norm-triangle-ineq*)
thus *?thesis* **by** *simp*
qed

lemma *norm-triangle-ineq3*:
fixes *a b* :: *'a*::*real-normed-vector*
shows |*norm a* − *norm b*| ≤ *norm* (*a* − *b*)
apply (*subst abs-le-iff*)
apply *auto*
apply (*rule norm-triangle-ineq2*)
apply (*subst norm-minus-commute*)
apply (*rule norm-triangle-ineq2*)
done

lemma *norm-triangle-ineq4*:
fixes *a b* :: *'a*::*real-normed-vector*
shows *norm* (*a* − *b*) ≤ *norm a* + *norm b*
proof −
have *norm* (*a* + − *b*) ≤ *norm a* + *norm* (− *b*)
by (*rule norm-triangle-ineq*)
thus *?thesis*
by (*simp only*: *diff-minus norm-minus-cancel*)
qed

lemma *norm-diff-ineq*:
fixes *a b* :: *'a*::*real-normed-vector*
shows *norm a − norm b ≤ norm* (*a* + *b*)
proof −
have *norm a − norm* (− *b*) ≤ *norm* (*a* − − *b*)
by (*rule norm-triangle-ineq2*)
thus *?thesis* **by** *simp*
qed

lemma *norm-diff-triangle-ineq*:
fixes *a b c d* :: *'a*::*real-normed-vector*
shows *norm* ((*a* + *b*) − (*c* + *d*)) ≤ *norm* (*a* − *c*) + *norm* (*b* − *d*)
proof −
have *norm* ((*a* + *b*) − (*c* + *d*)) = *norm* ((*a* − *c*) + (*b* − *d*))
by (*simp add*: *diff-minus add-ac*)
also have ... ≤ *norm* (*a* − *c*) + *norm* (*b* − *d*)
by (*rule norm-triangle-ineq*)
finally show *?thesis* **.**
qed

lemma *abs-norm-cancel* [*simp*]:
fixes *a* :: *'a*::*real-normed-vector*
shows |*norm a*| = *norm a*
by (*rule abs-of-nonneg* [*OF norm-ge-zero*])

lemma *norm-add-less*:
fixes *x y* :: *'a*::*real-normed-vector*
shows ⟦*norm x* < *r*; *norm y* < *s*⟧ ⟹ *norm* (*x* + *y*) < *r* + *s*
by (*rule order-le-less-trans* [*OF norm-triangle-ineq add-strict-mono*])

lemma *norm-mult-less*:
fixes *x y* :: *'a*::*real-normed-algebra*
shows ⟦*norm x* < *r*; *norm y* < *s*⟧ ⟹ *norm* (*x* ∗ *y*) < *r* ∗ *s*
apply (*rule order-le-less-trans* [*OF norm-mult-ineq*])
apply (*simp add*: *mult-strict-mono'*)
done

lemma *norm-of-real* [*simp*]:
norm (*of-real r* :: *'a*::*real-normed-algebra-1*) = |*r*|
unfolding *of-real-def* **by** (*simp add*: *norm-scaleR*)

lemma *norm-number-of* [*simp*]:
norm (*number-of w*::*'a*::{*number-ring*,*real-normed-algebra-1*})
= |*number-of w*|
by (*subst of-real-number-of-eq* [*symmetric*], *rule norm-of-real*)

lemma *norm-of-int* [*simp*]:
norm (*of-int z*::*'a*::*real-normed-algebra-1*) = |*of-int z*|
by (*subst of-real-of-int-eq* [*symmetric*], *rule norm-of-real*)

lemma *norm-of-nat* [*simp*]:
norm (*of-nat n*::*'a*::*real-normed-algebra-1*) = *of-nat n*
apply (*subst of-real-of-nat-eq* [*symmetric*])
apply (*subst norm-of-real*, *simp*)
done

lemma *nonzero-norm-inverse*:
fixes *a* :: *'a*::*real-normed-div-algebra*
shows *a* ≠ *0* ⟹ *norm* (*inverse a*) = *inverse* (*norm a*)
apply (*rule inverse-unique* [*symmetric*])
apply (*simp add*: *norm-mult* [*symmetric*])
done

lemma *norm-inverse*:
fixes *a* :: *'a*::{*real-normed-div-algebra*,*division-by-zero*}
shows *norm* (*inverse a*) = *inverse* (*norm a*)
apply (*case-tac a* = *0*, *simp*)
apply (*erule nonzero-norm-inverse*)
done

lemma *nonzero-norm-divide*:
fixes *a b* :: *'a*::*real-normed-field*
shows *b* ≠ *0* ⟹ *norm* (*a* / *b*) = *norm a* / *norm b*
by (*simp add*: *divide-inverse norm-mult nonzero-norm-inverse*)

lemma *norm-divide*:
fixes *a b* :: *'a*::{*real-normed-field*,*division-by-zero*}
shows *norm* (*a* / *b*) = *norm a* / *norm b*
by (*simp add*: *divide-inverse norm-mult norm-inverse*)

lemma *norm-power-ineq*:
fixes *x* :: *'a*::{*real-normed-algebra-1*,*recpower*}
shows *norm* (*x* ^ *n*) ≤ *norm x* ^ *n*
proof (*induct n*)
case *0* **show** *norm* (*x* ^ *0*) ≤ *norm x* ^ *0* **by** *simp*
next
case (*Suc n*)
have *norm* (*x* ∗ *x* ^ *n*) ≤ *norm x* ∗ *norm* (*x* ^ *n*)
by (*rule norm-mult-ineq*)
also from *Suc* **have** ... ≤ *norm x* ∗ *norm x* ^ *n*
using *norm-ge-zero* **by** (*rule mult-left-mono*)
finally show *norm* (*x* ^ *Suc n*) ≤ *norm x* ^ *Suc n*
by *simp*
qed

lemma *norm-power*:
fixes *x* :: *'a*::{*real-normed-div-algebra*,*recpower*}
shows *norm* (*x* ^ *n*) = *norm x* ^ *n*

by (*induct n*) (*simp-all add*: *norm-mult*)

55.7 Sign function

lemma *norm-sgn*:
  norm (*sgn*(*x*::'*a*::*real-normed-vector*)) = (*if x* = *0 then 0 else 1*)
by (*simp add*: *sgn-div-norm norm-scaleR*)

lemma *sgn-zero* [*simp*]: *sgn*(*0*::'*a*::*real-normed-vector*) = *0*
by (*simp add*: *sgn-div-norm*)

lemma *sgn-zero-iff*: (*sgn*(*x*::'*a*::*real-normed-vector*) = *0*) = (*x* = *0*)
by (*simp add*: *sgn-div-norm*)

lemma *sgn-minus*: *sgn* (− *x*) = − *sgn*(*x*::'*a*::*real-normed-vector*)
by (*simp add*: *sgn-div-norm*)

lemma *sgn-scaleR*:
  sgn (*scaleR r x*) = *scaleR* (*sgn r*) (*sgn*(*x*::'*a*::*real-normed-vector*))
by (*simp add*: *sgn-div-norm norm-scaleR mult-ac*)

lemma *sgn-one* [*simp*]: *sgn* (*1*::'*a*::*real-normed-algebra-1*) = *1*
by (*simp add*: *sgn-div-norm*)

lemma *sgn-of-real*:
  sgn (*of-real r*::'*a*::*real-normed-algebra-1*) = *of-real* (*sgn r*)
unfolding *of-real-def* **by** (*simp only*: *sgn-scaleR sgn-one*)

lemma *sgn-mult*:
  fixes *x y* :: '*a*::*real-normed-div-algebra*
  shows *sgn* (*x* ∗ *y*) = *sgn x* ∗ *sgn y*
by (*simp add*: *sgn-div-norm norm-mult mult-commute*)

lemma *real-sgn-eq*: *sgn* (*x*::*real*) = *x* / |*x*|
by (*simp add*: *sgn-div-norm divide-inverse*)

lemma *real-sgn-pos*: *0* < (*x*::*real*) ⟹ *sgn x* = *1*
unfolding *real-sgn-eq* **by** *simp*

lemma *real-sgn-neg*: (*x*::*real*) < *0* ⟹ *sgn x* = −*1*
unfolding *real-sgn-eq* **by** *simp*

55.8 Bounded Linear and Bilinear Operators

locale *bounded-linear* = *additive* +
  constrains *f* :: '*a*::*real-normed-vector* ⇒ '*b*::*real-normed-vector*
  assumes *scaleR*: *f* (*scaleR r x*) = *scaleR r* (*f x*)
  assumes *bounded*: ∃ *K*. ∀ *x*. *norm* (*f x*) ≤ *norm x* ∗ *K*
begin

lemma *pos-bounded*:
$\exists K{>}0.\ \forall x.\ norm\ (f\ x) \leq norm\ x * K$
proof −
obtain K **where** K: $\bigwedge x.\ norm\ (f\ x) \leq norm\ x * K$
using *bounded* **by** *fast*
show *?thesis*
proof (*intro exI impI conjI allI*)
show $0 < max\ 1\ K$
by (*rule order-less-le-trans* [*OF zero-less-one le-maxI1*])
next
fix x
have $norm\ (f\ x) \leq norm\ x * K$ **using** K **.**
also have $\ldots \leq norm\ x * max\ 1\ K$
by (*rule mult-left-mono* [*OF le-maxI2 norm-ge-zero*])
finally show $norm\ (f\ x) \leq norm\ x * max\ 1\ K$ **.**
qed
qed

lemma *nonneg-bounded*:
$\exists K{\geq}0.\ \forall x.\ norm\ (f\ x) \leq norm\ x * K$
proof −
from *pos-bounded*
show *?thesis* **by** (*auto intro*: *order-less-imp-le*)
qed

end

locale *bounded-bilinear* =
fixes *prod* :: ['*a*::*real-normed-vector*, '*b*::*real-normed-vector*]
⇒ '*c*::*real-normed-vector*
(**infixl** ** 70*)
assumes *add-left*: $prod\ (a + a')\ b = prod\ a\ b + prod\ a'\ b$
assumes *add-right*: $prod\ a\ (b + b') = prod\ a\ b + prod\ a\ b'$
assumes *scaleR-left*: $prod\ (scaleR\ r\ a)\ b = scaleR\ r\ (prod\ a\ b)$
assumes *scaleR-right*: $prod\ a\ (scaleR\ r\ b) = scaleR\ r\ (prod\ a\ b)$
assumes *bounded*: $\exists K.\ \forall a\ b.\ norm\ (prod\ a\ b) \leq norm\ a * norm\ b * K$
begin

lemma *pos-bounded*:
$\exists K{>}0.\ \forall a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * K$
apply (*cut-tac bounded*, *erule exE*)
apply (*rule-tac x*=*max 1 K* **in** *exI*, *safe*)
apply (*rule order-less-le-trans* [*OF zero-less-one le-maxI1*])
apply (*drule spec*, *drule spec*, *erule order-trans*)
apply (*rule mult-left-mono* [*OF le-maxI2*])
apply (*intro mult-nonneg-nonneg norm-ge-zero*)
done

lemma *nonneg-bounded*:

$\exists K{\geq}0.\ \forall a\ b.\ norm\ (a\ **\ b) \leq norm\ a * norm\ b * K$
proof −
from *pos-bounded*
show *?thesis* **by** (*auto intro*: *order-less-imp-le*)
qed

lemma *additive-right*: *additive* (λ*b*. *prod a b*)
by (*rule additive.intro*, *rule add-right*)

lemma *additive-left*: *additive* (λ*a*. *prod a b*)
by (*rule additive.intro*, *rule add-left*)

lemma *zero-left*: *prod 0 b* = *0*
by (*rule additive.zero* [*OF additive-left*])

lemma *zero-right*: *prod a 0* = *0*
by (*rule additive.zero* [*OF additive-right*])

lemma *minus-left*: *prod* (− *a*) *b* = − *prod a b*
by (*rule additive.minus* [*OF additive-left*])

lemma *minus-right*: *prod a* (− *b*) = − *prod a b*
by (*rule additive.minus* [*OF additive-right*])

lemma *diff-left*:
prod (*a* − *a'*) *b* = *prod a b* − *prod a' b*
by (*rule additive.diff* [*OF additive-left*])

lemma *diff-right*:
prod a (*b* − *b'*) = *prod a b* − *prod a b'*
by (*rule additive.diff* [*OF additive-right*])

lemma *bounded-linear-left*:
bounded-linear (λ*a*. *a ** *b*)
apply (*unfold-locales*)
apply (*rule add-left*)
apply (*rule scaleR-left*)
apply (*cut-tac bounded*, *safe*)
apply (*rule-tac x*=*norm b* * *K* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma *bounded-linear-right*:
bounded-linear (λ*b*. *a ** *b*)
apply (*unfold-locales*)
apply (*rule add-right*)
apply (*rule scaleR-right*)
apply (*cut-tac bounded*, *safe*)
apply (*rule-tac x*=*norm a* * *K* **in** *exI*)

apply (*simp add*: *mult-ac*)
done

lemma *prod-diff-prod*:
(*x* ** *y* − *a* ** *b*) = (*x* − *a*) ** (*y* − *b*) + (*x* − *a*) ** *b* + *a* ** (*y* − *b*)
by (*simp add*: *diff-left diff-right*)

end

interpretation *mult*:
bounded-bilinear op * :: *'a* ⇒ *'a* ⇒ *'a*::*real-normed-algebra*
apply (*rule bounded-bilinear.intro*)
apply (*rule left-distrib*)
apply (*rule right-distrib*)
apply (*rule mult-scaleR-left*)
apply (*rule mult-scaleR-right*)
apply (*rule-tac x*=*1* **in** *exI*)
apply (*simp add*: *norm-mult-ineq*)
done

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

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

interpretation *divide*:
bounded-linear (λ*x*::*'a*::*real-normed-field*. *x* / *y*)
unfolding *divide-inverse* **by** (*rule mult.bounded-linear-left*)

interpretation *scaleR*: *bounded-bilinear scaleR*
apply (*rule bounded-bilinear.intro*)
apply (*rule scaleR-left-distrib*)
apply (*rule scaleR-right-distrib*)
apply *simp*
apply (*rule scaleR-left-commute*)
apply (*rule-tac x*=*1* **in** *exI*)
apply (*simp add*: *norm-scaleR*)
done

interpretation *scaleR-left*: *bounded-linear* λ*r*. *scaleR r x*
by (*rule scaleR.bounded-linear-left*)

interpretation *scaleR-right*: *bounded-linear* λ*x*. *scaleR r x*
by (*rule scaleR.bounded-linear-right*)

interpretation *of-real*: *bounded-linear* λ*r*. *of-real r*

unfolding *of-real-def* **by** (*rule scaleR.bounded-linear-left*)

end

theory *Real*
imports *RComplete RealVector*
begin

end

56 Fact: Factorial Function

theory *Fact*
imports *Main*
begin

consts *fact* :: *nat* => *nat*
primrec
fact-0: *fact 0 = 1*
fact-Suc: *fact (Suc n) = (Suc n) * fact n*

lemma *fact-gt-zero* [*simp*]: *0 < fact n*
by (*induct n*) *auto*

lemma *fact-not-eq-zero* [*simp*]: *fact n ≠ 0*
by *simp*

lemma *of-nat-fact-not-zero* [*simp*]: *of-nat (fact n) ≠ (0::'a::semiring-char-0)*
by *auto*

class *ordered-semiring-1 = ordered-semiring + semiring-1*
class *ordered-semiring-1-strict = ordered-semiring-strict + semiring-1*

lemma *of-nat-fact-gt-zero* [*simp*]: *(0::'a::{ordered-semidom}) < of-nat(fact n)* **by** *auto*

lemma *of-nat-fact-ge-zero* [*simp*]: *(0::'a::ordered-semidom) ≤ of-nat(fact n)*
by *simp*

lemma *fact-ge-one* [*simp*]: *1 ≤ fact n*
by (*induct n*) *auto*

lemma *fact-mono*: *m ≤ n ==> fact m ≤ fact n*
apply (*drule le-imp-less-or-eq*)
apply (*auto dest*!: *less-imp-Suc-add*)
apply (*induct-tac k*, *auto*)
done

Note that *fact 0 = fact 1*

lemma *fact-less-mono*: [| *0 < m*; *m < n* |] ==> *fact m < fact n*
apply (*drule-tac m = m* **in** *less-imp-Suc-add*, *auto*)
apply (*induct-tac k*, *auto*)
done

lemma *inv-of-nat-fact-gt-zero* [*simp*]: (*0*::*'a*::*ordered-field*) < *inverse* (*of-nat* (*fact n*))
by (*auto simp add*: *positive-imp-inverse-positive*)

lemma *inv-of-nat-fact-ge-zero* [*simp*]: (*0*::*'a*::*ordered-field*) ≤ *inverse* (*of-nat* (*fact n*))
by (*auto intro*: *order-less-imp-le*)

lemma *fact-diff-Suc* [*rule-format*]:
n < Suc m ==> fact (*Suc m − n*) = (*Suc m − n*) ∗ *fact* (*m − n*)
apply (*induct n arbitrary*: *m*)
apply *auto*
apply (*drule-tac x = m − Suc 0* **in** *meta-spec*, *auto*)
done

lemma *fact-num0*: *fact 0 = 1*
by *auto*

lemma *fact-num-eq-if*: *fact m* = (*if m=0 then 1 else m ∗ fact* (*m − 1*))
by (*cases m*) *auto*

lemma *fact-add-num-eq-if*:
fact (*m + n*) = (*if m + n = 0 then 1 else* (*m + n*) ∗ *fact* (*m + n − 1*))
by (*cases m + n*) *auto*

lemma *fact-add-num-eq-if2*:
fact (*m + n*) = (*if m = 0 then fact n else* (*m + n*) ∗ *fact* ((*m − 1*) + *n*))
by (*cases m*) *auto*

end

57 SEQ: Sequences and Convergence

theory *SEQ*
imports *RealVector RComplete*
begin

definition
Zseq :: [*nat* ⇒ *'a*::*real-normed-vector*] ⇒ *bool* **where**
— Standard definition of sequence converging to zero
[*code del*]: *Zseq X* = (∀ *r*>*0*. ∃ *no*. ∀ *n*≥*no*. *norm* (*X n*) < *r*)

definition
LIMSEQ :: [*nat* => *'a*::*real-normed-vector*, *'a*] => *bool*
(((-)/ ----> (-)) [*60*, *60*] *60*) **where**
— Standard definition of convergence of sequence
[*code del*]: $X \longrightarrow L = (\forall r.\ 0 < r \longrightarrow (\exists no.\ \forall n.\ no \leq n \longrightarrow norm\ (X\ n - L) < r))$

definition
lim :: (*nat* => *'a*::*real-normed-vector*) => *'a* **where**
— Standard definition of limit using choice operator
lim X = (*THE L*. *X* ----> *L*)

definition
convergent :: (*nat* => *'a*::*real-normed-vector*) => *bool* **where**
— Standard definition of convergence
convergent X = (∃ *L*. *X* ----> *L*)

definition
Bseq :: (*nat* => *'a*::*real-normed-vector*) => *bool* **where**
— Standard definition for bounded sequence
[*code del*]: *Bseq X* = (∃ *K*>*0*.∀ *n*. *norm* (*X n*) ≤ *K*)

definition
monoseq :: (*nat*=>*real*)=>*bool* **where**
— Definition of monotonicity. The use of disjunction here complicates proofs considerably. One alternative is to add a Boolean argument to indicate the direction. Another is to develop the notions of increasing and decreasing first.
[*code del*]: *monoseq X* = ((∀ *m*. ∀ *n*≥*m*. *X m* ≤ *X n*) | (∀ *m*. ∀ *n*≥*m*. *X n* ≤ *X m*))

definition
incseq :: (*nat*=>*real*)=>*bool* **where**
— Increasing sequence
[*code del*]: *incseq X* = (∀ *m*. ∀ *n*≥*m*. *X m* ≤ *X n*)

definition
decseq :: (*nat*=>*real*)=>*bool* **where**
— Increasing sequence
[*code del*]: *decseq X* = (∀ *m*. ∀ *n*≥*m*. *X n* ≤ *X m*)

definition
subseq :: (*nat* => *nat*) => *bool* **where**
— Definition of subsequence
[*code del*]: *subseq f* = (∀ *m*. ∀ *n*>*m*. (*f m*) < (*f n*))

definition
Cauchy :: (*nat* => *'a*::*real-normed-vector*) => *bool* **where**
— Standard definition of the Cauchy condition

[*code del*]: *Cauchy X* = (∀ *e*>*0*. ∃ *M*. ∀ *m* ≥ *M*. ∀ *n* ≥ *M*. *norm* (*X m* − *X n*) < *e*)

57.1 Bounded Sequences

lemma *BseqI'*: **assumes** *K*: ⋀*n*. *norm* (*X n*) ≤ *K* **shows** *Bseq X*
unfolding *Bseq-def*
proof (*intro exI conjI allI*)
show *0* < *max K 1* **by** *simp*
next
fix *n*::*nat*
have *norm* (*X n*) ≤ *K* **by** (*rule K*)
thus *norm* (*X n*) ≤ *max K 1* **by** *simp*
qed

lemma *BseqE*: ⟦*Bseq X*; ⋀*K*. ⟦*0* < *K*; ∀ *n*. *norm* (*X n*) ≤ *K*⟧ ⟹ *Q*⟧ ⟹ *Q*
unfolding *Bseq-def* **by** *auto*

lemma *BseqI2'*: **assumes** *K*: ∀ *n*≥*N*. *norm* (*X n*) ≤ *K* **shows** *Bseq X*
proof (*rule BseqI'*)
let *?A* = *norm* ‘ *X* ‘ {*..N*}
have *1*: *finite ?A* **by** *simp*
fix *n*::*nat*
show *norm* (*X n*) ≤ *max K* (*Max ?A*)
proof (*cases rule*: *linorder-le-cases*)
assume *n* ≥ *N*
hence *norm* (*X n*) ≤ *K* **using** *K* **by** *simp*
thus *norm* (*X n*) ≤ *max K* (*Max ?A*) **by** *simp*
next
assume *n* ≤ *N*
hence *norm* (*X n*) ∈ *?A* **by** *simp*
with *1* **have** *norm* (*X n*) ≤ *Max ?A* **by** (*rule Max-ge*)
thus *norm* (*X n*) ≤ *max K* (*Max ?A*) **by** *simp*
qed
qed

lemma *Bseq-ignore-initial-segment*: *Bseq X* ⟹ *Bseq* (λ*n*. *X* (*n* + *k*))
unfolding *Bseq-def* **by** *auto*

lemma *Bseq-offset*: *Bseq* (λ*n*. *X* (*n* + *k*)) ⟹ *Bseq X*
apply (*erule BseqE*)
apply (*rule-tac N*=*k* **and** *K*=*K* **in** *BseqI2'*)
apply *clarify*
apply (*drule-tac x*=*n* − *k* **in** *spec*, *simp*)
done

57.2 Sequences That Converge to Zero

lemma *ZseqI*:
(⋀*r*. *0* < *r* ⟹ ∃ *no*. ∀ *n*≥*no*. *norm* (*X n*) < *r*) ⟹ *Zseq X*

unfolding *Zseq-def* **by** *simp*

lemma *ZseqD*:
  ⟦*Zseq X*; *0* < *r*⟧ ⟹ ∃ *no*. ∀ *n*≥*no*. *norm* (*X n*) < *r*
unfolding *Zseq-def* **by** *simp*

lemma *Zseq-zero*: *Zseq* (λ*n*. *0*)
unfolding *Zseq-def* **by** *simp*

lemma *Zseq-const-iff*: *Zseq* (λ*n*. *k*) = (*k* = *0*)
unfolding *Zseq-def* **by** *force*

lemma *Zseq-norm-iff*: *Zseq* (λ*n*. *norm* (*X n*)) = *Zseq* (λ*n*. *X n*)
unfolding *Zseq-def* **by** *simp*

lemma *Zseq-imp-Zseq*:
  assumes *X*: *Zseq X*
  assumes *Y*: ⋀*n*. *norm* (*Y n*) ≤ *norm* (*X n*) * *K*
  shows *Zseq* (λ*n*. *Y n*)
proof (*cases*)
  assume *K*: *0* < *K*
  show *?thesis*
  proof (*rule ZseqI*)
    fix *r*::*real* **assume** *0* < *r*
    hence *0* < *r* / *K*
      using *K* **by** (*rule divide-pos-pos*)
    then obtain *N* **where** ∀ *n*≥*N*. *norm* (*X n*) < *r* / *K*
      using *ZseqD* [*OF X*] **by** *fast*
    hence ∀ *n*≥*N*. *norm* (*X n*) * *K* < *r*
      by (*simp add*: *pos-less-divide-eq K*)
    hence ∀ *n*≥*N*. *norm* (*Y n*) < *r*
      by (*simp add*: *order-le-less-trans* [*OF Y*])
    thus ∃ *N*. ∀ *n*≥*N*. *norm* (*Y n*) < *r* **..**
  qed
next
  assume ¬ *0* < *K*
  hence *K*: *K* ≤ *0* **by** (*simp only*: *linorder-not-less*)
  {
    fix *n*::*nat*
    have *norm* (*Y n*) ≤ *norm* (*X n*) * *K* **by** (*rule Y*)
    also have ... ≤ *norm* (*X n*) * *0*
      using *K norm-ge-zero* **by** (*rule mult-left-mono*)
    finally have *norm* (*Y n*) = *0* **by** *simp*
  }
  thus *?thesis* **by** (*simp add*: *Zseq-zero*)
qed

lemma *Zseq-le*: ⟦*Zseq Y*; ∀ *n*. *norm* (*X n*) ≤ *norm* (*Y n*)⟧ ⟹ *Zseq X*
by (*erule-tac K=1* **in** *Zseq-imp-Zseq*, *simp*)

lemma *Zseq-add*:
assumes X: *Zseq X*
assumes Y: *Zseq Y*
shows *Zseq* $(\lambda n.\ X\ n + Y\ n)$
proof (*rule ZseqI*)
fix r::*real* **assume** $0 < r$
hence r: $0 < r\ /\ 2$ **by** *simp*
obtain M **where** M: $\forall\, n{\geq}M.\ norm\ (X\ n) < r/2$
using *ZseqD* [*OF X r*] **by** *fast*
obtain N **where** N: $\forall\, n{\geq}N.\ norm\ (Y\ n) < r/2$
using *ZseqD* [*OF Y r*] **by** *fast*
show $\exists\, N.\ \forall\, n{\geq}N.\ norm\ (X\ n + Y\ n) < r$
proof (*intro exI allI impI*)
fix n **assume** n: $max\ M\ N \leq n$
have $norm\ (X\ n + Y\ n) \leq norm\ (X\ n) + norm\ (Y\ n)$
by (*rule norm-triangle-ineq*)
also have $\ldots < r/2 + r/2$
proof (*rule add-strict-mono*)
from $M\ n$ **show** $norm\ (X\ n) < r/2$ **by** *simp*
from $N\ n$ **show** $norm\ (Y\ n) < r/2$ **by** *simp*
qed
finally show $norm\ (X\ n + Y\ n) < r$ **by** *simp*
qed
qed

lemma *Zseq-minus*: *Zseq X* $\Longrightarrow$ *Zseq* $(\lambda n.\ -\ X\ n)$
unfolding *Zseq-def* **by** *simp*

lemma *Zseq-diff*: $[\![$*Zseq X*; *Zseq Y*$]\!] \Longrightarrow$ *Zseq* $(\lambda n.\ X\ n - Y\ n)$
by (*simp only*: *diff-minus Zseq-add Zseq-minus*)

lemma (**in** *bounded-linear*) *Zseq*:
assumes X: *Zseq X*
shows *Zseq* $(\lambda n.\ f\ (X\ n))$
proof −
obtain K **where** $\bigwedge x.\ norm\ (f\ x) \leq norm\ x * K$
using *bounded* **by** *fast*
with X **show** *?thesis*
by (*rule Zseq-imp-Zseq*)
qed

lemma (**in** *bounded-bilinear*) *Zseq*:
assumes X: *Zseq X*
assumes Y: *Zseq Y*
shows *Zseq* $(\lambda n.\ X\ n **\ Y\ n)$
proof (*rule ZseqI*)
fix r::*real* **assume** r: $0 < r$
obtain K **where** K: $0 < K$

```
    and norm-le: ⋀x y. norm (x ** y) ≤ norm x * norm y * K
    using pos-bounded by fast
  from K have K': 0 < inverse K
    by (rule positive-imp-inverse-positive)
  obtain M where M: ∀n≥M. norm (X n) < r
    using ZseqD [OF X r] by fast
  obtain N where N: ∀n≥N. norm (Y n) < inverse K
    using ZseqD [OF Y K'] by fast
  show ∃N. ∀n≥N. norm (X n ** Y n) < r
  proof (intro exI allI impI)
    fix n assume n: max M N ≤ n
    have norm (X n ** Y n) ≤ norm (X n) * norm (Y n) * K
      by (rule norm-le)
    also have norm (X n) * norm (Y n) * K < r * inverse K * K
    proof (intro mult-strict-right-mono mult-strict-mono' norm-ge-zero K)
      from M n show Xn: norm (X n) < r by simp
      from N n show Yn: norm (Y n) < inverse K by simp
    qed
    also from K have r * inverse K * K = r by simp
    finally show norm (X n ** Y n) < r .
  qed
qed

lemma (in bounded-bilinear) Zseq-prod-Bseq:
  assumes X: Zseq X
  assumes Y: Bseq Y
  shows Zseq (λn. X n ** Y n)
proof -
  obtain K where K: 0 ≤ K
    and norm-le: ⋀x y. norm (x ** y) ≤ norm x * norm y * K
    using nonneg-bounded by fast
  obtain B where B: 0 < B
    and norm-Y: ⋀n. norm (Y n) ≤ B
    using Y [unfolded Bseq-def] by fast
  from X show ?thesis
  proof (rule Zseq-imp-Zseq)
    fix n::nat
    have norm (X n ** Y n) ≤ norm (X n) * norm (Y n) * K
      by (rule norm-le)
    also have ... ≤ norm (X n) * B * K
      by (intro mult-mono' order-refl norm-Y norm-ge-zero
                mult-nonneg-nonneg K)
    also have ... = norm (X n) * (B * K)
      by (rule mult-assoc)
    finally show norm (X n ** Y n) ≤ norm (X n) * (B * K) .
  qed
qed

lemma (in bounded-bilinear) Bseq-prod-Zseq:
```

assumes X: *Bseq X*
assumes Y: *Zseq Y*
shows *Zseq* ($\lambda n.$ $X\ n$ ** $Y\ n$)
proof −
obtain K **where** K: $0 \leq K$
and *norm-le*: $\bigwedge x\ y.$ *norm* $(x$ ** $y) \leq$ *norm* $x *$ *norm* $y * K$
using *nonneg-bounded* **by** *fast*
obtain B **where** B: $0 < B$
and *norm-X*: $\bigwedge n.$ *norm* $(X\ n) \leq B$
using X [*unfolded Bseq-def*] **by** *fast*
from Y **show** *?thesis*
proof (*rule Zseq-imp-Zseq*)
fix *n*::*nat*
have *norm* $(X\ n$ ** $Y\ n) \leq$ *norm* $(X\ n) *$ *norm* $(Y\ n) * K$
by (*rule norm-le*)
also have $\ldots \leq B *$ *norm* $(Y\ n) * K$
by (*intro mult-mono′ order-refl norm-X norm-ge-zero*
mult-nonneg-nonneg K)
also have $\ldots =$ *norm* $(Y\ n) * (B * K)$
by (*simp only*: *mult-ac*)
finally show *norm* $(X\ n$ ** $Y\ n) \leq$ *norm* $(Y\ n) * (B * K)$ **.**
qed
qed

lemma (**in** *bounded-bilinear*) *Zseq-left*:
Zseq $X \Longrightarrow$ *Zseq* $(\lambda n.$ $X\ n$ ** $a)$
by (*rule bounded-linear-left* [*THEN bounded-linear.Zseq*])

lemma (**in** *bounded-bilinear*) *Zseq-right*:
Zseq $X \Longrightarrow$ *Zseq* $(\lambda n.$ a ** $X\ n)$
by (*rule bounded-linear-right* [*THEN bounded-linear.Zseq*])

lemmas *Zseq-mult* = *mult.Zseq*
lemmas *Zseq-mult-right* = *mult.Zseq-right*
lemmas *Zseq-mult-left* = *mult.Zseq-left*

57.3 Limits of Sequences

lemma *LIMSEQ-iff*:
$(X$ −−−−> $L) = (\forall r{>}0.\ \exists no.\ \forall n \geq no.$ *norm* $(X\ n - L) < r)$
by (*rule LIMSEQ-def*)

lemma *LIMSEQ-Zseq-iff*: $((\lambda n.\ X\ n)$ −−−−> $L) =$ *Zseq* $(\lambda n.\ X\ n - L)$
by (*simp only*: *LIMSEQ-def Zseq-def*)

lemma *LIMSEQ-I*:
$(\bigwedge r.\ 0 < r \Longrightarrow \exists no.\ \forall n{\geq}no.$ *norm* $(X\ n - L) < r) \Longrightarrow X$ −−−−> L
by (*simp add*: *LIMSEQ-def*)

lemma *LIMSEQ-D*:
$[\![X \longrightarrow L;\ 0 < r]\!] \Longrightarrow \exists no.\ \forall n \geq no.\ norm\ (X\ n - L) < r$
by (*simp add*: *LIMSEQ-def*)

lemma *LIMSEQ-const*: $(\lambda n.\ k) \longrightarrow k$
by (*simp add*: *LIMSEQ-def*)

lemma *LIMSEQ-const-iff*: $(\lambda n.\ k) \longrightarrow l = (k = l)$
by (*simp add*: *LIMSEQ-Zseq-iff Zseq-const-iff*)

lemma *LIMSEQ-norm*: $X \longrightarrow a \Longrightarrow (\lambda n.\ norm\ (X\ n)) \longrightarrow norm\ a$
apply (*simp add*: *LIMSEQ-def*, *safe*)
apply (*drule-tac x=r* **in** *spec*, *safe*)
apply (*rule-tac x=no* **in** *exI*, *safe*)
apply (*drule-tac x=n* **in** *spec*, *safe*)
apply (*erule order-le-less-trans* [*OF norm-triangle-ineq3*])
done

lemma *LIMSEQ-ignore-initial-segment*:
$f \longrightarrow a \Longrightarrow (\lambda n.\ f\ (n + k)) \longrightarrow a$
apply (*rule LIMSEQ-I*)
apply (*drule* (*1*) *LIMSEQ-D*)
apply (*erule exE*, *rename-tac N*)
apply (*rule-tac x=N* **in** *exI*)
apply *simp*
done

lemma *LIMSEQ-offset*:
$(\lambda n.\ f\ (n + k)) \longrightarrow a \Longrightarrow f \longrightarrow a$
apply (*rule LIMSEQ-I*)
apply (*drule* (*1*) *LIMSEQ-D*)
apply (*erule exE*, *rename-tac N*)
apply (*rule-tac x=N + k* **in** *exI*)
apply *clarify*
apply (*drule-tac x=n - k* **in** *spec*)
apply (*simp add*: *le-diff-conv2*)
done

lemma *LIMSEQ-Suc*: $f \longrightarrow l \Longrightarrow (\lambda n.\ f\ (Suc\ n)) \longrightarrow l$
by (*drule-tac k=Suc 0* **in** *LIMSEQ-ignore-initial-segment*, *simp*)

lemma *LIMSEQ-imp-Suc*: $(\lambda n.\ f\ (Suc\ n)) \longrightarrow l \Longrightarrow f \longrightarrow l$
by (*rule-tac k=Suc 0* **in** *LIMSEQ-offset*, *simp*)

lemma *LIMSEQ-Suc-iff*: $(\lambda n.\ f\ (Suc\ n)) \longrightarrow l = f \longrightarrow l$
by (*blast intro*: *LIMSEQ-imp-Suc LIMSEQ-Suc*)

lemma *LIMSEQ-linear*: $[\![\ X \longrightarrow x\ ;\ l > 0\ ]\!] \Longrightarrow (\lambda\ n.\ X\ (n * l)) \longrightarrow x$

unfolding *LIMSEQ-def*
by (*metis div-le-dividend div-mult-self1-is-m le-trans nat-mult-commute*)

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 *minus-diff-minus*:
fixes *a b* :: *'a*::*ab-group-add*
shows $(- a) - (- b) = - (a - b)$
by *simp*

lemma *LIMSEQ-add*: $[\![X \longrightarrow a;\ Y \longrightarrow b]\!] \Longrightarrow (\lambda n.\ X\ n + Y\ n) \longrightarrow a + b$
by (*simp only*: *LIMSEQ-Zseq-iff add-diff-add Zseq-add*)

lemma *LIMSEQ-minus*: $X \longrightarrow a \Longrightarrow (\lambda n.\ - X\ n) \longrightarrow - a$
by (*simp only*: *LIMSEQ-Zseq-iff minus-diff-minus Zseq-minus*)

lemma *LIMSEQ-minus-cancel*: $(\lambda n.\ - X\ n) \longrightarrow - a \Longrightarrow X \longrightarrow a$
by (*drule LIMSEQ-minus*, *simp*)

lemma *LIMSEQ-diff*: $[\![X \longrightarrow a;\ Y \longrightarrow b]\!] \Longrightarrow (\lambda n.\ X\ n - Y\ n) \longrightarrow a - b$
by (*simp add*: *diff-minus LIMSEQ-add LIMSEQ-minus*)

lemma *LIMSEQ-unique*: $[\![X \longrightarrow a;\ X \longrightarrow b]\!] \Longrightarrow a = b$
by (*drule* (*1*) *LIMSEQ-diff*, *simp add*: *LIMSEQ-const-iff*)

lemma (**in** *bounded-linear*) *LIMSEQ*:
$X \longrightarrow a \Longrightarrow (\lambda n.\ f\ (X\ n)) \longrightarrow f\ a$
by (*simp only*: *LIMSEQ-Zseq-iff diff* [*symmetric*] *Zseq*)

lemma (**in** *bounded-bilinear*) *LIMSEQ*:
$[\![X \longrightarrow a;\ Y \longrightarrow b]\!] \Longrightarrow (\lambda n.\ X\ n ** Y\ n) \longrightarrow a ** b$
by (*simp only*: *LIMSEQ-Zseq-iff prod-diff-prod*
Zseq-add Zseq Zseq-left Zseq-right)

lemma *LIMSEQ-mult*:
fixes *a b* :: *'a*::*real-normed-algebra*
shows $[|\ X \longrightarrow a;\ Y \longrightarrow b\ |] ==> (\%n.\ X\ n * Y\ n) \longrightarrow a * b$
by (*rule mult.LIMSEQ*)

lemma *inverse-diff-inverse*:
$[\![(a::'a::division\text{-}ring) \neq 0;\ b \neq 0]\!]$
$\Longrightarrow inverse\ a - inverse\ b = - (inverse\ a * (a - b) * inverse\ b)$
by (*simp add*: *algebra-simps*)

lemma *Bseq-inverse-lemma*:
fixes x :: $'a$::*real-normed-div-algebra*
shows $[\![r \leq norm\ x;\ 0 < r]\!] \Longrightarrow norm\ (inverse\ x) \leq inverse\ r$
apply (*subst nonzero-norm-inverse*, *clarsimp*)
apply (*erule* (*1*) *le-imp-inverse-le*)
done

lemma *Bseq-inverse*:
fixes a :: $'a$::*real-normed-div-algebra*
assumes *X*: X ----> a
assumes *a*: $a \neq 0$
shows *Bseq* $(\lambda n.\ inverse\ (X\ n))$
proof −
from a **have** $0 < norm\ a$ **by** *simp*
hence $\exists r{>}0.\ r < norm\ a$ **by** (*rule dense*)
then obtain r **where** *r1*: $0 < r$ **and** *r2*: $r < norm\ a$ **by** *fast*
obtain N **where** *N*: $\bigwedge n.\ N \leq n \Longrightarrow norm\ (X\ n - a) < r$
using *LIMSEQ-D* [*OF X r1*] **by** *fast*
show *?thesis*
proof (*rule BseqI2'* [*rule-format*])
fix n **assume** *n*: $N \leq n$
hence *1*: $norm\ (X\ n - a) < r$ **by** (*rule N*)
hence *2*: $X\ n \neq 0$ **using** *r2* **by** *auto*
hence $norm\ (inverse\ (X\ n)) = inverse\ (norm\ (X\ n))$
by (*rule nonzero-norm-inverse*)
also have $\ldots \leq inverse\ (norm\ a - r)$
proof (*rule le-imp-inverse-le*)
show $0 < norm\ a - r$ **using** *r2* **by** *simp*
next
have $norm\ a - norm\ (X\ n) \leq norm\ (a - X\ n)$
by (*rule norm-triangle-ineq2*)
also have $\ldots = norm\ (X\ n - a)$
by (*rule norm-minus-commute*)
also have $\ldots < r$ **using** *1* **.**
finally show $norm\ a - r \leq norm\ (X\ n)$ **by** *simp*
qed
finally show $norm\ (inverse\ (X\ n)) \leq inverse\ (norm\ a - r)$ **.**
qed
qed

lemma *LIMSEQ-inverse-lemma*:
fixes a :: $'a$::*real-normed-div-algebra*
shows $[\![X$ ----> $a;\ a \neq 0;\ \forall n.\ X\ n \neq 0]\!]$
$\Longrightarrow (\lambda n.\ inverse\ (X\ n))$ ----> $inverse\ a$
apply (*subst LIMSEQ-Zseq-iff*)
apply (*simp add*: *inverse-diff-inverse nonzero-imp-inverse-nonzero*)
apply (*rule Zseq-minus*)
apply (*rule Zseq-mult-left*)

apply (*rule mult.Bseq-prod-Zseq*)
apply (*erule* (*1*) *Bseq-inverse*)
apply (*simp add*: *LIMSEQ-Zseq-iff*)
done

lemma *LIMSEQ-inverse*:
fixes *a* :: *'a*::*real-normed-div-algebra*
assumes *X*: *X* ----> *a*
assumes *a*: *a* ≠ *0*
shows (λ*n*. *inverse* (*X n*)) ----> *inverse a*
proof −
from *a* **have** *0* < *norm a* **by** *simp*
then obtain *k* **where** ∀ *n*≥*k*. *norm* (*X n* − *a*) < *norm a*
using *LIMSEQ-D* [*OF X*] **by** *fast*
hence ∀ *n*≥*k*. *X n* ≠ *0* **by** *auto*
hence *k*: ∀ *n*. *X* (*n* + *k*) ≠ *0* **by** *simp*

from *X* **have** (λ*n*. *X* (*n* + *k*)) ----> *a*
by (*rule LIMSEQ-ignore-initial-segment*)
hence (λ*n*. *inverse* (*X* (*n* + *k*))) ----> *inverse a*
using *a k* **by** (*rule LIMSEQ-inverse-lemma*)
thus (λ*n*. *inverse* (*X n*)) ----> *inverse a*
by (*rule LIMSEQ-offset*)
qed

lemma *LIMSEQ-divide*:
fixes *a b* :: *'a*::*real-normed-field*
shows ⟦*X* ----> *a*; *Y* ----> *b*; *b* ≠ *0*⟧ ⟹ (λ*n*. *X n* / *Y n*) ----> *a* / *b*
by (*simp add*: *LIMSEQ-mult LIMSEQ-inverse divide-inverse*)

lemma *LIMSEQ-pow*:
fixes *a* :: *'a*::{*real-normed-algebra*,*recpower*}
shows *X* ----> *a* ⟹ (λ*n*. (*X n*) ^ *m*) ----> *a* ^ *m*
by (*induct m*) (*simp-all add*: *LIMSEQ-const LIMSEQ-mult*)

lemma *LIMSEQ-setsum*:
assumes *n*: ⋀*n*. *n* ∈ *S* ⟹ *X n* ----> *L n*
shows (λ*m*. ∑ *n*∈*S*. *X n m*) ----> (∑ *n*∈*S*. *L n*)
proof (*cases finite S*)
case *True*
thus *?thesis* **using** *n*
proof (*induct*)
case *empty*
show *?case*
by (*simp add*: *LIMSEQ-const*)
next
case *insert*
thus *?case*

```
    by (simp add: LIMSEQ-add)
  qed
next
  case False
  thus ?thesis
    by (simp add: LIMSEQ-const)
qed

lemma LIMSEQ-setprod:
  fixes L :: 'a ⇒ 'b::{real-normed-algebra,comm-ring-1}
  assumes n: ⋀n. n ∈ S ⟹ X n ----> L n
  shows (λm. ∏ n∈S. X n m) ----> (∏ n∈S. L n)
proof (cases finite S)
  case True
  thus ?thesis using n
  proof (induct)
    case empty
    show ?case
      by (simp add: LIMSEQ-const)
  next
    case insert
    thus ?case
      by (simp add: LIMSEQ-mult)
  qed
next
  case False
  thus ?thesis
    by (simp add: setprod-def LIMSEQ-const)
qed

lemma LIMSEQ-add-const: f ----> a ==> (%n.(f n + b)) ----> a + b
by (simp add: LIMSEQ-add LIMSEQ-const)


lemma LIMSEQ-add-minus:
    [| X ----> a; Y ----> b |] ==> (%n. X n + -Y n) ----> a + -b
by (simp only: LIMSEQ-add LIMSEQ-minus)

lemma LIMSEQ-diff-const: f ----> a ==> (%n.(f n  - b)) ----> a - b
by (simp add: LIMSEQ-diff LIMSEQ-const)

lemma LIMSEQ-diff-approach-zero:
  g ----> L ==> (%x. f x - g x) ----> 0  ==>
    f ----> L
  apply (drule LIMSEQ-add)
  apply assumption
  apply simp
done
```

lemma *LIMSEQ-diff-approach-zero2*:
f ----> *L* ==> (%*x*. *f x* − *g x*) ----> *0* ==>
g ----> *L*
apply (*drule LIMSEQ-diff*)
apply *assumption*
apply *simp*
done

A sequence tends to zero iff its abs does

lemma *LIMSEQ-norm-zero*: ((λ*n*. *norm* (*X n*)) ----> *0*) = (*X* ----> *0*)
by (*simp add*: *LIMSEQ-def*)

lemma *LIMSEQ-rabs-zero*: ((%*n*. |*f n*|) ----> *0*) = (*f* ----> (*0*::*real*))
by (*simp add*: *LIMSEQ-def*)

lemma *LIMSEQ-imp-rabs*: *f* ----> (*l*::*real*) ==> (%*n*. |*f n*|) ----> |*l*|
by (*drule LIMSEQ-norm*, *simp*)

An unbounded sequence's inverse tends to 0

lemma *LIMSEQ-inverse-zero*:
∀ *r*::*real*. ∃ *N*. ∀ *n*≥*N*. *r* < *X n* ⟹ (λ*n*. *inverse* (*X n*)) ----> *0*
apply (*rule LIMSEQ-I*)
apply (*drule-tac x*=*inverse r* **in** *spec*, *safe*)
apply (*rule-tac x*=*N* **in** *exI*, *safe*)
apply (*drule-tac x*=*n* **in** *spec*, *safe*)
apply (*frule positive-imp-inverse-positive*)
apply (*frule* (*1*) *less-imp-inverse-less*)
apply (*subgoal-tac* *0* < *X n*, *simp*)
apply (*erule* (*1*) *order-less-trans*)
done

The sequence (*1*::′*a*) / *n* tends to 0 as *n* tends to infinity

lemma *LIMSEQ-inverse-real-of-nat*: (%*n*. *inverse*(*real*(*Suc n*))) ----> *0*
apply (*rule LIMSEQ-inverse-zero*, *safe*)
apply (*cut-tac x* = *r* **in** *reals-Archimedean2*)
apply (*safe*, *rule-tac x* = *n* **in** *exI*)
apply (*auto simp add*: *real-of-nat-Suc*)
done

The sequence *r* + (*1*::′*a*) / *n* tends to *r* as *n* tends to infinity is now easily proved

lemma *LIMSEQ-inverse-real-of-nat-add*:
(%*n*. *r* + *inverse*(*real*(*Suc n*))) ----> *r*
by (*cut-tac LIMSEQ-add* [*OF LIMSEQ-const LIMSEQ-inverse-real-of-nat*], *auto*)

lemma *LIMSEQ-inverse-real-of-nat-add-minus*:
(%*n*. *r* + −*inverse*(*real*(*Suc n*))) ----> *r*
by (*cut-tac LIMSEQ-add-minus* [*OF LIMSEQ-const LIMSEQ-inverse-real-of-nat*], *auto*)

lemma *LIMSEQ-inverse-real-of-nat-add-minus-mult*:
(%*n*. *r*∗(*1* + −*inverse*(*real*(*Suc n*)))) −−−−> *r*
by (*cut-tac b=1* **in**
LIMSEQ-mult [*OF LIMSEQ-const LIMSEQ-inverse-real-of-nat-add-minus*],
auto)

lemma *LIMSEQ-le-const*:
⟦*X* −−−−> (*x*::*real*); ∃ *N*. ∀ *n*≥*N*. *a* ≤ *X n*⟧ ⟹ *a* ≤ *x*
apply (*rule ccontr*, *simp only*: *linorder-not-le*)
apply (*drule-tac r=a* − *x* **in** *LIMSEQ-D*, *simp*)
apply *clarsimp*
apply (*drule-tac x=max N no* **in** *spec*, *drule mp*, *rule le-maxI1*)
apply (*drule-tac x=max N no* **in** *spec*, *drule mp*, *rule le-maxI2*)
apply *simp*
done

lemma *LIMSEQ-le-const2*:
⟦*X* −−−−> (*x*::*real*); ∃ *N*. ∀ *n*≥*N*. *X n* ≤ *a*⟧ ⟹ *x* ≤ *a*
apply (*subgoal-tac* − *a* ≤ − *x*, *simp*)
apply (*rule LIMSEQ-le-const*)
apply (*erule LIMSEQ-minus*)
apply *simp*
done

lemma *LIMSEQ-le*:
⟦*X* −−−−> *x*; *Y* −−−−> *y*; ∃ *N*. ∀ *n*≥*N*. *X n* ≤ *Y n*⟧ ⟹ *x* ≤ (*y*::*real*)
apply (*subgoal-tac 0* ≤ *y* − *x*, *simp*)
apply (*rule LIMSEQ-le-const*)
apply (*erule* (*1*) *LIMSEQ-diff*)
apply (*simp add*: *le-diff-eq*)
done

57.4 Convergence

lemma *limI*: *X* −−−−> *L* ==> *lim X* = *L*
apply (*simp add*: *lim-def*)
apply (*blast intro*: *LIMSEQ-unique*)
done

lemma *convergentD*: *convergent X* ==> ∃ *L*. (*X* −−−−> *L*)
by (*simp add*: *convergent-def*)

lemma *convergentI*: (*X* −−−−> *L*) ==> *convergent X*
by (*auto simp add*: *convergent-def*)

lemma *convergent-LIMSEQ-iff*: *convergent X* = (*X* −−−−> *lim X*)
by (*auto intro*: *theI LIMSEQ-unique simp add*: *convergent-def lim-def*)

lemma *convergent-minus-iff*: (*convergent X*) = (*convergent* (%*n*. −(*X n*)))
apply (*simp add*: *convergent-def*)
apply (*auto dest*: *LIMSEQ-minus*)
apply (*drule LIMSEQ-minus*, *auto*)
done

Given a binary function $f :: nat \Rightarrow 'a \Rightarrow 'a$, its values are uniquely determined by a function g

lemma *nat-function-unique*: *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

Subsequence (alternative definition, (e.g. Hoskins)

lemma *subseq-Suc-iff*: *subseq f* = (∀ *n*. (*f n*) < (*f* (*Suc n*)))
apply (*simp add*: *subseq-def*)
apply (*auto dest*!: *less-imp-Suc-add*)
apply (*induct-tac k*)
apply (*auto intro*: *less-trans*)
done

lemma *monoseq-Suc*:
  monoseq X = ((∀ *n*. *X n* ≤ *X* (*Suc n*))
    | (∀ *n*. *X* (*Suc n*) ≤ *X n*))
apply (*simp add*: *monoseq-def*)
apply (*auto dest*!: *le-imp-less-or-eq*)
apply (*auto intro*!: *lessI* [*THEN less-imp-le*] *dest*!: *less-imp-Suc-add*)
apply (*induct-tac ka*)
apply (*auto intro*: *order-trans*)
apply (*erule contrapos-np*)
apply (*induct-tac k*)
apply (*auto intro*: *order-trans*)
done

lemma *monoI1*: ∀ *m*. ∀ *n* ≥ *m*. *X m* ≤ *X n* ==> *monoseq X*
by (*simp add*: *monoseq-def*)

lemma *monoI2*: ∀ *m*. ∀ *n* ≥ *m*. *X n* ≤ *X m* ==> *monoseq X*
by (*simp add*: *monoseq-def*)

lemma *mono-SucI1*: $\forall n.\ X\ n \leq X\ (Suc\ n) ==> monoseq\ X$
by (*simp add*: *monoseq-Suc*)

lemma *mono-SucI2*: $\forall n.\ X\ (Suc\ n) \leq X\ n ==> monoseq\ X$
by (*simp add*: *monoseq-Suc*)

lemma *monoseq-minus*: **assumes** *monoseq a*
shows *monoseq* $(\lambda\ n.\ -\ a\ n)$
proof (*cases* $\forall\ m.\ \forall\ n \geq m.\ a\ m \leq a\ n$)
case *True*
hence $\forall\ m.\ \forall\ n \geq m. - a\ n \leq - a\ m$ **by** *auto*
thus *?thesis* **by** (*rule monoI2*)
next
case *False*
hence $\forall\ m.\ \forall\ n \geq m. - a\ m \leq - a\ n$ **using** ‹*monoseq a*›[*unfolded monoseq-def*]
by *auto*
thus *?thesis* **by** (*rule monoI1*)
qed

lemma *monoseq-le*: **assumes** *monoseq a* **and** a ----> x
shows $((\forall\ n.\ a\ n \leq x) \wedge (\forall m.\ \forall n \geq m.\ a\ m \leq a\ n)) \vee$
$((\forall\ n.\ x \leq a\ n) \wedge (\forall m.\ \forall n \geq m.\ a\ n \leq a\ m))$
proof −
{ fix *x n* **fix** *a* :: *nat* ⇒ *real*
assume a ----> x **and** $\forall\ m.\ \forall\ n \geq m.\ a\ m \leq a\ n$
hence *monotone*: $\bigwedge m\ n.\ m \leq n \Longrightarrow a\ m \leq a\ n$ **by** *auto*
have $a\ n \leq x$
proof (*rule ccontr*)
assume $\neg\ a\ n \leq x$ **hence** $x < a\ n$ **by** *auto*
hence $0 < a\ n - x$ **by** *auto*
from ‹a ----> x›[*THEN LIMSEQ-D*, *OF this*]
obtain *no* **where** $\bigwedge n'.\ no \leq n' \Longrightarrow norm\ (a\ n' - x) < a\ n - x$ **by** *blast*
hence $norm\ (a\ (max\ no\ n) - x) < a\ n - x$ **by** *auto*
moreover
{ fix n' **have** $n \leq n' \Longrightarrow x < a\ n'$ **using** *monotone*[**where** $m{=}n$ **and** $n{=}n'$]
and ‹$x < a\ n$› **by** *auto* **}**
hence $x < a\ (max\ no\ n)$ **by** *auto*
ultimately
have $a\ (max\ no\ n) < a\ n$ **by** *auto*
with *monotone*[**where** $m{=}n$ **and** $n{=}max\ no\ n$]
show *False* **by** *auto*
qed
} note *top-down* = *this*
{ fix *x n m* **fix** *a* :: *nat* ⇒ *real*
assume a ----> x **and** *monoseq a* **and** $a\ m < x$
have $a\ n \leq x \wedge (\forall\ m.\ \forall\ n \geq m.\ a\ m \leq a\ n)$
proof (*cases* $\forall\ m.\ \forall\ n \geq m.\ a\ m \leq a\ n$)
case *True* **with** *top-down* **and** ‹a ----> x› **show** *?thesis* **by** *auto*
next

```
    case False with ‹monoseq a›[unfolded monoseq-def] have ∀ m. ∀ n ≥ m. −
a m ≤ − a n by auto
     hence − a m ≤ − x using top-down[OF LIMSEQ-minus[OF ‹a −−−−> x›]] by blast
    hence False using ‹a m < x› by auto
    thus ?thesis ..
   qed
  } note when-decided = this

  show ?thesis
  proof (cases ∃ m. a m ≠ x)
   case True then obtain m where a m ≠ x by auto
   show ?thesis
   proof (cases a m < x)
    case True with when-decided[OF ‹a −−−−> x› ‹monoseq a›, where m2=m]
     show ?thesis by blast
   next
    case False hence − a m < − x using ‹a m ≠ x› by auto
   with when-decided[OF LIMSEQ-minus[OF ‹a −−−−> x›] monoseq-minus[OF
‹monoseq a›], where m2=m]
    show ?thesis by auto
   qed
  qed auto
qed
```

for any sequence, there is a mootonic subsequence

```
lemma seq-monosub: ∃f. subseq f ∧ monoseq (λ n. (s (f n)))
proof−
  {assume H: ∀ n. ∃ p >n. ∀ m≥p. s m ≤ s p
   let ?P = λ p n. p > n ∧ (∀ m ≥ p. s m ≤ s p)
   from nat-function-unique[of SOME p. ?P p 0 λp n. SOME p. ?P p n]
   obtain f where f: f 0 = (SOME p. ?P p 0) ∀ n. f (Suc n) = (SOME p. ?P
p (f n)) by blast
   have ?P (f 0) 0 unfolding f(1) some-eq-ex[of λp. ?P p 0]
    using H apply −
    apply (erule allE[where x=0], erule exE, rule-tac x=p in exI)
    unfolding order-le-less by blast
   hence f0: f 0 > 0 ∀ m ≥ f 0. s m ≤ s (f 0) by blast+
   {fix n
    have ?P (f (Suc n)) (f n)
     unfolding f(2)[rule-format, of n] some-eq-ex[of λp. ?P p (f n)]
     using H apply −
    apply (erule allE[where x=f n], erule exE, rule-tac x=p in exI)
    unfolding order-le-less by blast
   hence f (Suc n) > f n ∀ m ≥ f (Suc n). s m ≤ s (f (Suc n)) by blast+}
  note fSuc = this
   {fix p q assume pq: p ≥ f q
    have s p ≤ s(f(q)) using f0(2)[rule-format, of p] pq fSuc
     by (cases q, simp-all) }
```

```
  note pqth = this
  {fix q
   have f (Suc q) > f q apply (induct q)
    using f0(1) fSuc(1)[of 0] apply simp by (rule fSuc(1))}
  note fss = this
  from fss have th1: subseq f unfolding subseq-Suc-iff ..
  {fix a b
   have f a ≤ f (a + b)
   proof(induct b)
    case 0 thus ?case by simp
   next
    case (Suc b)
    from fSuc(1)[of a + b] Suc.hyps show ?case by simp
   qed}
  note fmon0 = this
  have monoseq (λn. s (f n))
  proof−
   {fix n
    have s (f n) ≥ s (f (Suc n))
    proof(cases n)
     case 0
     assume n0: n = 0
     from fSuc(1)[of 0] have th0: f 0 ≤ f (Suc 0) by simp
     from f0(2)[rule-format, OF th0] show ?thesis using n0 by simp
    next
     case (Suc m)
     assume m: n = Suc m
     from fSuc(1)[of n] m have th0: f (Suc m) ≤ f (Suc (Suc m)) by simp
     from m fSuc(2)[rule-format, OF th0] show ?thesis by simp
    qed}
   thus monoseq (λn. s (f n)) unfolding monoseq-Suc by blast
  qed
  with th1 have ?thesis by blast}
 moreover
 {fix N assume N: ∀p >N. ∃ m≥p. s m > s p
  {fix p assume p: p ≥ Suc N
   hence pN: p > N by arith with N obtain m where m: m ≥ p s m > s p
by blast
   have m ≠ p using m(2) by auto
   with m have ∃m>p. s p < s m by − (rule exI[where x=m], auto)}
  note th0 = this
  let ?P = λm x. m > x ∧ s x < s m
  from nat-function-unique[of SOME x. ?P x (Suc N) λm x. SOME y. ?P y x]
  obtain f where f: f 0 = (SOME x. ?P x (Suc N))
   ∀n. f (Suc n) = (SOME m. ?P m (f n)) by blast
  have ?P (f 0) (Suc N) unfolding f(1) some-eq-ex[of λp. ?P p (Suc N)]
   using N apply −
   apply (erule allE[where x=Suc N], clarsimp)
   apply (rule-tac x=m in exI)
```

apply *auto*
apply (*subgoal-tac Suc N* $\neq$ *m*)
apply *simp*
apply (*rule ccontr*, *simp*)
done
hence *f0*: *f 0* > *Suc N s* (*Suc N*) < *s* (*f 0*) **by** *blast*+
{fix *n*
have *f n* > *N* $\wedge$ *?P* (*f* (*Suc n*)) (*f n*)
unfolding *f*(*2*)[*rule-format*, *of n*] *some-eq-ex*[*of* λ*p*. *?P p* (*f n*)]
proof (*induct n*)
case *0* **thus** *?case*
using *f0 N* **apply** *auto*
apply (*erule allE*[**where** *x*=*f 0*], *clarsimp*)
apply (*rule-tac x*=*m* **in** *exI*, *simp*)
by (*subgoal-tac f 0* $\neq$ *m*, *auto*)
next
case (*Suc n*)
from *Suc.hyps* **have** *Nfn*: *N* < *f n* **by** *blast*
from *Suc.hyps* **obtain** *m* **where** *m*: *m* > *f n s* (*f n*) < *s m* **by** *blast*
with *Nfn* **have** *mN*: *m* > *N* **by** *arith*
note *key* = *Suc.hyps*[*unfolded some-eq-ex*[*of* λ*p*. *?P p* (*f n*), *symmetric*] *f*(*2*)[*rule-format*, *of n*, *symmetric*]]

from *key* **have** *th0*: *f* (*Suc n*) > *N* **by** *simp*
from *N*[*rule-format*, *OF th0*]
obtain *m'* **where** *m'*: *m'* $\geq$ *f* (*Suc n*) *s* (*f* (*Suc n*)) < *s m'* **by** *blast*
have *m'* $\neq$ *f* (*Suc* (*n*)) **apply** (*rule ccontr*) **using** *m'*(*2*) **by** *auto*
hence *m'* > *f* (*Suc n*) **using** *m'*(*1*) **by** *simp*
with *key m'*(*2*) **show** *?case* **by** *auto*
qed}
note *fSuc* = *this*
{fix *n*
have *f n* $\geq$ *Suc N* $\wedge$ *f*(*Suc n*) > *f n* $\wedge$ *s*(*f n*) < *s*(*f*(*Suc n*)) **using** *fSuc*[*of n*] **by** *auto*
hence *f n* $\geq$ *Suc N f*(*Suc n*) > *f n s*(*f n*) < *s*(*f*(*Suc n*)) **by** *blast*+**}**
note *thf* = *this*
have *sqf*: *subseq f* **unfolding** *subseq-Suc-iff* **using** *thf* **by** *simp*
have *monoseq* (λ*n*. *s* (*f n*)) **unfolding** *monoseq-Suc* **using** *thf*
apply −
apply (*rule disjI1*)
apply *auto*
apply (*rule order-less-imp-le*)
apply *blast*
done
then have *?thesis* **using** *sqf* **by** *blast***}**
ultimately show *?thesis* **unfolding** *linorder-not-less*[*symmetric*] **by** *blast*
qed

lemma *seq-suble*: **assumes** *sf*: *subseq f* **shows** *n* $\leq$ *f n*

proof(*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
from *sf*[*unfolded subseq-Suc-iff*, *rule-format*, *of n*] *Suc.hyps*
have $n < f\ (Suc\ n)$ **by** *arith*
thus *?case* **by** *arith*
qed

lemma *LIMSEQ-subseq-LIMSEQ*:
$[\![X \longrightarrow L;\ subseq\ f\ ]\!] \Longrightarrow (X\ o\ f) \longrightarrow L$
apply (*auto simp add*: *LIMSEQ-def*)
apply (*drule-tac x=r* **in** *spec*, *clarify*)
apply (*rule-tac x=no* **in** *exI*, *clarify*)
apply (*blast intro*: *seq-suble le-trans dest*!: *spec*)
done

57.5 Bounded Monotonic Sequences

Bounded Sequence

lemma *BseqD*: $Bseq\ X \Longrightarrow \exists K.\ 0 < K\ \&\ (\forall n.\ norm\ (X\ n) \leq K)$
by (*simp add*: *Bseq-def*)

lemma *BseqI*: $[\!|\ 0 < K;\ \forall n.\ norm\ (X\ n) \leq K\ |\!] \Longrightarrow Bseq\ X$
by (*auto simp add*: *Bseq-def*)

lemma *lemma-NBseq-def*:
$(\exists K > 0.\ \forall n.\ norm\ (X\ n) \leq K) =$
$(\exists N.\ \forall n.\ norm\ (X\ n) \leq real(Suc\ N))$
apply *auto*
prefer *2* **apply** *force*
apply (*cut-tac x = K* **in** *reals-Archimedean2*, *clarify*)
apply (*rule-tac x = n* **in** *exI*, *clarify*)
apply (*drule-tac x = na* **in** *spec*)
apply (*auto simp add*: *real-of-nat-Suc*)
done

alternative definition for Bseq

lemma *Bseq-iff*: $Bseq\ X = (\exists N.\ \forall n.\ norm\ (X\ n) \leq real(Suc\ N))$
apply (*simp add*: *Bseq-def*)
apply (*simp* (*no-asm*) *add*: *lemma-NBseq-def*)
done

lemma *lemma-NBseq-def2*:
$(\exists K > 0.\ \forall n.\ norm\ (X\ n) \leq K) = (\exists N.\ \forall n.\ norm\ (X\ n) < real(Suc\ N))$
apply (*subst lemma-NBseq-def*, *auto*)
apply (*rule-tac x = Suc N* **in** *exI*)
apply (*rule-tac* [*2*] *x = N* **in** *exI*)
apply (*auto simp add*: *real-of-nat-Suc*)

prefer *2* **apply** (*blast intro*: *order-less-imp-le*)
apply (*drule-tac* $x = n$ **in** *spec*, *simp*)
done

lemma *Bseq-iff1a*: *Bseq* $X = (\exists N.\ \forall n.\ norm\ (X\ n) < real(Suc\ N))$
by (*simp add*: *Bseq-def lemma-NBseq-def2*)

57.5.1 Upper Bounds and Lubs of Bounded Sequences

lemma *Bseq-isUb*:
$!!(X::nat=>real).\ Bseq\ X\ ==>\ \exists U.\ isUb\ (UNIV::real\ set)\ \{x.\ \exists n.\ X\ n = x\}\ U$
by (*auto intro*: *isUbI setleI simp add*: *Bseq-def abs-le-iff*)

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *Bseq-isLub*:
$!!(X::nat=>real).\ Bseq\ X\ ==>$
$\exists U.\ isLub\ (UNIV::real\ set)\ \{x.\ \exists n.\ X\ n = x\}\ U$
by (*blast intro*: *reals-complete Bseq-isUb*)

57.5.2 A Bounded and Monotonic Sequence Converges

lemma *lemma-converg1*:
$!!(X::nat=>real).\ [|\ \forall m.\ \forall\ n \geq m.\ X\ m \leq X\ n;$
$isLub\ (UNIV::real\ set)\ \{x.\ \exists n.\ X\ n = x\}\ (X\ ma)$
$|]\ ==>\ \forall n \geq ma.\ X\ n = X\ ma$
apply *safe*
apply (*drule-tac* $y = X\ n$ **in** *isLubD2*)
apply (*blast dest*: *order-antisym*)+
done

The best of both worlds: Easier to prove this result as a standard theorem and then use equivalence to "transfer" it into the equivalent nonstandard form if needed!

lemma *Bmonoseq-LIMSEQ*: $\forall n.\ m \leq n\ --> X\ n = X\ m\ ==>\ \exists L.\ (X\ ----> L)$
apply (*simp add*: *LIMSEQ-def*)
apply (*rule-tac* $x = X\ m$ **in** *exI*, *safe*)
apply (*rule-tac* $x = m$ **in** *exI*, *safe*)
apply (*drule spec*, *erule impE*, *auto*)
done

lemma *lemma-converg2*:
$!!(X::nat=>real).$
$[|\ \forall m.\ X\ m \sim= U;\ \ isLub\ UNIV\ \{x.\ \exists n.\ X\ n = x\}\ U\ |]\ ==>\ \forall m.\ X\ m < U$
apply *safe*
apply (*drule-tac* $y = X\ m$ **in** *isLubD2*)
apply (*auto dest!*: *order-le-imp-less-or-eq*)

done

lemma *lemma-converg3*: !!(X ::*nat*=>*real*). $\forall m.\ X\ m \leq U$ ==> *isUb UNIV* $\{x.\ \exists n.\ X\ n = x\}\ U$
by (*rule setleI* [*THEN isUbI*], *auto*)

FIXME: $U - T < U$ is redundant

lemma *lemma-converg4*: !!(X::*nat*=> *real*).
[| $\forall m.\ X\ m \sim= U$;
isLub UNIV $\{x.\ \exists n.\ X\ n = x\}\ U$;
$0 < T$;
$U + -\ T < U$
|] ==> $\exists m.\ U + -T < X\ m\ \&\ X\ m < U$
apply (*drule lemma-converg2*, *assumption*)
apply (*rule ccontr*, *simp*)
apply (*simp add*: *linorder-not-less*)
apply (*drule lemma-converg3*)
apply (*drule isLub-le-isUb*, *assumption*)
apply (*auto dest*: *order-less-le-trans*)
done

A standard proof of the theorem for monotone increasing sequence

lemma *Bseq-mono-convergent*:
[| *Bseq* X; $\forall m.\ \forall n \geq m.\ X\ m \leq X\ n$ |] ==> *convergent* (X::*nat*=>*real*)
apply (*simp add*: *convergent-def*)
apply (*frule Bseq-isLub*, *safe*)
apply (*case-tac* $\exists m.\ X\ m = U$, *auto*)
apply (*blast dest*: *lemma-converg1 Bmonoseq-LIMSEQ*)

apply (*rule-tac* $x = U$ **in** *exI*)
apply (*subst LIMSEQ-iff*, *safe*)
apply (*frule lemma-converg2*, *assumption*)
apply (*drule lemma-converg4*, *auto*)
apply (*rule-tac* $x = m$ **in** *exI*, *safe*)
apply (*subgoal-tac* $X\ m \leq X\ n$)
prefer *2* **apply** *blast*
apply (*drule-tac* x=n **and** P=%$m.\ X\ m < U$ **in** *spec*, *arith*)
done

lemma *Bseq-minus-iff*: *Bseq* (%$n.\ -(X\ n)$) = *Bseq* X
by (*simp add*: *Bseq-def*)

Main monotonicity theorem

lemma *Bseq-monoseq-convergent*: [| *Bseq* X; *monoseq* X |] ==> *convergent* X
apply (*simp add*: *monoseq-def*, *safe*)
apply (*rule-tac* [*2*] *convergent-minus-iff* [*THEN ssubst*])
apply (*drule-tac* [*2*] *Bseq-minus-iff* [*THEN ssubst*])
apply (*auto intro*!: *Bseq-mono-convergent*)
done

57.5.3 Increasing and Decreasing Series

lemma *incseq-imp-monoseq*: *incseq X* $\Longrightarrow$ *monoseq X*
by (*simp add*: *incseq-def monoseq-def*)

lemma *incseq-le*: **assumes** *inc*: *incseq X* **and** *lim*: *X* $----> L$ **shows** $X\ n \leq L$
using *monoseq-le* [*OF incseq-imp-monoseq* [*OF inc*] *lim*]
proof
assume $(\forall n.\ X\ n \leq L) \wedge (\forall m\ n.\ m \leq n \longrightarrow X\ m \leq X\ n)$
thus *?thesis* **by** *simp*
next
assume $(\forall n.\ L \leq X\ n) \wedge (\forall m\ n.\ m \leq n \longrightarrow X\ n \leq X\ m)$
hence *const*: $(!!m\ n.\ m \leq n \Longrightarrow X\ n = X\ m)$ **using** *inc*
by (*auto simp add*: *incseq-def intro*: *order-antisym*)
have *X*: $!!n.\ X\ n = X\ 0$
by (*blast intro*: *const* [*of 0*])
have $X = (\lambda n.\ X\ 0)$
by (*blast intro*: *ext X*)
hence $L = X\ 0$ **using** *LIMSEQ-const* [*of X 0*]
by (*auto intro*: *LIMSEQ-unique lim*)
thus *?thesis*
by (*blast intro*: *eq-refl X*)
qed

lemma *decseq-imp-monoseq*: *decseq X* $\Longrightarrow$ *monoseq X*
by (*simp add*: *decseq-def monoseq-def*)

lemma *decseq-eq-incseq*: *decseq X* $=$ *incseq* $(\lambda n.\ -\ X\ n)$
by (*simp add*: *decseq-def incseq-def*)

lemma *decseq-le*: **assumes** *dec*: *decseq X* **and** *lim*: *X* $----> L$ **shows** $L \leq X\ n$
proof $-$
have *inc*: *incseq* $(\lambda n.\ -\ X\ n)$ **using** *dec*
by (*simp add*: *decseq-eq-incseq*)
have $-\ X\ n \leq -\ L$
by (*blast intro*: *incseq-le* [*OF inc*] *LIMSEQ-minus lim*)
thus *?thesis*
by *simp*
qed

57.5.4 A Few More Equivalence Theorems for Boundedness

alternative formulation for boundedness

lemma *Bseq-iff2*: *Bseq X* $= (\exists k > 0.\ \exists x.\ \forall n.\ norm\ (X(n) + -x) \leq k)$
apply (*unfold Bseq-def*, *safe*)
apply (*rule-tac* [*2*] $x = k + norm\ x$ **in** *exI*)

apply (*rule-tac* $x = K$ **in** *exI*, *simp*)
apply (*rule exI* [**where** $x = 0$], *auto*)
apply (*erule order-less-le-trans*, *simp*)
apply (*drule-tac* $x{=}n$ **in** *spec*, *fold diff-def*)
apply (*drule order-trans* [*OF norm-triangle-ineq2*])
apply *simp*
done

alternative formulation for boundedness

lemma *Bseq-iff3*: $Bseq\ X = (\exists k > 0.\ \exists N.\ \forall n.\ norm(X(n) + -X(N)) \leq k)$
apply *safe*
apply (*simp add*: *Bseq-def*, *safe*)
apply (*rule-tac* $x = K + norm\ (X\ N)$ **in** *exI*)
apply *auto*
apply (*erule order-less-le-trans*, *simp*)
apply (*rule-tac* $x = N$ **in** *exI*, *safe*)
apply (*drule-tac* $x = n$ **in** *spec*)
apply (*rule order-trans* [*OF norm-triangle-ineq*], *simp*)
apply (*auto simp add*: *Bseq-iff2*)
done

lemma *BseqI2*: $(\forall n.\ k \leq f\ n\ \&\ f\ n \leq (K::real)) ==> Bseq\ f$
apply (*simp add*: *Bseq-def*)
apply (*rule-tac* $x = (|k| + |K|) + 1$ **in** *exI*, *auto*)
apply (*drule-tac* $x = n$ **in** *spec*, *arith*)
done

57.6 Cauchy Sequences

lemma *CauchyI*:
$(\bigwedge e.\ 0 < e \Longrightarrow \exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ norm\ (X\ m - X\ n) < e) \Longrightarrow Cauchy\ X$
by (*simp add*: *Cauchy-def*)

lemma *CauchyD*:
$[\![Cauchy\ X;\ 0 < e]\!] \Longrightarrow \exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ norm\ (X\ m - X\ n) < e$
by (*simp add*: *Cauchy-def*)

lemma *Cauchy-subseq-Cauchy*:
$[\![\ Cauchy\ X;\ subseq\ f\ ]\!] \Longrightarrow Cauchy\ (X\ o\ f)$
apply (*auto simp add*: *Cauchy-def*)
apply (*drule-tac* $x{=}e$ **in** *spec*, *clarify*)
apply (*rule-tac* $x{=}M$ **in** *exI*, *clarify*)
apply (*blast intro*: *seq-suble le-trans dest*!: *spec*)
done

57.6.1 Cauchy Sequences are Bounded

A Cauchy sequence is bounded – this is the standard proof mechanization rather than the nonstandard proof

lemma *lemmaCauchy*: $\forall n \geq M.\ norm\ (X\ M - X\ n) < (1::real)$
$\quad ==>\ \forall n \geq M.\ norm\ (X\ n :: 'a::real\text{-}normed\text{-}vector) < 1 + norm\ (X\ M)$
apply (*clarify*, *drule spec*, *drule* (*1*) *mp*)
apply (*simp only*: *norm-minus-commute*)
apply (*drule order-le-less-trans* [*OF norm-triangle-ineq2*])
apply *simp*
done

lemma *Cauchy-Bseq*: *Cauchy X ==> Bseq X*
apply (*simp add*: *Cauchy-def*)
apply (*drule spec*, *drule mp*, *rule zero-less-one*, *safe*)
apply (*drule-tac x=M* **in** *spec*, *simp*)
apply (*drule lemmaCauchy*)
apply (*rule-tac k=M* **in** *Bseq-offset*)
apply (*simp add*: *Bseq-def*)
apply (*rule-tac x=1 + norm* (*X M*) **in** *exI*)
apply (*rule conjI*, *rule order-less-le-trans* [*OF zero-less-one*], *simp*)
apply (*simp add*: *order-less-imp-le*)
done

57.6.2 Cauchy Sequences are Convergent

axclass *banach* $\subseteq$ *real-normed-vector*
Cauchy-convergent: *Cauchy X* $\Longrightarrow$ *convergent X*

theorem *LIMSEQ-imp-Cauchy*:
assumes *X*: *X* ----> *a* **shows** *Cauchy X*
proof (*rule CauchyI*)
fix *e*::*real* **assume** *0 < e*
hence *0 < e/2* **by** *simp*
with *X* **have** $\exists N.\ \forall n \geq N.\ norm\ (X\ n - a) < e/2$ **by** (*rule LIMSEQ-D*)
then obtain *N* **where** *N*: $\forall n \geq N.\ norm\ (X\ n - a) < e/2$ **..**
show $\exists N.\ \forall m \geq N.\ \forall n \geq N.\ norm\ (X\ m - X\ n) < e$
proof (*intro exI allI impI*)
fix *m* **assume** $N \leq m$
hence *m*: *norm* $(X\ m - a) < e/2$ **using** *N* **by** *fast*
fix *n* **assume** $N \leq n$
hence *n*: *norm* $(X\ n - a) < e/2$ **using** *N* **by** *fast*
have *norm* $(X\ m - X\ n) = norm\ ((X\ m - a) - (X\ n - a))$ **by** *simp*
also have $\ldots \leq norm\ (X\ m - a) + norm\ (X\ n - a)$
by (*rule norm-triangle-ineq4*)
also from *m n* **have** $\ldots < e$ **by**(*simp add*:*field-simps*)
finally show *norm* $(X\ m - X\ n) < e$ **.**
qed
qed

lemma *convergent-Cauchy*: *convergent X* $\Longrightarrow$ *Cauchy X*
unfolding *convergent-def*
by (*erule exE*, *erule LIMSEQ-imp-Cauchy*)

Proof that Cauchy sequences converge based on the one from http://pirate.shu.edu/ wachsmut/ira/nu

If sequence X is Cauchy, then its limit is the lub of $\{r.\ \exists N.\ \forall n{\geq}N.\ r < X\ n\}$

lemma *isUb-UNIV-I*: $(\bigwedge y.\ y \in S \Longrightarrow y \leq u) \Longrightarrow$ *isUb UNIV S u*
by (*simp add*: *isUbI setleI*)

lemma *real-abs-diff-less-iff*:
$(|x - a| < (r{::}real)) = (a - r < x \wedge x < a + r)$
by *auto*

locale *real-Cauchy* =
fixes X :: *nat* ⇒ *real*
assumes *X*: *Cauchy X*
fixes S :: *real set*
defines *S-def*: $S \equiv \{x{::}real.\ \exists N.\ \forall n{\geq}N.\ x < X\ n\}$

lemma *real-CauchyI*:
assumes *Cauchy X*
shows *real-Cauchy X*
proof qed (*fact assms*)

lemma (**in** *real-Cauchy*) *mem-S*: $\forall n{\geq}N.\ x < X\ n \Longrightarrow x \in S$
by (*unfold S-def*, *auto*)

lemma (**in** *real-Cauchy*) *bound-isUb*:
assumes *N*: $\forall n{\geq}N.\ X\ n < x$
shows *isUb UNIV S x*
proof (*rule isUb-UNIV-I*)
fix $y{::}real$ **assume** $y \in S$
hence $\exists M.\ \forall n{\geq}M.\ y < X\ n$
by (*simp add*: *S-def*)
then obtain *M* **where** $\forall n{\geq}M.\ y < X\ n$ **..**
hence $y < X\ (max\ M\ N)$ **by** *simp*
also have $\ldots < x$ **using** *N* **by** *simp*
finally show $y \leq x$
by (*rule order-less-imp-le*)
qed

lemma (**in** *real-Cauchy*) *isLub-ex*: $\exists u.$ *isLub UNIV S u*
proof (*rule reals-complete*)
obtain *N* **where** $\forall m{\geq}N.\ \forall n{\geq}N.\ norm\ (X\ m - X\ n) < 1$
using *CauchyD* [*OF X zero-less-one*] **by** *fast*
hence *N*: $\forall n{\geq}N.\ norm\ (X\ n - X\ N) < 1$ **by** *simp*
show $\exists x.\ x \in S$
proof
from *N* **have** $\forall n{\geq}N.\ X\ N - 1 < X\ n$
by (*simp add*: *real-abs-diff-less-iff*)
thus $X\ N - 1 \in S$ **by** (*rule mem-S*)

```
  qed
  show ∃ u. isUb UNIV S u
  proof
    from N have ∀ n≥N. X n < X N + 1
      by (simp add: real-abs-diff-less-iff)
    thus isUb UNIV S (X N + 1)
      by (rule bound-isUb)
  qed
qed

lemma (in real-Cauchy) isLub-imp-LIMSEQ:
  assumes x: isLub UNIV S x
  shows X ----> x
proof (rule LIMSEQ-I)
  fix r::real assume 0 < r
  hence r: 0 < r/2 by simp
  obtain N where ∀ n≥N. ∀ m≥N. norm (X n - X m) < r/2
    using CauchyD [OF X r] by fast
  hence ∀ n≥N. norm (X n - X N) < r/2 by simp
  hence N: ∀ n≥N. X N - r/2 < X n ∧ X n < X N + r/2
    by (simp only: real-norm-def real-abs-diff-less-iff)

  from N have ∀ n≥N. X N - r/2 < X n by fast
  hence X N - r/2 ∈ S by (rule mem-S)
  hence 1: X N - r/2 ≤ x using x isLub-isUb isUbD by fast

  from N have ∀ n≥N. X n < X N + r/2 by fast
  hence isUb UNIV S (X N + r/2) by (rule bound-isUb)
  hence 2: x ≤ X N + r/2 using x isLub-le-isUb by fast

  show ∃ N. ∀ n≥N. norm (X n - x) < r
  proof (intro exI allI impI)
    fix n assume n: N ≤ n
    from N n have X n < X N + r/2 and X N - r/2 < X n by simp+
    thus norm (X n - x) < r using 1 2
      by (simp add: real-abs-diff-less-iff)
  qed
qed

lemma (in real-Cauchy) LIMSEQ-ex: ∃ x. X ----> x
proof -
  obtain x where isLub UNIV S x
    using isLub-ex by fast
  hence X ----> x
    by (rule isLub-imp-LIMSEQ)
  thus ?thesis ..
qed

lemma real-Cauchy-convergent:
```

fixes $X :: nat \Rightarrow real$
shows $Cauchy\ X \Longrightarrow convergent\ X$
unfolding *convergent-def*
by (*rule real-Cauchy.LIMSEQ-ex*)
(*rule real-CauchyI*)

instance *real* :: *banach*
by *intro-classes* (*rule real-Cauchy-convergent*)

lemma *Cauchy-convergent-iff*:
fixes $X :: nat \Rightarrow 'a::banach$
shows $Cauchy\ X = convergent\ X$
by (*fast intro*: *Cauchy-convergent convergent-Cauchy*)

lemma *convergent-subseq-convergent*:
fixes $X :: nat \Rightarrow 'a::banach$
shows $[\![convergent\ X; subseq\ f]\!] \Longrightarrow convergent\ (X\ o\ f)$
by (*simp add*: *Cauchy-subseq-Cauchy Cauchy-convergent-iff* [*symmetric*])

57.7 Power Sequences

The sequence $x \hat{}\ n$ tends to 0 if $(0::'a) \leq x$ and $x < (1::'a)$. Proof will use (NS) Cauchy equivalence for convergence and also fact that bounded and monotonic sequence converges.

lemma *Bseq-realpow*: [| $0 \leq (x::real)$; $x \leq 1$ |] ==> *Bseq* (%*n*. $x \hat{}\ n$)
apply (*simp add*: *Bseq-def*)
apply (*rule-tac* $x = 1$ **in** *exI*)
apply (*simp add*: *power-abs*)
apply (*auto dest*: *power-mono*)
done

lemma *monoseq-realpow*: [| $0 \leq x$; $x \leq 1$ |] ==> *monoseq* (%*n*. $x \hat{}\ n$)
apply (*clarify intro*!: *mono-SucI2*)
apply (*cut-tac* $n = n$ **and** $N = Suc\ n$ **and** $a = x$ **in** *power-decreasing*, *auto*)
done

lemma *convergent-realpow*:
[| $0 \leq (x::real)$; $x \leq 1$ |] ==> *convergent* (%*n*. $x \hat{}\ n$)
by (*blast intro*!: *Bseq-monoseq-convergent Bseq-realpow monoseq-realpow*)

lemma *LIMSEQ-inverse-realpow-zero-lemma*:
fixes $x :: real$
assumes x: $0 \leq x$
shows $real\ n * x + 1 \leq (x + 1) \hat{}\ n$
apply (*induct n*)
apply *simp*
apply *simp*
apply (*rule order-trans*)
prefer *2*

apply (*erule mult-left-mono*)
apply (*rule add-increasing* [*OF x*], *simp*)
apply (*simp add*: *real-of-nat-Suc*)
apply (*simp add*: *ring-distribs*)
apply (*simp add*: *mult-nonneg-nonneg x*)
done

lemma *LIMSEQ-inverse-realpow-zero*:
1 < (x::real) ⟹ (λn. inverse (x ^ n)) ----> 0
proof (*rule LIMSEQ-inverse-zero* [*rule-format*])
fix *y* :: *real*
assume *x*: *1 < x*
hence *0 < x − 1* **by** *simp*
hence ∀ *y*. ∃ *N*::*nat*. *y < real N ∗ (x − 1)*
by (*rule reals-Archimedean3*)
hence ∃ *N*::*nat*. *y < real N ∗ (x − 1)* **..**
then obtain *N*::*nat* **where** *y < real N ∗ (x − 1)* **..**
also have *... ≤ real N ∗ (x − 1) + 1* **by** *simp*
also have *... ≤ (x − 1 + 1) ^ N*
by (*rule LIMSEQ-inverse-realpow-zero-lemma*, *cut-tac x*, *simp*)
also have *... = x ^ N* **by** *simp*
finally have *y < x ^ N* **.**
hence ∀ *n*≥*N*. *y < x ^ n*
apply *clarify*
apply (*erule order-less-le-trans*)
apply (*erule power-increasing*)
apply (*rule order-less-imp-le* [*OF x*])
done
thus ∃ *N*. ∀ *n*≥*N*. *y < x ^ n* **..**
qed

lemma *LIMSEQ-realpow-zero*:
⟦0 ≤ (x::real); x < 1⟧ ⟹ (λn. x ^ n) ----> 0
proof (*cases*)
assume *x = 0*
hence *(λn. x ^ Suc n) ----> 0* **by** (*simp add*: *LIMSEQ-const*)
thus *?thesis* **by** (*rule LIMSEQ-imp-Suc*)
next
assume *0 ≤ x* **and** *x ≠ 0*
hence *x0*: *0 < x* **by** *simp*
assume *x1*: *x < 1*
from *x0 x1* **have** *1 < inverse x*
by (*rule real-inverse-gt-one*)
hence *(λn. inverse (inverse x ^ n)) ----> 0*
by (*rule LIMSEQ-inverse-realpow-zero*)
thus *?thesis* **by** (*simp add*: *power-inverse*)
qed

lemma *LIMSEQ-power-zero*:

fixes x :: $'a$::{*real-normed-algebra-1*,*recpower*}
shows *norm* $x < 1 \Longrightarrow (\lambda n.\ x\ \hat{}\ n) \longrightarrow 0$
apply (*drule LIMSEQ-realpow-zero* [*OF norm-ge-zero*])
apply (*simp only*: *LIMSEQ-Zseq-iff*, *erule Zseq-le*)
apply (*simp add*: *power-abs norm-power-ineq*)
done

lemma *LIMSEQ-divide-realpow-zero*:
$1 < (x::real) \Longrightarrow (\%n.\ a\ /\ (x\ \hat{}\ n)) \longrightarrow 0$
apply (*cut-tac* $a = a$ **and** *x1* = *inverse* x **in**
LIMSEQ-mult [*OF LIMSEQ-const LIMSEQ-realpow-zero*])
apply (*auto simp add*: *divide-inverse power-inverse*)
apply (*simp add*: *inverse-eq-divide pos-divide-less-eq*)
done

Limit of $c\ \hat{}\ n$ for $|c| < (1::'a)$

lemma *LIMSEQ-rabs-realpow-zero*: $|c| < (1::real) \Longrightarrow (\%n.\ |c|\ \hat{}\ n) \longrightarrow 0$
by (*rule LIMSEQ-realpow-zero* [*OF abs-ge-zero*])

lemma *LIMSEQ-rabs-realpow-zero2*: $|c| < (1::real) \Longrightarrow (\%n.\ c\ \hat{}\ n) \longrightarrow 0$
apply (*rule LIMSEQ-rabs-zero* [*THEN iffD1*])
apply (*auto intro*: *LIMSEQ-rabs-realpow-zero simp add*: *power-abs*)
done

end

58 Series: Finite Summation and Infinite Series

theory *Series*
imports *SEQ*
begin

definition
sums :: (*nat* ⇒ $'a$::*real-normed-vector*) ⇒ $'a$ ⇒ *bool*
(**infixr** *sums 80*) **where**
f sums s = $(\%n.\ setsum\ f\ \{0..<n\}) \longrightarrow s$

definition
summable :: (*nat* ⇒ $'a$::*real-normed-vector*) ⇒ *bool* **where**
summable f = $(\exists s.\ f\ sums\ s)$

definition
suminf :: (*nat* ⇒ $'a$::*real-normed-vector*) ⇒ $'a$ **where**
suminf f = $(THE\ s.\ f\ sums\ s)$

syntax
-suminf :: *idt* ⇒ $'a$ ⇒ $'a$ ($\sum$ -. - [0, *10*] *10*)
translations

$\sum i.\ b == CONST\ suminf\ (\%i.\ b)$

lemma *sumr-diff-mult-const*:
$setsum\ f\ \{0..<n\} - (real\ n*r) = setsum\ (\%i.\ f\ i - r)\ \{0..<n::nat\}$
by (*simp add*: *diff-minus setsum-addf real-of-nat-def*)

lemma *real-setsum-nat-ivl-bounded*:
$(!!p.\ p < n \Longrightarrow f(p) \leq K)$
$\Longrightarrow setsum\ f\ \{0..<n::nat\} \leq real\ n * K$
using *setsum-bounded*[**where** $A = \{0..<n\}$]
by (*auto simp*:*real-of-nat-def*)

lemma *sumr-minus-one-realpow-zero* [*simp*]:
$(\sum i=0..<2*n.\ (-1)\ \hat{}\ Suc\ i) = (0::real)$
by (*induct n*, *auto*)

lemma *sumr-one-lb-realpow-zero* [*simp*]:
$(\sum n=Suc\ 0..<n.\ f(n) * (0::real)\ \hat{}\ n) = 0$
by (*rule setsum-0'*, *simp*)

lemma *sumr-group*:
$(\sum m=0..<n::nat.\ setsum\ f\ \{m * k\ ..<\ m*k + k\}) = setsum\ f\ \{0\ ..<\ n * k\}$
apply (*subgoal-tac* $k = 0 \mid 0 < k$, *auto*)
apply (*induct n*)
apply (*simp-all add*: *setsum-add-nat-ivl add-commute*)
done

lemma *sumr-offset3*:
$setsum\ f\ \{0::nat..<n+k\} = (\sum m=0..<n.\ f\ (m+k)) + setsum\ f\ \{0..<k\}$
apply (*subst setsum-shift-bounds-nat-ivl* [*symmetric*])
apply (*simp add*: *setsum-add-nat-ivl add-commute*)
done

lemma *sumr-offset*:
fixes $f :: nat \Rightarrow 'a::ab\text{-}group\text{-}add$
shows $(\sum m=0..<n.\ f(m+k)) = setsum\ f\ \{0..<n+k\} - setsum\ f\ \{0..<k\}$
by (*simp add*: *sumr-offset3*)

lemma *sumr-offset2*:
$\forall f.\ (\sum m=0..<n::nat.\ f(m+k)::real) = setsum\ f\ \{0..<n+k\} - setsum\ f\ \{0..<k\}$
by (*simp add*: *sumr-offset*)

lemma *sumr-offset4*:
$\forall n\ f.\ setsum\ f\ \{0::nat..<n+k\} = (\sum m=0..<n.\ f\ (m+k)::real) + setsum\ f\ \{0..<k\}$
by (*clarify*, *rule sumr-offset3*)

58.1 Infinite Sums, by the Properties of Limits

lemma *sums-summable*: *f sums l ==> summable f*
by (*simp add*: *sums-def summable-def*, *blast*)

lemma *summable-sums*: *summable f ==> f sums (suminf f)*
apply (*simp add*: *summable-def suminf-def sums-def*)
apply (*blast intro*: *theI LIMSEQ-unique*)
done

lemma *summable-sumr-LIMSEQ-suminf*:
summable f ==> (%n. setsum f {0..<n}) ----> (suminf f)
by (*rule summable-sums* [*unfolded sums-def*])

lemma *sums-unique*: *f sums s ==> (s = suminf f)*
apply (*frule sums-summable* [*THEN summable-sums*])
apply (*auto intro*!: *LIMSEQ-unique simp add*: *sums-def*)
done

lemma *sums-split-initial-segment*: *f sums s ==>*
(%n. f(n + k)) sums (s - (SUM i = 0..< k. f i))
apply (*unfold sums-def*)
apply (*simp add*: *sumr-offset*)
apply (*rule LIMSEQ-diff-const*)
apply (*rule LIMSEQ-ignore-initial-segment*)
apply *assumption*
done

lemma *summable-ignore-initial-segment*: *summable f ==>*
summable (%n. f(n + k))
apply (*unfold summable-def*)
apply (*auto intro*: *sums-split-initial-segment*)
done

lemma *suminf-minus-initial-segment*: *summable f ==>*
suminf f = s ==> suminf (%n. f(n + k)) = s - (SUM i = 0..< k. f i)
apply (*frule summable-ignore-initial-segment*)
apply (*rule sums-unique* [*THEN sym*])
apply (*frule summable-sums*)
apply (*rule sums-split-initial-segment*)
apply *auto*
done

lemma *suminf-split-initial-segment*: *summable f ==>*
suminf f = (SUM i = 0..< k. f i) + suminf (%n. f(n + k))
by (*auto simp add*: *suminf-minus-initial-segment*)

lemma *suminf-exist-split*: **fixes** *r* :: *real* **assumes** *0 < r* **and** *summable a*
shows $\exists\ N.\ \forall\ n \geq N.\ |\sum\ i.\ a\ (i + n)| < r$

proof −
from *LIMSEQ-D*[*OF summable-sumr-LIMSEQ-suminf*[*OF* ‹*summable a*›] ‹*0* < *r*›]
obtain *N* :: *nat* **where** ∀ *n* ≥ *N*. *norm* (*setsum a* {*0*..<*n*} − *suminf a*) < *r*
by *auto*
thus *?thesis* **unfolding** *suminf-minus-initial-segment*[*OF* ‹*summable a*› *refl*] *abs-minus-commute real-norm-def*
by *auto*
qed

lemma *sums-Suc*: **assumes** *sumSuc*: (λ *n*. *f* (*Suc n*)) *sums l* **shows** *f sums* (*l* + *f 0*)
proof −
from *sumSuc*[*unfolded sums-def*]
have (λ*i*. ∑ *n* = *Suc 0*..<*Suc i*. *f n*) −−−−> *l* **unfolding** *setsum-reindex*[*OF inj-Suc*] *image-Suc-atLeastLessThan*[*symmetric*] *comp-def* **.**
from *LIMSEQ-add-const*[*OF this*, **where** *b*=*f 0*]
have (λ*i*. ∑ *n* = *0*..<*Suc i*. *f n*) −−−−> *l* + *f 0* **unfolding** *add-commute setsum-head-upt-Suc*[*OF zero-less-Suc*] **.**
thus *?thesis* **unfolding** *sums-def* **by** (*rule LIMSEQ-imp-Suc*)
qed

lemma *series-zero*:
(∀ *m*. *n* ≤ *m* −−> *f*(*m*) = *0*) ==> *f sums* (*setsum f* {*0*..<*n*})
apply (*simp add*: *sums-def LIMSEQ-def diff-minus*[*symmetric*], *safe*)
apply (*rule-tac x* = *n* **in** *exI*)
apply (*clarsimp simp add*:*setsum-diff*[*symmetric*] *cong*:*setsum-ivl-cong*)
done

lemma *sums-zero*: (λ*n*. *0*) *sums 0*
unfolding *sums-def* **by** (*simp add*: *LIMSEQ-const*)

lemma *summable-zero*: *summable* (λ*n*. *0*)
by (*rule sums-zero* [*THEN sums-summable*])

lemma *suminf-zero*: *suminf* (λ*n*. *0*) = *0*
by (*rule sums-zero* [*THEN sums-unique*, *symmetric*])

lemma (**in** *bounded-linear*) *sums*:
(λ*n*. *X n*) *sums a* ⟹ (λ*n*. *f* (*X n*)) *sums* (*f a*)
unfolding *sums-def* **by** (*drule LIMSEQ*, *simp only*: *setsum*)

lemma (**in** *bounded-linear*) *summable*:
summable (λ*n*. *X n*) ⟹ *summable* (λ*n*. *f* (*X n*))
unfolding *summable-def* **by** (*auto intro*: *sums*)

lemma (**in** *bounded-linear*) *suminf*:
summable (λ*n*. *X n*) ⟹ *f* (∑ *n*. *X n*) = (∑ *n*. *f* (*X n*))
by (*intro sums-unique sums summable-sums*)

lemma *sums-mult*:
fixes $c :: 'a$::*real-normed-algebra*
shows f *sums* $a \Longrightarrow (\lambda n.\ c * f\ n)$ *sums* $(c * a)$
by (*rule mult-right.sums*)

lemma *summable-mult*:
fixes $c :: 'a$::*real-normed-algebra*
shows *summable* $f \Longrightarrow$ *summable* $(\%n.\ c * f\ n)$
by (*rule mult-right.summable*)

lemma *suminf-mult*:
fixes $c :: 'a$::*real-normed-algebra*
shows *summable* $f \Longrightarrow$ *suminf* $(\lambda n.\ c * f\ n) =$ $c *$ *suminf* f
by (*rule mult-right.suminf* [*symmetric*])

lemma *sums-mult2*:
fixes $c :: 'a$::*real-normed-algebra*
shows f *sums* $a \Longrightarrow (\lambda n.\ f\ n * c)$ *sums* $(a * c)$
by (*rule mult-left.sums*)

lemma *summable-mult2*:
fixes $c :: 'a$::*real-normed-algebra*
shows *summable* $f \Longrightarrow$ *summable* $(\lambda n.\ f\ n * c)$
by (*rule mult-left.summable*)

lemma *suminf-mult2*:
fixes $c :: 'a$::*real-normed-algebra*
shows *summable* $f \Longrightarrow$ *suminf* $f * c = (\sum n.\ f\ n * c)$
by (*rule mult-left.suminf*)

lemma *sums-divide*:
fixes $c :: 'a$::*real-normed-field*
shows f *sums* $a \Longrightarrow (\lambda n.\ f\ n\ /\ c)$ *sums* $(a\ /\ c)$
by (*rule divide.sums*)

lemma *summable-divide*:
fixes $c :: 'a$::*real-normed-field*
shows *summable* $f \Longrightarrow$ *summable* $(\lambda n.\ f\ n\ /\ c)$
by (*rule divide.summable*)

lemma *suminf-divide*:
fixes $c :: 'a$::*real-normed-field*
shows *summable* $f \Longrightarrow$ *suminf* $(\lambda n.\ f\ n\ /\ c) =$ *suminf* $f\ /\ c$
by (*rule divide.suminf* [*symmetric*])

lemma *sums-add*: $[\![X$ *sums* $a;\ Y$ *sums* $b]\!] \Longrightarrow (\lambda n.\ X\ n + Y\ n)$ *sums* $(a + b)$
unfolding *sums-def* **by** (*simp add*: *setsum-addf LIMSEQ-add*)

lemma *summable-add*: ⟦*summable X*; *summable Y*⟧ ⟹ *summable* (λ*n*. *X n* + *Y n*)
unfolding *summable-def* **by** (*auto intro*: *sums-add*)

lemma *suminf-add*:
⟦*summable X*; *summable Y*⟧ ⟹ *suminf X* + *suminf Y* = (∑ *n*. *X n* + *Y n*)
by (*intro sums-unique sums-add summable-sums*)

lemma *sums-diff*: ⟦*X sums a*; *Y sums b*⟧ ⟹ (λ*n*. *X n* − *Y n*) *sums* (*a* − *b*)
unfolding *sums-def* **by** (*simp add*: *setsum-subtractf LIMSEQ-diff*)

lemma *summable-diff*: ⟦*summable X*; *summable Y*⟧ ⟹ *summable* (λ*n*. *X n* − *Y n*)
unfolding *summable-def* **by** (*auto intro*: *sums-diff*)

lemma *suminf-diff*:
⟦*summable X*; *summable Y*⟧ ⟹ *suminf X* − *suminf Y* = (∑ *n*. *X n* − *Y n*)
by (*intro sums-unique sums-diff summable-sums*)

lemma *sums-minus*: *X sums a* ==> (λ*n*. − *X n*) *sums* (− *a*)
unfolding *sums-def* **by** (*simp add*: *setsum-negf LIMSEQ-minus*)

lemma *summable-minus*: *summable X* ⟹ *summable* (λ*n*. − *X n*)
unfolding *summable-def* **by** (*auto intro*: *sums-minus*)

lemma *suminf-minus*: *summable X* ⟹ (∑ *n*. − *X n*) = − (∑ *n*. *X n*)
by (*intro sums-unique* [*symmetric*] *sums-minus summable-sums*)

lemma *sums-group*:
[|*summable f*; *0* < *k* |] ==> (%*n*. *setsum f* {*n*∗*k*..<*n*∗*k*+*k*}) *sums* (*suminf f*)
apply (*drule summable-sums*)
apply (*simp only*: *sums-def sumr-group*)
apply (*unfold LIMSEQ-def*, *safe*)
apply (*drule-tac x*=*r* **in** *spec*, *safe*)
apply (*rule-tac x*=*no* **in** *exI*, *safe*)
apply (*drule-tac x*=*n*∗*k* **in** *spec*)
apply (*erule mp*)
apply (*erule order-trans*)
apply *simp*
done

A summable series of positive terms has limit that is at least as great as any partial sum.

lemma *series-pos-le*:
fixes *f* :: *nat* ⇒ *real*
shows ⟦*summable f*; ∀ *m*≥*n*. *0* ≤ *f m*⟧ ⟹ *setsum f* {*0*..<*n*} ≤ *suminf f*
apply (*drule summable-sums*)
apply (*simp add*: *sums-def*)
apply (*cut-tac k* = *setsum f* {*0*..<*n*} **in** *LIMSEQ-const*)

apply (*erule LIMSEQ-le*, *blast*)
apply (*rule-tac x=n* **in** *exI*, *clarify*)
apply (*rule setsum-mono2*)
apply *auto*
done

lemma *series-pos-less*:
  fixes *f* :: *nat* ⇒ *real*
  shows ⟦*summable f*; ∀ *m≥n*. *0* < *f m*⟧ ⟹ *setsum f* {*0..<n*} < *suminf f*
apply (*rule-tac y=setsum f* {*0..<Suc n*} **in** *order-less-le-trans*)
apply *simp*
apply (*erule series-pos-le*)
apply (*simp add*: *order-less-imp-le*)
done

lemma *suminf-gt-zero*:
  fixes *f* :: *nat* ⇒ *real*
  shows ⟦*summable f*; ∀ *n*. *0* < *f n*⟧ ⟹ *0* < *suminf f*
by (*drule-tac n=0* **in** *series-pos-less*, *simp-all*)

lemma *suminf-ge-zero*:
  fixes *f* :: *nat* ⇒ *real*
  shows ⟦*summable f*; ∀ *n*. *0* ≤ *f n*⟧ ⟹ *0* ≤ *suminf f*
by (*drule-tac n=0* **in** *series-pos-le*, *simp-all*)

lemma *sumr-pos-lt-pair*:
  fixes *f* :: *nat* ⇒ *real*
  shows ⟦*summable f*;
       ∀ *d*. *0* < *f* (*k* + (*Suc*(*Suc 0*) ∗ *d*)) + *f* (*k* + ((*Suc*(*Suc 0*) ∗ *d*) + *1*))⟧
     ⟹ *setsum f* {*0..<k*} < *suminf f*
unfolding *One-nat-def*
apply (*subst suminf-split-initial-segment* [**where** *k=k*])
apply *assumption*
apply *simp*
apply (*drule-tac k=k* **in** *summable-ignore-initial-segment*)
apply (*drule-tac k=Suc* (*Suc 0*) **in** *sums-group*, *simp*)
apply *simp*
apply (*frule sums-unique*)
apply (*drule sums-summable*)
apply *simp*
apply (*erule suminf-gt-zero*)
apply (*simp add*: *add-ac*)
done

Sum of a geometric progression.

lemmas *sumr-geometric* = *geometric-sum* [**where** *'a* = *real*]

lemma *geometric-sums*:
  fixes *x* :: *'a*::{*real-normed-field*,*recpower*}

shows *norm x < 1 ⟹ (λn. x ^ n) sums (1 / (1 − x))*
proof −
assume *less-1*: *norm x < 1*
hence *neq-1*: *x ≠ 1* **by** *auto*
hence *neq-0*: *x − 1 ≠ 0* **by** *simp*
from *less-1* **have** *lim-0*: *(λn. x ^ n) −−−−> 0*
by (*rule LIMSEQ-power-zero*)
hence *(λn. x ^ n / (x − 1) − 1 / (x − 1)) −−−−> 0 / (x − 1) − 1 / (x − 1)*
using *neq-0* **by** (*intro LIMSEQ-divide LIMSEQ-diff LIMSEQ-const*)
hence *(λn. (x ^ n − 1) / (x − 1)) −−−−> 1 / (1 − x)*
by (*simp add*: *nonzero-minus-divide-right* [*OF neq-0*] *diff-divide-distrib*)
thus *(λn. x ^ n) sums (1 / (1 − x))*
by (*simp add*: *sums-def geometric-sum neq-1*)
qed

lemma *summable-geometric*:
fixes *x* :: *'a*::{*real-normed-field*,*recpower*}
shows *norm x < 1 ⟹ summable (λn. x ^ n)*
by (*rule geometric-sums* [*THEN sums-summable*])

Cauchy-type criterion for convergence of series (c.f. Harrison)

lemma *summable-convergent-sumr-iff*:
summable f = convergent (%n. setsum f {0..<n})
by (*simp add*: *summable-def sums-def convergent-def*)

lemma *summable-LIMSEQ-zero*: *summable f ⟹ f −−−−> 0*
apply (*drule summable-convergent-sumr-iff* [*THEN iffD1*])
apply (*drule convergent-Cauchy*)
apply (*simp only*: *Cauchy-def LIMSEQ-def*, *safe*)
apply (*drule-tac x=r* **in** *spec*, *safe*)
apply (*rule-tac x=M* **in** *exI*, *safe*)
apply (*drule-tac x=Suc n* **in** *spec*, *simp*)
apply (*drule-tac x=n* **in** *spec*, *simp*)
done

lemma *summable-Cauchy*:
summable (f::*nat ⇒ 'a*::*banach) =*
(∀ e > 0. ∃ N. ∀ m ≥ N. ∀ n. norm (setsum f {m..<n}) < e)
apply (*simp only*: *summable-convergent-sumr-iff Cauchy-convergent-iff* [*symmetric*] *Cauchy-def*, *safe*)
apply (*drule spec*, *drule* (*1*) *mp*)
apply (*erule exE*, *rule-tac x=M* **in** *exI*, *clarify*)
apply (*rule-tac x=m* **and** *y=n* **in** *linorder-le-cases*)
apply (*frule* (*1*) *order-trans*)
apply (*drule-tac x=n* **in** *spec*, *drule* (*1*) *mp*)
apply (*drule-tac x=m* **in** *spec*, *drule* (*1*) *mp*)
apply (*simp add*: *setsum-diff* [*symmetric*])
apply *simp*

apply (*drule spec*, *drule* (*1*) *mp*)
apply (*erule exE*, *rule-tac x=N* **in** *exI*, *clarify*)
apply (*rule-tac x=m* **and** *y=n* **in** *linorder-le-cases*)
apply (*subst norm-minus-commute*)
apply (*simp add*: *setsum-diff* [*symmetric*])
apply (*simp add*: *setsum-diff* [*symmetric*])
done

Comparison test

lemma *norm-setsum*:
  fixes f :: $'a \Rightarrow 'b$::*real-normed-vector*
  shows *norm* (*setsum f A*) $\leq$ ($\sum$ $i \in A$. *norm* (*f i*))
apply (*case-tac finite A*)
apply (*erule finite-induct*)
apply *simp*
apply *simp*
apply (*erule order-trans* [*OF norm-triangle-ineq add-left-mono*])
apply *simp*
done

lemma *summable-comparison-test*:
  fixes f :: *nat* $\Rightarrow$ $'a$::*banach*
  shows $[\![\exists N.\ \forall n \geq N.$ *norm* (*f n*) $\leq$ *g n*; *summable g*$]\!] \Longrightarrow$ *summable f*
apply (*simp add*: *summable-Cauchy*, *safe*)
apply (*drule-tac x=e* **in** *spec*, *safe*)
apply (*rule-tac x = N + Na* **in** *exI*, *safe*)
apply (*rotate-tac 2*)
apply (*drule-tac x = m* **in** *spec*)
apply (*auto*, *rotate-tac 2*, *drule-tac x = n* **in** *spec*)
apply (*rule-tac y* = $\sum k{=}m..{<}n$. *norm* (*f k*) **in** *order-le-less-trans*)
apply (*rule norm-setsum*)
apply (*rule-tac y = setsum g* {*m*..<*n*} **in** *order-le-less-trans*)
apply (*auto intro*: *setsum-mono simp add*: *abs-less-iff*)
done

lemma *summable-norm-comparison-test*:
  fixes f :: *nat* $\Rightarrow$ $'a$::*banach*
  shows $[\![\exists N.\ \forall n \geq N.$ *norm* (*f n*) $\leq$ *g n*; *summable g*$]\!]$
    $\Longrightarrow$ *summable* ($\lambda n.$ *norm* (*f n*))
apply (*rule summable-comparison-test*)
apply (*auto*)
done

lemma *summable-rabs-comparison-test*:
  fixes f :: *nat* $\Rightarrow$ *real*
  shows $[\![\exists N.\ \forall n \geq N.\ |f\ n| \leq g\ n$; *summable g*$]\!] \Longrightarrow$ *summable* ($\lambda n.\ |f\ n|$)
apply (*rule summable-comparison-test*)
apply (*auto*)
done

Summability of geometric series for real algebras

lemma *complete-algebra-summable-geometric*:
fixes $x :: 'a::\{real\text{-}normed\text{-}algebra\text{-}1, banach, recpower\}$
shows $norm\ x < 1 \Longrightarrow summable\ (\lambda n.\ x\ \hat{}\ n)$
proof (*rule summable-comparison-test*)
show $\exists N.\ \forall n \geq N.\ norm\ (x\ \hat{}\ n) \leq norm\ x\ \hat{}\ n$
by (*simp add*: *norm-power-ineq*)
show $norm\ x < 1 \Longrightarrow summable\ (\lambda n.\ norm\ x\ \hat{}\ n)$
by (*simp add*: *summable-geometric*)
qed

Limit comparison property for series (c.f. jrh)

lemma *summable-le*:
fixes $f\ g :: nat \Rightarrow real$
shows $[\![\forall n.\ f\ n \leq g\ n;\ summable\ f;\ summable\ g]\!] \Longrightarrow suminf\ f \leq suminf\ g$
apply (*drule summable-sums*)+
apply (*simp only*: *sums-def*, *erule* (*1*) *LIMSEQ-le*)
apply (*rule exI*)
apply (*auto intro*!: *setsum-mono*)
done

lemma *summable-le2*:
fixes $f\ g :: nat \Rightarrow real$
shows $[\![\forall n.\ |f\ n| \leq g\ n;\ summable\ g]\!] \Longrightarrow summable\ f \wedge suminf\ f \leq suminf\ g$
apply (*subgoal-tac summable f*)
apply (*auto intro*!: *summable-le*)
apply (*simp add*: *abs-le-iff*)
apply (*rule-tac g=g* **in** *summable-comparison-test*, *simp-all*)
done

lemma *suminf-0-le*:
fixes $f::nat \Rightarrow real$
assumes *gt0*: $\forall n.\ 0 \leq f\ n$ **and** *sm*: *summable f*
shows $0 \leq suminf\ f$
proof −
let $?g = (\lambda n.\ (0::real))$
from *gt0* **have** $\forall n.\ ?g\ n \leq f\ n$ **by** *simp*
moreover have *summable ?g* **by** (*rule summable-zero*)
moreover from *sm* **have** *summable f* **.**
ultimately have $suminf\ ?g \leq suminf\ f$ **by** (*rule summable-le*)
then show $0 \leq suminf\ f$ **by** (*simp add*: *suminf-zero*)
qed

Absolute convergence imples normal convergence

lemma *summable-norm-cancel*:
fixes $f :: nat \Rightarrow 'a::banach$
shows $summable\ (\lambda n.\ norm\ (f\ n)) \Longrightarrow summable\ f$
apply (*simp only*: *summable-Cauchy*, *safe*)

apply (*drule-tac x=e* **in** *spec*, *safe*)
apply (*rule-tac x=N* **in** *exI*, *safe*)
apply (*drule-tac x=m* **in** *spec*, *safe*)
apply (*rule order-le-less-trans* [*OF norm-setsum*])
apply (*rule order-le-less-trans* [*OF abs-ge-self*])
apply *simp*
done

lemma *summable-rabs-cancel*:
fixes f :: $nat \Rightarrow real$
shows *summable* $(\lambda n.\ |f\ n|) \Longrightarrow$ *summable* f
by (*rule summable-norm-cancel*, *simp*)

Absolute convergence of series

lemma *summable-norm*:
fixes f :: $nat \Rightarrow 'a$::*banach*
shows *summable* $(\lambda n.\ norm\ (f\ n)) \Longrightarrow norm\ (suminf\ f) \leq (\sum n.\ norm\ (f\ n))$
by (*auto intro*: *LIMSEQ-le LIMSEQ-norm summable-norm-cancel*
summable-sumr-LIMSEQ-suminf norm-setsum)

lemma *summable-rabs*:
fixes f :: $nat \Rightarrow real$
shows *summable* $(\lambda n.\ |f\ n|) \Longrightarrow |suminf\ f| \leq (\sum n.\ |f\ n|)$
by (*fold real-norm-def*, *rule summable-norm*)

58.2 The Ratio Test

lemma *norm-ratiotest-lemma*:
fixes $x\ y$ :: $'a$::*real-normed-vector*
shows $[\![c \leq 0;\ norm\ x \leq c * norm\ y]\!] \Longrightarrow x = 0$
apply (*subgoal-tac norm x* ≤ 0, *simp*)
apply (*erule order-trans*)
apply (*simp add*: *mult-le-0-iff*)
done

lemma *rabs-ratiotest-lemma*: $[|\ c \leq 0;\ abs\ x \leq c * abs\ y\ |] ==> x = (0$::*real*$)$
by (*erule norm-ratiotest-lemma*, *simp*)

lemma *le-Suc-ex*: $(k$::*nat*$) \leq l ==> (\exists n.\ l = k + n)$
apply (*drule le-imp-less-or-eq*)
apply (*auto dest*: *less-imp-Suc-add*)
done

lemma *le-Suc-ex-iff*: $((k$::*nat*$) \leq l) = (\exists n.\ l = k + n)$
by (*auto simp add*: *le-Suc-ex*)

lemma *ratio-test-lemma2*:
fixes f :: $nat \Rightarrow 'a$::*banach*

shows $[\![\forall n{\geq}N.\ norm\ (f\ (Suc\ n)) \leq c * norm\ (f\ n)]\!] \Longrightarrow 0 < c \vee summable\ f$
apply (*simp* (*no-asm*) *add*: *linorder-not-le* [*symmetric*])
apply (*simp add*: *summable-Cauchy*)
apply (*safe*, *subgoal-tac* $\forall n.\ N < n \longrightarrow f\ (n) = 0$)
prefer *2*
apply *clarify*
apply(*erule-tac* $x = n - Suc\ 0$ **in** *allE*)
apply (*simp add*:*diff-Suc split*:*nat.splits*)
apply (*blast intro*: *norm-ratiotest-lemma*)
apply (*rule-tac* $x = Suc\ N$ **in** *exI*, *clarify*)
apply(*simp cong*:*setsum-ivl-cong*)
done

lemma *ratio-test*:
fixes $f :: nat \Rightarrow 'a{::}banach$
shows $[\![c < 1; \forall n{\geq}N.\ norm\ (f\ (Suc\ n)) \leq c * norm\ (f\ n)]\!] \Longrightarrow summable\ f$
apply (*frule ratio-test-lemma2*, *auto*)
apply (*rule-tac* $g = \%n.\ (norm\ (f\ N)\ /\ (c\ \hat{}\ N)) * c\ \hat{}\ n$
in *summable-comparison-test*)
apply (*rule-tac* $x = N$ **in** *exI*, *safe*)
apply (*drule le-Suc-ex-iff* [*THEN iffD1*])
apply (*auto simp add*: *power-add field-power-not-zero*)
apply (*induct-tac na*, *auto*)
apply (*rule-tac* $y = c * norm\ (f\ (N + n))$ **in** *order-trans*)
apply (*auto intro*: *mult-right-mono simp add*: *summable-def*)
apply (*rule-tac* $x = norm\ (f\ N) * (1/\ (1 - c))\ /\ (c\ \hat{}\ N)$ **in** *exI*)
apply (*rule sums-divide*)
apply (*rule sums-mult*)
apply (*auto intro*!: *geometric-sums*)
done

58.3 Cauchy Product Formula

lemma *setsum-triangle-reindex*:
fixes $n :: nat$
shows $(\sum (i,j){\in}\{(i,j).\ i{+}j < n\}.\ f\ i\ j) = (\sum k{=}0..{<}n.\ \sum i{=}0..k.\ f\ i\ (k - i))$
proof −
have $(\sum (i,\ j){\in}\{(i,\ j).\ i + j < n\}.\ f\ i\ j) =$
$(\sum (k,\ i){\in}(SIGMA\ k{:}\{0..{<}n\}.\ \{0..k\}).\ f\ i\ (k - i))$
proof (*rule setsum-reindex-cong*)
show *inj-on* $(\lambda(k,i).\ (i,\ k - i))\ (SIGMA\ k{:}\{0..{<}n\}.\ \{0..k\})$
by (*rule inj-on-inverseI* [**where** $g{=}\lambda(i,j).\ (i{+}j,\ i)$], *auto*)
show $\{(i,j).\ i + j < n\} = (\lambda(k,i).\ (i,\ k - i))\ `\ (SIGMA\ k{:}\{0..{<}n\}.\ \{0..k\})$
by (*safe*, *rule-tac* $x{=}(a{+}b,a)$ **in** *image-eqI*, *auto*)
show $\bigwedge a.\ (\lambda(k,\ i).\ f\ i\ (k - i))\ a = split\ f\ ((\lambda(k,\ i).\ (i,\ k - i))\ a)$
by *clarify*
qed
thus *?thesis* **by** (*simp add*: *setsum-Sigma*)
qed

lemma *Cauchy-product-sums*:
fixes *a b* :: *nat* ⇒ *'a*::{*real-normed-algebra*,*banach*}
assumes *a*: *summable* (λ*k*. *norm* (*a k*))
assumes *b*: *summable* (λ*k*. *norm* (*b k*))
shows (λ*k*. ∑ *i*=*0*..*k*. *a i* ∗ *b* (*k* − *i*)) *sums* ((∑ *k*. *a k*) ∗ (∑ *k*. *b k*))
proof −
let *?S1* = λ*n*::*nat*. {*0*..<*n*} × {*0*..<*n*}
let *?S2* = λ*n*::*nat*. {(*i*,*j*). *i* + *j* < *n*}
have *S1-mono*: ⋀*m n*. *m* ≤ *n* ⟹ *?S1 m* ⊆ *?S1 n* **by** *auto*
have *S2-le-S1*: ⋀*n*. *?S2 n* ⊆ *?S1 n* **by** *auto*
have *S1-le-S2*: ⋀*n*. *?S1* (*n div 2*) ⊆ *?S2 n* **by** *auto*
have *finite-S1*: ⋀*n*. *finite* (*?S1 n*) **by** *simp*
with *S2-le-S1* **have** *finite-S2*: ⋀*n*. *finite* (*?S2 n*) **by** (*rule finite-subset*)

let *?g* = λ(*i*,*j*). *a i* ∗ *b j*
let *?f* = λ(*i*,*j*). *norm* (*a i*) ∗ *norm* (*b j*)
have *f-nonneg*: ⋀*x*. *0* ≤ *?f x*
by (*auto simp add*: *mult-nonneg-nonneg*)
hence *norm-setsum-f*: ⋀*A*. *norm* (*setsum ?f A*) = *setsum ?f A*
unfolding *real-norm-def*
by (*simp only*: *abs-of-nonneg setsum-nonneg* [*rule-format*])

have (λ*n*. (∑ *k*=*0*..<*n*. *a k*) ∗ (∑ *k*=*0*..<*n*. *b k*))
−−−−> (∑ *k*. *a k*) ∗ (∑ *k*. *b k*)
by (*intro LIMSEQ-mult summable-sumr-LIMSEQ-suminf*
summable-norm-cancel [*OF a*] *summable-norm-cancel* [*OF b*])
hence *1*: (λ*n*. *setsum ?g* (*?S1 n*)) −−−−> (∑ *k*. *a k*) ∗ (∑ *k*. *b k*)
by (*simp only*: *setsum-product setsum-Sigma* [*rule-format*]
finite-atLeastLessThan)

have (λ*n*. (∑ *k*=*0*..<*n*. *norm* (*a k*)) ∗ (∑ *k*=*0*..<*n*. *norm* (*b k*)))
−−−−> (∑ *k*. *norm* (*a k*)) ∗ (∑ *k*. *norm* (*b k*))
using *a b* **by** (*intro LIMSEQ-mult summable-sumr-LIMSEQ-suminf*)
hence (λ*n*. *setsum ?f* (*?S1 n*)) −−−−> (∑ *k*. *norm* (*a k*)) ∗ (∑ *k*. *norm* (*b k*))
by (*simp only*: *setsum-product setsum-Sigma* [*rule-format*]
finite-atLeastLessThan)
hence *convergent* (λ*n*. *setsum ?f* (*?S1 n*))
by (*rule convergentI*)
hence *Cauchy*: *Cauchy* (λ*n*. *setsum ?f* (*?S1 n*))
by (*rule convergent-Cauchy*)
have *Zseq* (λ*n*. *setsum ?f* (*?S1 n* − *?S2 n*))
proof (*rule ZseqI*, *simp only*: *norm-setsum-f*)
fix *r* :: *real*
assume *r*: *0* < *r*
from *CauchyD* [*OF Cauchy r*] **obtain** *N*
where ∀ *m*≥*N*. ∀ *n*≥*N*. *norm* (*setsum ?f* (*?S1 m*) − *setsum ?f* (*?S1 n*)) < *r*
..
hence ⋀*m n*. ⟦*N* ≤ *n*; *n* ≤ *m*⟧ ⟹ *norm* (*setsum ?f* (*?S1 m* − *?S1 n*)) < *r*

by (*simp only*: *setsum-diff finite-S1 S1-mono*)
hence *N*: $\bigwedge m\ n.\ [\![N \leq n;\ n \leq m]\!] \Longrightarrow$ *setsum ?f* (*?S1 m* − *?S1 n*) < *r*
by (*simp only*: *norm-setsum-f*)
show $\exists N.\ \forall n{\geq}N.$ *setsum ?f* (*?S1 n* − *?S2 n*) < *r*
proof (*intro exI allI impI*)
fix *n* **assume** *2* ∗ *N* ≤ *n*
hence *n*: *N* ≤ *n div 2* **by** *simp*
have *setsum ?f* (*?S1 n* − *?S2 n*) ≤ *setsum ?f* (*?S1 n* − *?S1* (*n div 2*))
by (*intro setsum-mono2 finite-Diff finite-S1 f-nonneg*
Diff-mono subset-refl S1-le-S2)
also have ... < *r*
using *n div-le-dividend* **by** (*rule N*)
finally show *setsum ?f* (*?S1 n* − *?S2 n*) < *r* **.**
qed
qed
hence *Zseq* (λ*n*. *setsum ?g* (*?S1 n* − *?S2 n*))
apply (*rule Zseq-le* [*rule-format*])
apply (*simp only*: *norm-setsum-f*)
apply (*rule order-trans* [*OF norm-setsum setsum-mono*])
apply (*auto simp add*: *norm-mult-ineq*)
done
hence *2*: (λ*n*. *setsum ?g* (*?S1 n*) − *setsum ?g* (*?S2 n*)) −−−−> *0*
by (*simp only*: *LIMSEQ-Zseq-iff setsum-diff finite-S1 S2-le-S1 diff-0-right*)

with *1* **have** (λ*n*. *setsum ?g* (*?S2 n*)) −−−−> (∑ *k*. *a k*) ∗ (∑ *k*. *b k*)
by (*rule LIMSEQ-diff-approach-zero2*)
thus *?thesis* **by** (*simp only*: *sums-def setsum-triangle-reindex*)
qed

lemma *Cauchy-product*:
fixes *a b* :: *nat* ⇒ *'a*::{*real-normed-algebra*,*banach*}
assumes *a*: *summable* (λ*k*. *norm* (*a k*))
assumes *b*: *summable* (λ*k*. *norm* (*b k*))
shows (∑ *k*. *a k*) ∗ (∑ *k*. *b k*) = (∑ *k*. ∑ *i*=*0*..*k*. *a i* ∗ *b* (*k* − *i*))
using *a b*
by (*rule Cauchy-product-sums* [*THEN sums-unique*])

end

59 Lim: Limits and Continuity

theory *Lim*
imports *SEQ*
begin

Standard Definitions

definition
LIM :: [*'a*::*real-normed-vector* => *'b*::*real-normed-vector*, *'a*, *'b*] => *bool*

(((-)/ −− (-)/ −−> (-)) [60, 0, 60] 60) **where**
[code del]: f −− a −−> L =
$(\forall r > 0.\ \exists s > 0.\ \forall x.\ x \neq a\ \&\ norm\ (x - a) < s$
−−> $norm\ (f\ x - L) < r)$

definition
isCont :: ['a::real-normed-vector => 'b::real-normed-vector, 'a] => *bool* **where**
isCont f a = (f −− a −−> ($f\ a$))

definition
isUCont :: ['a::real-normed-vector => 'b::real-normed-vector] => *bool* **where**
[code del]: *isUCont f* = ($\forall r{>}0.\ \exists s{>}0.\ \forall x\ y.\ norm\ (x - y) < s \longrightarrow norm\ (f\ x - f\ y) < r$)

59.1 Limits of Functions

59.1.1 Purely standard proofs

lemma *LIM-eq*:
f −− a −−> L =
$(\forall r{>}0.\exists s{>}0.\forall x.\ x \neq a\ \&\ norm\ (x{-}a) < s$ −−> $norm\ (f\ x - L) < r)$
by (*simp add*: *LIM-def diff-minus*)

lemma *LIM-I*:
$(!!r.\ 0{<}r$ ==> $\exists s{>}0.\forall x.\ x \neq a\ \&\ norm\ (x{-}a) < s$ −−> $norm\ (f\ x - L) < r)$
==> f −− a −−> L
by (*simp add*: *LIM-eq*)

lemma *LIM-D*:
[| f −− a −−> L; $0{<}r$ |]
==> $\exists s{>}0.\forall x.\ x \neq a\ \&\ norm\ (x{-}a) < s$ −−> $norm\ (f\ x - L) < r$
by (*simp add*: *LIM-eq*)

lemma *LIM-offset*: f −− a −−> $L \Longrightarrow (\lambda x.\ f\ (x + k))$ −− $a - k$ −−> L
apply (*rule LIM-I*)
apply (*drule-tac r=r* **in** *LIM-D*, *safe*)
apply (*rule-tac x=s* **in** *exI*, *safe*)
apply (*drule-tac x=x + k* **in** *spec*)
apply (*simp add*: *algebra-simps*)
done

lemma *LIM-offset-zero*: f −− a −−> $L \Longrightarrow (\lambda h.\ f\ (a + h))$ −− 0 −−> L
by (*drule-tac k=a* **in** *LIM-offset*, *simp add*: *add-commute*)

lemma *LIM-offset-zero-cancel*: $(\lambda h.\ f\ (a + h))$ −− 0 −−> $L \Longrightarrow f$ −− a −−> L
by (*drule-tac k=− a* **in** *LIM-offset*, *simp*)

lemma *LIM-const* [*simp*]: $(\%x.\ k)$ −− x −−> k

by (*simp add*: *LIM-def*)

lemma *LIM-add*:
fixes *f g* :: *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-vector*
assumes *f*: *f* −− *a* −−> *L* **and** *g*: *g* −− *a* −−> *M*
shows (%*x*. *f x* + *g*(*x*)) −− *a* −−> (*L* + *M*)
proof (*rule LIM-I*)
fix *r* :: *real*
assume *r*: *0* < *r*
from *LIM-D* [*OF f half-gt-zero* [*OF r*]]
obtain *fs*
where *fs*: *0* < *fs*
and *fs-lt*: ∀ *x*. *x* ≠ *a* & *norm* (*x*−*a*) < *fs* −−> *norm* (*f x* − *L*) < *r*/*2*
by *blast*
from *LIM-D* [*OF g half-gt-zero* [*OF r*]]
obtain *gs*
where *gs*: *0* < *gs*
and *gs-lt*: ∀ *x*. *x* ≠ *a* & *norm* (*x*−*a*) < *gs* −−> *norm* (*g x* − *M*) < *r*/*2*
by *blast*
show ∃ *s*>*0*.∀ *x*. *x* ≠ *a* ∧ *norm* (*x*−*a*) < *s* ⟶ *norm* (*f x* + *g x* − (*L* + *M*)) < *r*
proof (*intro exI conjI strip*)
show *0* < *min fs gs* **by** (*simp add*: *fs gs*)
fix *x* :: *'a*
assume *x* ≠ *a* ∧ *norm* (*x*−*a*) < *min fs gs*
hence *x* ≠ *a* ∧ *norm* (*x*−*a*) < *fs* ∧ *norm* (*x*−*a*) < *gs* **by** *simp*
with *fs-lt gs-lt*
have *norm* (*f x* − *L*) < *r*/*2* **and** *norm* (*g x* − *M*) < *r*/*2* **by** *blast*+
hence *norm* (*f x* − *L*) + *norm* (*g x* − *M*) < *r* **by** *arith*
thus *norm* (*f x* + *g x* − (*L* + *M*)) < *r*
by (*blast intro*: *norm-diff-triangle-ineq order-le-less-trans*)
qed
qed

lemma *LIM-add-zero*:
⟦*f* −− *a* −−> *0*; *g* −− *a* −−> *0*⟧ ⟹ (λ*x*. *f x* + *g x*) −− *a* −−> *0*
by (*drule* (*1*) *LIM-add*, *simp*)

lemma *minus-diff-minus*:
fixes *a b* :: *'a*::*ab-group-add*
shows (− *a*) − (− *b*) = − (*a* − *b*)
by *simp*

lemma *LIM-minus*: *f* −− *a* −−> *L* ==> (%*x*. −*f*(*x*)) −− *a* −−> −*L*
by (*simp only*: *LIM-eq minus-diff-minus norm-minus-cancel*)

lemma *LIM-add-minus*:
[| *f* −− *x* −−> *l*; *g* −− *x* −−> *m* |] ==> (%*x*. *f*(*x*) + −*g*(*x*)) −− *x* −−> (*l* + −*m*)

by (*intro LIM-add LIM-minus*)

lemma *LIM-diff*:
[| f −− x −−> l; g −− x −−> m |] ==> (%x. $f(x) - g(x)$) −− x −−> $l-m$
by (*simp only*: *diff-minus LIM-add LIM-minus*)

lemma *LIM-zero*: f −− a −−> $l \Longrightarrow (\lambda x.\ f\ x - l)$ −− a −−> 0
by (*simp add*: *LIM-def*)

lemma *LIM-zero-cancel*: $(\lambda x.\ f\ x - l)$ −− a −−> $0 \Longrightarrow f$ −− a −−> l
by (*simp add*: *LIM-def*)

lemma *LIM-zero-iff*: $(\lambda x.\ f\ x - l)$ −− a −−> $0 = f$ −− a −−> l
by (*simp add*: *LIM-def*)

lemma *LIM-imp-LIM*:
assumes *f*: f −− a −−> l
assumes *le*: $\bigwedge x.\ x \neq a \Longrightarrow norm\ (g\ x - m) \leq norm\ (f\ x - l)$
shows g −− a −−> m
apply (*rule LIM-I*, *drule LIM-D* [*OF f*], *safe*)
apply (*rule-tac x*=*s* **in** *exI*, *safe*)
apply (*drule-tac x*=*x* **in** *spec*, *safe*)
apply (*erule* (*1*) *order-le-less-trans* [*OF le*])
done

lemma *LIM-norm*: f −− a −−> $l \Longrightarrow (\lambda x.\ norm\ (f\ x))$ −− a −−> $norm\ l$
by (*erule LIM-imp-LIM*, *simp add*: *norm-triangle-ineq3*)

lemma *LIM-norm-zero*: f −− a −−> $0 \Longrightarrow (\lambda x.\ norm\ (f\ x))$ −− a −−> 0
by (*drule LIM-norm*, *simp*)

lemma *LIM-norm-zero-cancel*: $(\lambda x.\ norm\ (f\ x))$ −− a −−> $0 \Longrightarrow f$ −− a −−> 0
by (*erule LIM-imp-LIM*, *simp*)

lemma *LIM-norm-zero-iff*: $(\lambda x.\ norm\ (f\ x))$ −− a −−> $0 = f$ −− a −−> 0
by (*rule iffI* [*OF LIM-norm-zero-cancel LIM-norm-zero*])

lemma *LIM-rabs*: f −− a −−> $(l::real) \Longrightarrow (\lambda x.\ |f\ x|)$ −− a −−> $|l|$
by (*fold real-norm-def*, *rule LIM-norm*)

lemma *LIM-rabs-zero*: f −− a −−> $(0::real) \Longrightarrow (\lambda x.\ |f\ x|)$ −− a −−> 0
by (*fold real-norm-def*, *rule LIM-norm-zero*)

lemma *LIM-rabs-zero-cancel*: $(\lambda x.\ |f\ x|)$ −− a −−> $(0::real) \Longrightarrow f$ −− a −−> 0
by (*fold real-norm-def*, *rule LIM-norm-zero-cancel*)

lemma *LIM-rabs-zero-iff*: $(\lambda x.\ |f\ x|)$ −− a −−> $(0::real) = f$ −− a −−> 0

by (*fold real-norm-def*, *rule LIM-norm-zero-iff*)

lemma *LIM-const-not-eq*:
  fixes *a* :: *'a*::*real-normed-algebra-1*
  shows $k \neq L \Longrightarrow \neg\ (\lambda x.\ k) \dashrightarrow a \longrightarrow L$
apply (*simp add*: *LIM-eq*)
apply (*rule-tac x*=*norm* $(k - L)$ **in** *exI*, *simp*, *safe*)
apply (*rule-tac x*=*a* + *of-real* $(s/2)$ **in** *exI*, *simp add*: *norm-of-real*)
done

lemmas *LIM-not-zero* = *LIM-const-not-eq* [**where** $L = 0$]

lemma *LIM-const-eq*:
  fixes *a* :: *'a*::*real-normed-algebra-1*
  shows $(\lambda x.\ k) \dashrightarrow a \longrightarrow L \Longrightarrow k = L$
apply (*rule ccontr*)
apply (*blast dest*: *LIM-const-not-eq*)
done

lemma *LIM-unique*:
  fixes *a* :: *'a*::*real-normed-algebra-1*
  shows $[\![f \dashrightarrow a \longrightarrow L; f \dashrightarrow a \longrightarrow M]\!] \Longrightarrow L = M$
apply (*drule* (*1*) *LIM-diff*)
apply (*auto dest*!: *LIM-const-eq*)
done

lemma *LIM-ident* [*simp*]: $(\lambda x.\ x) \dashrightarrow a \longrightarrow a$
by (*auto simp add*: *LIM-def*)

Limits are equal for functions equal except at limit point

lemma *LIM-equal*:
    $[|\ \forall x.\ x \neq a \longrightarrow (f\ x = g\ x)\ |] ==> (f \dashrightarrow a \longrightarrow l) = (g \dashrightarrow a \longrightarrow l)$
by (*simp add*: *LIM-def*)

lemma *LIM-cong*:
  $[\![a = b; \bigwedge x.\ x \neq b \Longrightarrow f\ x = g\ x; l = m]\!]$
   $\Longrightarrow ((\lambda x.\ f\ x) \dashrightarrow a \longrightarrow l) = ((\lambda x.\ g\ x) \dashrightarrow b \longrightarrow m)$
by (*simp add*: *LIM-def*)

lemma *LIM-equal2*:
  assumes *1*: $0 < R$
  assumes *2*: $\bigwedge x.\ [\![x \neq a; norm\ (x - a) < R]\!] \Longrightarrow f\ x = g\ x$
  shows $g \dashrightarrow a \longrightarrow l \Longrightarrow f \dashrightarrow a \longrightarrow l$
apply (*unfold LIM-def*, *safe*)
apply (*drule-tac x*=*r* **in** *spec*, *safe*)
apply (*rule-tac x*=*min s R* **in** *exI*, *safe*)
apply (*simp add*: *1*)
apply (*simp add*: *2*)
done

Two uses in Hyperreal/Transcendental.ML

lemma *LIM-trans*:
[| (%x. f(x) + -g(x)) -- a --> 0; g -- a --> l |] ==> f -- a --> l
apply (*drule LIM-add*, *assumption*)
apply (*auto simp add*: *add-assoc*)
done

lemma *LIM-compose*:
assumes *g*: $g -- l --> g\ l$
assumes *f*: $f -- a --> l$
shows $(\lambda x.\ g\ (f\ x)) -- a --> g\ l$
proof (*rule LIM-I*)
fix *r*::*real* **assume** *r*: $0 < r$
obtain *s* **where** *s*: $0 < s$
and *less-r*: $\bigwedge y.\ [\![y \neq l;\ norm\ (y - l) < s]\!] \Longrightarrow norm\ (g\ y - g\ l) < r$
using *LIM-D* [*OF g r*] **by** *fast*
obtain *t* **where** *t*: $0 < t$
and *less-s*: $\bigwedge x.\ [\![x \neq a;\ norm\ (x - a) < t]\!] \Longrightarrow norm\ (f\ x - l) < s$
using *LIM-D* [*OF f s*] **by** *fast*

show $\exists\, t{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < t \longrightarrow norm\ (g\ (f\ x) - g\ l) < r$
proof (*rule exI*, *safe*)
show $0 < t$ **using** *t* **.**
next
fix *x* **assume** $x \neq a$ **and** $norm\ (x - a) < t$
hence $norm\ (f\ x - l) < s$ **by** (*rule less-s*)
thus $norm\ (g\ (f\ x) - g\ l) < r$
using *r less-r* **by** (*case-tac f x = l*, *simp-all*)
qed
qed

lemma *LIM-compose2*:
assumes *f*: $f -- a --> b$
assumes *g*: $g -- b --> c$
assumes *inj*: $\exists\, d{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < d \longrightarrow f\ x \neq b$
shows $(\lambda x.\ g\ (f\ x)) -- a --> c$
proof (*rule LIM-I*)
fix *r* :: *real*
assume *r*: $0 < r$
obtain *s* **where** *s*: $0 < s$
and *less-r*: $\bigwedge y.\ [\![y \neq b;\ norm\ (y - b) < s]\!] \Longrightarrow norm\ (g\ y - c) < r$
using *LIM-D* [*OF g r*] **by** *fast*
obtain *t* **where** *t*: $0 < t$
and *less-s*: $\bigwedge x.\ [\![x \neq a;\ norm\ (x - a) < t]\!] \Longrightarrow norm\ (f\ x - b) < s$
using *LIM-D* [*OF f s*] **by** *fast*
obtain *d* **where** *d*: $0 < d$
and *neq-b*: $\bigwedge x.\ [\![x \neq a;\ norm\ (x - a) < d]\!] \Longrightarrow f\ x \neq b$
using *inj* **by** *fast*

show $\exists t{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < t \longrightarrow norm\ (g\ (f\ x) - c) < r$
proof (*safe intro*!: *exI*)
show $0 < min\ d\ t$ **using** $d\ t$ **by** *simp*
next
fix x
assume $x \neq a$ **and** $norm\ (x - a) < min\ d\ t$
hence $f\ x \neq b$ **and** $norm\ (f\ x - b) < s$
using *neq-b less-s* **by** *simp-all*
thus $norm\ (g\ (f\ x) - c) < r$
by (*rule less-r*)
qed
qed

lemma *LIM-o*: $[\![g \mathrel{--} l \mathrel{--\!>} g\ l; f \mathrel{--} a \mathrel{--\!>} l]\!] \Longrightarrow (g \circ f) \mathrel{--} a \mathrel{--\!>} g\ l$
unfolding *o-def* **by** (*rule LIM-compose*)

lemma *real-LIM-sandwich-zero*:
fixes $f\ g :: {'a}$::*real-normed-vector* $\Rightarrow$ *real*
assumes *f*: $f \mathrel{--} a \mathrel{--\!>} 0$
assumes *1*: $\bigwedge x.\ x \neq a \Longrightarrow 0 \leq g\ x$
assumes *2*: $\bigwedge x.\ x \neq a \Longrightarrow g\ x \leq f\ x$
shows $g \mathrel{--} a \mathrel{--\!>} 0$
proof (*rule LIM-imp-LIM* [*OF f*])
fix x **assume** *x*: $x \neq a$
have $norm\ (g\ x - 0) = g\ x$ **by** (*simp add*: *1 x*)
also have $g\ x \leq f\ x$ **by** (*rule 2* [*OF x*])
also have $f\ x \leq |f\ x|$ **by** (*rule abs-ge-self*)
also have $|f\ x| = norm\ (f\ x - 0)$ **by** *simp*
finally show $norm\ (g\ x - 0) \leq norm\ (f\ x - 0)$ **.**
qed

Bounded Linear Operators

lemma (**in** *bounded-linear*) *cont*: $f \mathrel{--} a \mathrel{--\!>} f\ a$
proof (*rule LIM-I*)
fix *r*::*real* **assume** *r*: $0 < r$
obtain *K* **where** *K*: $0 < K$ **and** *norm-le*: $\bigwedge x.\ norm\ (f\ x) \leq norm\ x * K$
using *pos-bounded* **by** *fast*
show $\exists s{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < s \longrightarrow norm\ (f\ x - f\ a) < r$
proof (*rule exI*, *safe*)
from *r K* **show** $0 < r\ /\ K$ **by** (*rule divide-pos-pos*)
next
fix *x* **assume** *x*: $norm\ (x - a) < r\ /\ K$
have $norm\ (f\ x - f\ a) = norm\ (f\ (x - a))$ **by** (*simp only*: *diff*)
also have $\ldots \leq norm\ (x - a) * K$ **by** (*rule norm-le*)
also from *K x* **have** $\ldots < r$ **by** (*simp only*: *pos-less-divide-eq*)
finally show $norm\ (f\ x - f\ a) < r$ **.**
qed
qed

lemma (**in** *bounded-linear*) *LIM*:
  $g \mathrel{--} a \mathrel{--\!>} l \Longrightarrow (\lambda x.\ f\ (g\ x)) \mathrel{--} a \mathrel{--\!>} f\ l$
by (*rule LIM-compose* [*OF cont*])

lemma (**in** *bounded-linear*) *LIM-zero*:
  $g \mathrel{--} a \mathrel{--\!>} 0 \Longrightarrow (\lambda x.\ f\ (g\ x)) \mathrel{--} a \mathrel{--\!>} 0$
by (*drule LIM*, *simp only*: *zero*)

Bounded Bilinear Operators

lemma (**in** *bounded-bilinear*) *LIM-prod-zero*:
  assumes *f*: $f \mathrel{--} a \mathrel{--\!>} 0$
  assumes *g*: $g \mathrel{--} a \mathrel{--\!>} 0$
  shows $(\lambda x.\ f\ x\ **\ g\ x) \mathrel{--} a \mathrel{--\!>} 0$
proof (*rule LIM-I*)
  fix *r*::*real* **assume** *r*: $0 < r$
  obtain *K* **where** *K*: $0 < K$
    and *norm-le*: $\bigwedge x\ y.\ norm\ (x\ **\ y) \leq norm\ x * norm\ y * K$
    using *pos-bounded* **by** *fast*
  from *K* **have** *K'*: $0 < inverse\ K$
    by (*rule positive-imp-inverse-positive*)
  obtain *s* **where** *s*: $0 < s$
    and *norm-f*: $\bigwedge x.\ [\![x \neq a;\ norm\ (x - a) < s]\!] \Longrightarrow norm\ (f\ x) < r$
    using *LIM-D* [*OF f r*] **by** *auto*
  obtain *t* **where** *t*: $0 < t$
    and *norm-g*: $\bigwedge x.\ [\![x \neq a;\ norm\ (x - a) < t]\!] \Longrightarrow norm\ (g\ x) < inverse\ K$
    using *LIM-D* [*OF g K'*] **by** *auto*
  show $\exists\, s{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < s \longrightarrow norm\ (f\ x\ **\ g\ x - 0) < r$
  proof (*rule exI*, *safe*)
    from *s t* **show** $0 < min\ s\ t$ **by** *simp*
  next
    fix *x* **assume** *x*: $x \neq a$
    assume $norm\ (x - a) < min\ s\ t$
    hence *xs*: $norm\ (x - a) < s$ **and** *xt*: $norm\ (x - a) < t$ **by** *simp-all*
    from *x xs* **have** *1*: $norm\ (f\ x) < r$ **by** (*rule norm-f*)
    from *x xt* **have** *2*: $norm\ (g\ x) < inverse\ K$ **by** (*rule norm-g*)
    have $norm\ (f\ x\ **\ g\ x) \leq norm\ (f\ x) * norm\ (g\ x) * K$ **by** (*rule norm-le*)
    also from *1 2 K* **have** $\ldots < r * inverse\ K * K$
      by (*intro mult-strict-right-mono mult-strict-mono' norm-ge-zero*)
    also from *K* **have** $r * inverse\ K * K = r$ **by** *simp*
    finally show $norm\ (f\ x\ **\ g\ x - 0) < r$ **by** *simp*
  qed
qed

lemma (**in** *bounded-bilinear*) *LIM-left-zero*:
  $f \mathrel{--} a \mathrel{--\!>} 0 \Longrightarrow (\lambda x.\ f\ x\ **\ c) \mathrel{--} a \mathrel{--\!>} 0$
by (*rule bounded-linear.LIM-zero* [*OF bounded-linear-left*])

lemma (**in** *bounded-bilinear*) *LIM-right-zero*:
  $f \mathrel{--} a \mathrel{--\!>} 0 \Longrightarrow (\lambda x.\ c\ **\ f\ x) \mathrel{--} a \mathrel{--\!>} 0$

by (*rule bounded-linear.LIM-zero* [*OF bounded-linear-right*])

lemma (**in** *bounded-bilinear*) *LIM*:
  ⟦*f* –– *a* ––> *L*; *g* –– *a* ––> *M*⟧ ⟹ (λ*x*. *f x* ** *g x*) –– *a* ––> *L* ** *M*
apply (*drule LIM-zero*)
apply (*drule LIM-zero*)
apply (*rule LIM-zero-cancel*)
apply (*subst prod-diff-prod*)
apply (*rule LIM-add-zero*)
apply (*rule LIM-add-zero*)
apply (*erule* (*1*) *LIM-prod-zero*)
apply (*erule LIM-left-zero*)
apply (*erule LIM-right-zero*)
done

lemmas *LIM-mult* = *mult.LIM*

lemmas *LIM-mult-zero* = *mult.LIM-prod-zero*

lemmas *LIM-mult-left-zero* = *mult.LIM-left-zero*

lemmas *LIM-mult-right-zero* = *mult.LIM-right-zero*

lemmas *LIM-scaleR* = *scaleR.LIM*

lemmas *LIM-of-real* = *of-real.LIM*

lemma *LIM-power*:
  fixes *f* :: *'a*::*real-normed-vector* ⇒ *'b*::{*recpower*,*real-normed-algebra*}
  assumes *f*: *f* –– *a* ––> *l*
  shows (λ*x*. *f x* ^ *n*) –– *a* ––> *l* ^ *n*
by (*induct n*, *simp*, *simp add*: *LIM-mult f*)

59.1.2 Derived theorems about *LIM*

lemma *LIM-inverse-lemma*:
  fixes *x* :: *'a*::*real-normed-div-algebra*
  assumes *r*: *0* < *r*
  assumes *x*: *norm* (*x* − *1*) < *min* (*1*/*2*) (*r*/*2*)
  shows *norm* (*inverse x* − *1*) < *r*
proof −
  from *r* **have** *r2*: *0* < *r*/*2* **by** *simp*
  from *x* **have** *0*: *x* ≠ *0* **by** *clarsimp*
  from *x* **have** *x'*: *norm* (*1* − *x*) < *min* (*1*/*2*) (*r*/*2*)
    by (*simp only*: *norm-minus-commute*)
  hence *less1*: *norm* (*1* − *x*) < *r*/*2* **by** *simp*
  have *norm* (*1*::*'a*) − *norm x* ≤ *norm* (*1* − *x*) **by** (*rule norm-triangle-ineq2*)
  also from *x'* **have** *norm* (*1* − *x*) < *1*/*2* **by** *simp*
  finally have *1*/*2* < *norm x* **by** *simp*

hence *inverse* (*norm x*) < *inverse* (*1/2*)
by (*rule less-imp-inverse-less*, *simp*)
hence *less2*: *norm* (*inverse x*) < *2*
by (*simp add*: *nonzero-norm-inverse 0*)
from *less1 less2 r2 norm-ge-zero*
have *norm* (*1* − *x*) ∗ *norm* (*inverse x*) < (*r*/*2*) ∗ *2*
by (*rule mult-strict-mono*)
thus *norm* (*inverse x* − *1*) < *r*
by (*simp only*: *norm-mult* [*symmetric*] *left-diff-distrib*, *simp add*: *0*)
qed

lemma *LIM-inverse-fun*:
assumes *a*: *a* ≠ (*0*::'*a*::*real-normed-div-algebra*)
shows *inverse* −− *a* −−> *inverse a*
proof (*rule LIM-equal2*)
from *a* **show** *0* < *norm a* **by** *simp*
next
fix *x* **assume** *norm* (*x* − *a*) < *norm a*
hence *x* ≠ *0* **by** *auto*
with *a* **show** *inverse x* = *inverse* (*inverse a* ∗ *x*) ∗ *inverse a*
by (*simp add*: *nonzero-inverse-mult-distrib*
nonzero-imp-inverse-nonzero
nonzero-inverse-inverse-eq mult-assoc)
next
have *1*: *inverse* −− *1* −−> *inverse* (*1*::'*a*)
apply (*rule LIM-I*)
apply (*rule-tac x*=*min* (*1/2*) (*r*/*2*) **in** *exI*)
apply (*simp add*: *LIM-inverse-lemma*)
done
have (λ*x*. *inverse a* ∗ *x*) −− *a* −−> *inverse a* ∗ *a*
by (*intro LIM-mult LIM-ident LIM-const*)
hence (λ*x*. *inverse a* ∗ *x*) −− *a* −−> *1*
by (*simp add*: *a*)
with *1* **have** (λ*x*. *inverse* (*inverse a* ∗ *x*)) −− *a* −−> *inverse 1*
by (*rule LIM-compose*)
hence (λ*x*. *inverse* (*inverse a* ∗ *x*)) −− *a* −−> *1*
by *simp*
hence (λ*x*. *inverse* (*inverse a* ∗ *x*) ∗ *inverse a*) −− *a* −−> *1* ∗ *inverse a*
by (*intro LIM-mult LIM-const*)
thus (λ*x*. *inverse* (*inverse a* ∗ *x*) ∗ *inverse a*) −− *a* −−> *inverse a*
by *simp*
qed

lemma *LIM-inverse*:
fixes *L* :: '*a*::*real-normed-div-algebra*
shows ⟦*f* −− *a* −−> *L*; *L* ≠ *0*⟧ ⟹ (λ*x*. *inverse* (*f x*)) −− *a* −−> *inverse L*
by (*rule LIM-inverse-fun* [*THEN LIM-compose*])

lemma *LIM-sgn*:

⟦f −− a −−> l; l ≠ 0⟧ ⟹ (λx. sgn (f x)) −− a −−> sgn l
unfolding *sgn-div-norm*
by (*simp add*: *LIM-scaleR LIM-inverse LIM-norm*)

59.2 Continuity

59.2.1 Purely standard proofs

lemma *LIM-isCont-iff*: (f −− a −−> f a) = ((λh. f (a + h)) −− 0 −−> f a)
by (*rule iffI* [*OF LIM-offset-zero LIM-offset-zero-cancel*])

lemma *isCont-iff*: *isCont f x* = (λh. f (x + h)) −− 0 −−> f x
by (*simp add*: *isCont-def LIM-isCont-iff*)

lemma *isCont-ident* [*simp*]: *isCont* (λx. x) a
unfolding *isCont-def* **by** (*rule LIM-ident*)

lemma *isCont-const* [*simp*]: *isCont* (λx. k) a
unfolding *isCont-def* **by** (*rule LIM-const*)

lemma *isCont-norm*: *isCont f a* ⟹ *isCont* (λx. norm (f x)) a
unfolding *isCont-def* **by** (*rule LIM-norm*)

lemma *isCont-rabs*: *isCont f a* ⟹ *isCont* (λx. |f x :: *real*|) a
unfolding *isCont-def* **by** (*rule LIM-rabs*)

lemma *isCont-add*: ⟦*isCont f a*; *isCont g a*⟧ ⟹ *isCont* (λx. f x + g x) a
unfolding *isCont-def* **by** (*rule LIM-add*)

lemma *isCont-minus*: *isCont f a* ⟹ *isCont* (λx. − f x) a
unfolding *isCont-def* **by** (*rule LIM-minus*)

lemma *isCont-diff*: ⟦*isCont f a*; *isCont g a*⟧ ⟹ *isCont* (λx. f x − g x) a
unfolding *isCont-def* **by** (*rule LIM-diff*)

lemma *isCont-mult*:
fixes f g :: 'a::*real-normed-vector* ⇒ 'b::*real-normed-algebra*
shows ⟦*isCont f a*; *isCont g a*⟧ ⟹ *isCont* (λx. f x * g x) a
unfolding *isCont-def* **by** (*rule LIM-mult*)

lemma *isCont-inverse*:
fixes f :: 'a::*real-normed-vector* ⇒ 'b::*real-normed-div-algebra*
shows ⟦*isCont f a*; f a ≠ 0⟧ ⟹ *isCont* (λx. *inverse* (f x)) a
unfolding *isCont-def* **by** (*rule LIM-inverse*)

lemma *isCont-LIM-compose*:
⟦*isCont g l*; f −− a −−> l⟧ ⟹ (λx. g (f x)) −− a −−> g l
unfolding *isCont-def* **by** (*rule LIM-compose*)

lemma *isCont-LIM-compose2*:

assumes *f* [*unfolded isCont-def*]: *isCont f a*
assumes *g*: *g* −− *f a* −−> *l*
assumes *inj*: ∃ *d*>*0*. ∀ *x*. *x* ≠ *a* ∧ *norm* (*x* − *a*) < *d* ⟶ *f x* ≠ *f a*
shows (λ*x*. *g* (*f x*)) −− *a* −−> *l*
by (*rule LIM-compose2* [*OF f g inj*])

lemma *isCont-o2*: ⟦*isCont f a*; *isCont g* (*f a*)⟧ ⟹ *isCont* (λ*x*. *g* (*f x*)) *a*
unfolding *isCont-def* **by** (*rule LIM-compose*)

lemma *isCont-o*: ⟦*isCont f a*; *isCont g* (*f a*)⟧ ⟹ *isCont* (*g o f*) *a*
unfolding *o-def* **by** (*rule isCont-o2*)

lemma (**in** *bounded-linear*) *isCont*: *isCont f a*
unfolding *isCont-def* **by** (*rule cont*)

lemma (**in** *bounded-bilinear*) *isCont*:
⟦*isCont f a*; *isCont g a*⟧ ⟹ *isCont* (λ*x*. *f x* ** *g x*) *a*
unfolding *isCont-def* **by** (*rule LIM*)

lemmas *isCont-scaleR* = *scaleR.isCont*

lemma *isCont-of-real*:
isCont f a ⟹ *isCont* (λ*x*. *of-real* (*f x*)) *a*
unfolding *isCont-def* **by** (*rule LIM-of-real*)

lemma *isCont-power*:
fixes *f* :: *'a*::*real-normed-vector* ⇒ *'b*::{*recpower*,*real-normed-algebra*}
shows *isCont f a* ⟹ *isCont* (λ*x*. *f x* ^ *n*) *a*
unfolding *isCont-def* **by** (*rule LIM-power*)

lemma *isCont-sgn*:
⟦*isCont f a*; *f a* ≠ *0*⟧ ⟹ *isCont* (λ*x*. *sgn* (*f x*)) *a*
unfolding *isCont-def* **by** (*rule LIM-sgn*)

lemma *isCont-abs* [*simp*]: *isCont abs* (*a*::*real*)
by (*rule isCont-rabs* [*OF isCont-ident*])

lemma *isCont-setsum*: **fixes** *A* :: *nat set* **assumes** *finite A*
shows ∀ *i* ∈ *A*. *isCont* (*f i*) *x* ⟹ *isCont* (λ *x*. ∑ *i* ∈ *A*. *f i x*) *x*
using ‹*finite A*›
proof *induct*
case (*insert a F*) **show** *isCont* (λ *x*. ∑ *i* ∈ (*insert a F*). *f i x*) *x*
unfolding *setsum-insert*[*OF* ‹*finite F*› ‹*a* ∉ *F*›] **by** (*rule isCont-add*, *auto simp add*: *insert*)
qed *auto*

lemma *LIM-less-bound*: **fixes** *f* :: *real* ⇒ *real* **assumes** *b* < *x*
and *all-le*: ∀ *x'* ∈ { *b* <..< *x*}. *0* ≤ *f x'* **and** *isCont*: *isCont f x*
shows *0* ≤ *f x*

proof (*rule ccontr*)
assume ¬ *0* ≤ *f x* **hence** *f x* < *0* **by** *auto*
hence *0* < − *f x* / *2* **by** *auto*
from *isCont*[*unfolded isCont-def*, *THEN LIM-D*, *OF this*]
obtain *s* **where** *s* > *0* **and** *s-D*: ⋀*x'*. ⟦ *x'* ≠ *x* ; | *x'* − *x* | < *s* ⟧ ⟹ | *f x'* − *f x* | < − *f x* / *2* **by** *auto*

let *?x* = *x* − *min* (*s* / *2*) ((*x* − *b*) / *2*)
have *?x* < *x* **and** | *?x* − *x* | < *s*
using ‹*b* < *x*› **and** ‹*0* < *s*› **by** *auto*
have *b* < *?x*
proof (*cases s* < *x* − *b*)
case *True* **thus** *?thesis* **using** ‹*0* < *s*› **by** *auto*
next
case *False* **hence** *s* / *2* ≥ (*x* − *b*) / *2* **by** *auto*
from *inf-absorb2*[*OF this*, *unfolded inf-real-def*]
have *?x* = (*x* + *b*) / *2* **by** *auto*
thus *?thesis* **using** ‹*b* < *x*› **by** *auto*
qed
hence *0* ≤ *f ?x* **using** *all-le* ‹*?x* < *x*› **by** *auto*
moreover have |*f ?x* − *f x*| < − *f x* / *2*
using *s-D*[*OF* - ‹| *?x* − *x* | < *s*›] ‹*?x* < *x*› **by** *auto*
hence *f ?x* − *f x* < − *f x* / *2* **by** *auto*
hence *f ?x* < *f x* / *2* **by** *auto*
hence *f ?x* < *0* **using** ‹*f x* < *0*› **by** *auto*
thus *False* **using** ‹*0* ≤ *f ?x*› **by** *auto*
qed

59.3 Uniform Continuity

lemma *isUCont-isCont*: *isUCont f* ==> *isCont f x*
by (*simp add*: *isUCont-def isCont-def LIM-def*, *force*)

lemma *isUCont-Cauchy*:
⟦*isUCont f*; *Cauchy X*⟧ ⟹ *Cauchy* (λ*n*. *f* (*X n*))
unfolding *isUCont-def*
apply (*rule CauchyI*)
apply (*drule-tac x*=*e* **in** *spec*, *safe*)
apply (*drule-tac e*=*s* **in** *CauchyD*, *safe*)
apply (*rule-tac x*=*M* **in** *exI*, *simp*)
done

lemma (**in** *bounded-linear*) *isUCont*: *isUCont f*
unfolding *isUCont-def*
proof (*intro allI impI*)
fix *r*::*real* **assume** *r*: *0* < *r*
obtain *K* **where** *K*: *0* < *K* **and** *norm-le*: ⋀*x*. *norm* (*f x*) ≤ *norm x* ∗ *K*
using *pos-bounded* **by** *fast*
show ∃ *s*>*0*. ∀ *x y*. *norm* (*x* − *y*) < *s* ⟶ *norm* (*f x* − *f y*) < *r*

proof (*rule exI*, *safe*)
from *r K* **show** *0 < r / K* **by** (*rule divide-pos-pos*)
next
fix *x y* :: *'a*
assume *xy*: *norm (x − y) < r / K*
have *norm (f x − f y) = norm (f (x − y))* **by** (*simp only*: *diff*)
also have *... ≤ norm (x − y) * K* **by** (*rule norm-le*)
also from *K xy* **have** *... < r* **by** (*simp only*: *pos-less-divide-eq*)
finally show *norm (f x − f y) < r* **.**
qed
qed

lemma (**in** *bounded-linear*) *Cauchy*: *Cauchy X ⟹ Cauchy (λn. f (X n))*
by (*rule isUCont* [*THEN isUCont-Cauchy*])

59.4 Relation of LIM and LIMSEQ

lemma *LIMSEQ-SEQ-conv1*:
fixes *a* :: *'a*::*real-normed-vector*
assumes *X*: *X −− a −−> L*
shows *∀ S. (∀ n. S n ≠ a) ∧ S −−−−> a ⟶ (λn. X (S n)) −−−−> L*
proof (*safe intro*!: *LIMSEQ-I*)
fix *S* :: *nat ⇒ 'a*
fix *r* :: *real*
assume *rgz*: *0 < r*
assume *as*: *∀ n. S n ≠ a*
assume *S*: *S −−−−> a*
from *LIM-D* [*OF X rgz*] **obtain** *s*
where *sgz*: *0 < s*
and *aux*: *⋀x. ⟦x ≠ a; norm (x − a) < s⟧ ⟹ norm (X x − L) < r*
by *fast*
from *LIMSEQ-D* [*OF S sgz*]
obtain *no* **where** *∀ n≥no. norm (S n − a) < s* **by** *blast*
hence *∀ n≥no. norm (X (S n) − L) < r* **by** (*simp add*: *aux as*)
thus *∃ no. ∀ n≥no. norm (X (S n) − L) < r* **..**
qed

lemma *LIMSEQ-SEQ-conv2*:
fixes *a* :: *real*
assumes *∀ S. (∀ n. S n ≠ a) ∧ S −−−−> a ⟶ (λn. X (S n)) −−−−> L*
shows *X −− a −−> L*
proof (*rule ccontr*)
assume *¬ (X −− a −−> L)*
hence *¬ (∀ r > 0. ∃ s > 0. ∀ x. x ≠ a & norm (x − a) < s −−> norm (X x − L) < r)* **by** (*unfold LIM-def*)
hence *∃ r > 0. ∀ s > 0. ∃ x. ¬(x ≠ a ∧ |x − a| < s −−> norm (X x − L) < r)* **by** *simp*
hence *∃ r > 0. ∀ s > 0. ∃ x. (x ≠ a ∧ |x − a| < s ∧ norm (X x − L) ≥ r)* **by** (*simp add*: *linorder-not-less*)

then obtain r **where** *rdef*: $r > 0 \wedge (\forall s > 0.\ \exists x.\ (x \neq a \wedge |x - a| < s \wedge norm\ (X\ x - L) \geq r))$ **by** *auto*

let *?F* = $\lambda n{::}nat.\ SOME\ x.\ x{\neq}a \wedge |x - a| < inverse\ (real\ (Suc\ n)) \wedge norm\ (X\ x - L) \geq r$
have $\bigwedge n.\ \exists x.\ x{\neq}a \wedge |x - a| < inverse\ (real\ (Suc\ n)) \wedge norm\ (X\ x - L) \geq r$
using *rdef* **by** *simp*
hence *F*: $\bigwedge n.\ ?F\ n \neq a \wedge |?F\ n - a| < inverse\ (real\ (Suc\ n)) \wedge norm\ (X\ (?F\ n) - L) \geq r$
by (*rule someI-ex*)
hence *F1*: $\bigwedge n.\ ?F\ n \neq a$
and *F2*: $\bigwedge n.\ |?F\ n - a| < inverse\ (real\ (Suc\ n))$
and *F3*: $\bigwedge n.\ norm\ (X\ (?F\ n) - L) \geq r$
by *fast+*

have *?F* $-\!\!-\!\!-\!\!-\!\!-\!\!>$ *a*
proof (*rule LIMSEQ-I*, *unfold real-norm-def*)
fix *e*::*real*
assume $0 < e$

then have $\exists no.\ inverse\ (real\ (Suc\ no)) < e$ **by** (*rule reals-Archimedean*)
then obtain m **where** *nodef*: $inverse\ (real\ (Suc\ m)) < e$ **by** *auto*
show $\exists no.\ \forall n.\ no \leq n \longrightarrow |?F\ n - a| < e$
proof (*intro exI allI impI*)
fix n
assume *mlen*: $m \leq n$
have $|?F\ n - a| < inverse\ (real\ (Suc\ n))$
by (*rule F2*)
also have $inverse\ (real\ (Suc\ n)) \leq inverse\ (real\ (Suc\ m))$
using *mlen* **by** *auto*
also from *nodef* **have**
$inverse\ (real\ (Suc\ m)) < e$ **.**
finally show $|?F\ n - a| < e$ **.**
qed
qed

moreover have $\forall n.\ ?F\ n \neq a$
by (*rule allI*) (*rule F1*)

moreover from *prems* **have** $\forall S.\ (\forall n.\ S\ n \neq a) \wedge S\ -\!\!-\!\!-\!\!-\!\!-\!\!>\ a \longrightarrow (\lambda n.\ X\ (S\ n))\ -\!\!-\!\!-\!\!-\!\!-\!\!>\ L$ **by** *simp*
ultimately have $(\lambda n.\ X\ (?F\ n))\ -\!\!-\!\!-\!\!-\!\!-\!\!>\ L$ **by** *simp*

moreover have $\neg\ ((\lambda n.\ X\ (?F\ n))\ -\!\!-\!\!-\!\!-\!\!-\!\!>\ L)$
proof −
{
fix *no*::*nat*
obtain n **where** $n = no + 1$ **by** *simp*
then have *nolen*: $no \leq n$ **by** *simp*

have *norm* (*X* (*?F n*) − *L*) ≥ *r*
by (*rule F3*)
with *nolen* **have** ∃ *n*. *no* ≤ *n* ∧ *norm* (*X* (*?F n*) − *L*) ≥ *r* **by** *fast*
}
then have (∀ *no*. ∃ *n*. *no* ≤ *n* ∧ *norm* (*X* (*?F n*) − *L*) ≥ *r*) **by** *simp*
with *rdef* **have** ∃ *e*>*0*. (∀ *no*. ∃ *n*. *no* ≤ *n* ∧ *norm* (*X* (*?F n*) − *L*) ≥ *e*) **by** *auto*
thus *?thesis* **by** (*unfold LIMSEQ-def*, *auto simp add*: *linorder-not-less*)
qed
ultimately show *False* **by** *simp*
qed

lemma *LIMSEQ-SEQ-conv*:
(∀ *S*. (∀ *n*. *S n* ≠ *a*) ∧ *S* −−−−> (*a*::*real*) ⟶ (λ*n*. *X* (*S n*)) −−−−> *L*) =
(*X* −− *a* −−> *L*)
proof
assume ∀ *S*. (∀ *n*. *S n* ≠ *a*) ∧ *S* −−−−> *a* ⟶ (λ*n*. *X* (*S n*)) −−−−> *L*
thus *X* −− *a* −−> *L* **by** (*rule LIMSEQ-SEQ-conv2*)
next
assume (*X* −− *a* −−> *L*)
thus ∀ *S*. (∀ *n*. *S n* ≠ *a*) ∧ *S* −−−−> *a* ⟶ (λ*n*. *X* (*S n*)) −−−−> *L* **by** (*rule LIMSEQ-SEQ-conv1*)
qed

end

60 Deriv: Differentiation

theory *Deriv*
imports *Lim*
begin

Standard Definitions

definition
deriv :: [*'a*::*real-normed-field* ⇒ *'a*, *'a*, *'a*] ⇒ *bool*
— Differentiation: D is derivative of function f at x
((*DERIV* (-)/ (-)/ :> (-)) [*1000*, *1000*, *60*] *60*) **where**
DERIV f x :> *D* = ((%*h*. (*f*(*x* + *h*) − *f x*) / *h*) −− *0* −−> *D*)

consts
Bolzano-bisect :: [*real*∗*real*=>*bool*, *real*, *real*, *nat*] => (*real*∗*real*)
primrec
Bolzano-bisect P a b 0 = (*a*,*b*)
Bolzano-bisect P a b (*Suc n*) =
(*let* (*x*,*y*) = *Bolzano-bisect P a b n*
in if P(*x*, (*x*+*y*)/*2*) *then* ((*x*+*y*)/*2*, *y*)
else (*x*, (*x*+*y*)/*2*))

60.1 Derivatives

lemma *DERIV-iff*: $(DERIV\ f\ x :> D) = ((\%h.\ (f(x + h) - f(x))/h) -- 0 --> D)$
by (*simp add*: *deriv-def*)

lemma *DERIV-D*: $DERIV\ f\ x :> D ==> (\%h.\ (f(x + h) - f(x))/h) -- 0 --> D$
by (*simp add*: *deriv-def*)

lemma *DERIV-const* [*simp*]: $DERIV\ (\lambda x.\ k)\ x :> 0$
by (*simp add*: *deriv-def*)

lemma *DERIV-ident* [*simp*]: $DERIV\ (\lambda x.\ x)\ x :> 1$
by (*simp add*: *deriv-def cong*: *LIM-cong*)

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 *DERIV-add*:
$[\![DERIV\ f\ x :> D;\ DERIV\ g\ x :> E]\!] \Longrightarrow DERIV\ (\lambda x.\ f\ x + g\ x)\ x :> D + E$
by (*simp only*: *deriv-def add-diff-add add-divide-distrib LIM-add*)

lemma *DERIV-minus*:
$DERIV\ f\ x :> D \Longrightarrow DERIV\ (\lambda x.\ -\ f\ x)\ x :> -\ D$
by (*simp only*: *deriv-def minus-diff-minus divide-minus-left LIM-minus*)

lemma *DERIV-diff*:
$[\![DERIV\ f\ x :> D;\ DERIV\ g\ x :> E]\!] \Longrightarrow DERIV\ (\lambda x.\ f\ x - g\ x)\ x :> D - E$
by (*simp only*: *diff-def DERIV-add DERIV-minus*)

lemma *DERIV-add-minus*:
$[\![DERIV\ f\ x :> D;\ DERIV\ g\ x :> E]\!] \Longrightarrow DERIV\ (\lambda x.\ f\ x + -\ g\ x)\ x :> D + -\ E$
by (*simp only*: *DERIV-add DERIV-minus*)

lemma *DERIV-isCont*: $DERIV\ f\ x :> D \Longrightarrow isCont\ f\ x$
proof (*unfold isCont-iff*)
assume $DERIV\ f\ x :> D$
hence $(\lambda h.\ (f(x+h) - f(x))\ /\ h) -- 0 --> D$
by (*rule DERIV-D*)
hence $(\lambda h.\ (f(x+h) - f(x))\ /\ h * h) -- 0 --> D * 0$
by (*intro LIM-mult LIM-ident*)
hence $(\lambda h.\ (f(x+h) - f(x)) * (h\ /\ h)) -- 0 --> 0$
by *simp*
hence $(\lambda h.\ f(x+h) - f(x)) -- 0 --> 0$
by (*simp cong*: *LIM-cong*)
thus $(\lambda h.\ f(x+h)) -- 0 --> f(x)$

by (*simp add*: *LIM-def*)
qed

lemma *DERIV-mult-lemma*:
fixes *a b c d* :: *'a*::*real-field*
shows $(a * b - c * d) / h = a * ((b - d) / h) + ((a - c) / h) * d$
by (*simp add*: *diff-minus add-divide-distrib* [*symmetric*] *ring-distribs*)

lemma *DERIV-mult'*:
assumes *f*: *DERIV f x* :> *D*
assumes *g*: *DERIV g x* :> *E*
shows *DERIV* $(\lambda x.\ f\ x * g\ x)\ x$:> $f\ x * E + D * g\ x$
proof (*unfold deriv-def*)
from *f* **have** *isCont f x*
by (*rule DERIV-isCont*)
hence $(\lambda h.\ f(x+h)) -- 0 --> f\ x$
by (*simp only*: *isCont-iff*)
hence $(\lambda h.\ f(x+h) * ((g(x+h) - g\ x) / h) +$
$((f(x+h) - f\ x) / h) * g\ x)$
$-- 0 --> f\ x * E + D * g\ x$
by (*intro LIM-add LIM-mult LIM-const DERIV-D f g*)
thus $(\lambda h.\ (f(x+h) * g(x+h) - f\ x * g\ x) / h)$
$-- 0 --> f\ x * E + D * g\ x$
by (*simp only*: *DERIV-mult-lemma*)
qed

lemma *DERIV-mult*:
$[|$ *DERIV f x* :> *Da*; *DERIV g x* :> *Db* $|]$
$==>$ *DERIV* $(\%x.\ f\ x * g\ x)\ x$:> $(Da * g(x)) + (Db * f(x))$
by (*drule* (*1*) *DERIV-mult'*, *simp only*: *mult-commute add-commute*)

lemma *DERIV-unique*:
$[|$ *DERIV f x* :> *D*; *DERIV f x* :> *E* $|] ==> D = E$
apply (*simp add*: *deriv-def*)
apply (*blast intro*: *LIM-unique*)
done

Differentiation of finite sum

lemma *DERIV-sumr* [*rule-format* (*no-asm*)]:
$(\forall r.\ m \leq r\ \&\ r < (m + n) --> DERIV\ (\%x.\ f\ r\ x)\ x$:> $(f'\ r\ x))$
$--> DERIV\ (\%x.\ \sum n{=}m..{<}n{::}nat.\ f\ n\ x :: real)\ x$:> $(\sum r{=}m..{<}n.\ f'\ r\ x)$
apply (*induct n*)
apply (*auto intro*: *DERIV-add*)
done

Alternative definition for differentiability

lemma *DERIV-LIM-iff*:
$((\%h.\ (f(a + h) - f(a)) / h) -- 0 --> D) =$

```
    ((%x. (f(x)-f(a)) / (x-a)) -- a --> D)
apply (rule iffI)
apply (drule-tac k=- a in LIM-offset)
apply (simp add: diff-minus)
apply (drule-tac k=a in LIM-offset)
apply (simp add: add-commute)
done

lemma DERIV-iff2: (DERIV f x :> D) = ((%z. (f(z) - f(x)) / (z-x)) -- x
--> D)
by (simp add: deriv-def diff-minus [symmetric] DERIV-LIM-iff)

lemma inverse-diff-inverse:
 [|(a::'a::division-ring) ≠ 0; b ≠ 0|]
  ⟹ inverse a - inverse b = - (inverse a * (a - b) * inverse b)
by (simp add: algebra-simps)

lemma DERIV-inverse-lemma:
 [|a ≠ 0; b ≠ (0::'a::real-normed-field)|]
  ⟹ (inverse a - inverse b) / h
    = - (inverse a * ((a - b) / h) * inverse b)
by (simp add: inverse-diff-inverse)

lemma DERIV-inverse':
 assumes der: DERIV f x :> D
 assumes neq: f x ≠ 0
 shows DERIV (λx. inverse (f x)) x :> - (inverse (f x) * D * inverse (f x))
   (is DERIV - - :> ?E)
proof (unfold DERIV-iff2)
 from der have lim-f: f -- x --> f x
   by (rule DERIV-isCont [unfolded isCont-def])

 from neq have 0 < norm (f x) by simp
 with LIM-D [OF lim-f] obtain s
   where s: 0 < s
   and less-fx: ⋀z. [|z ≠ x; norm (z - x) < s|]
             ⟹ norm (f z - f x) < norm (f x)
   by fast

 show (λz. (inverse (f z) - inverse (f x)) / (z - x)) -- x --> ?E
 proof (rule LIM-equal2 [OF s])
   fix z
   assume z ≠ x norm (z - x) < s
   hence norm (f z - f x) < norm (f x) by (rule less-fx)
   hence f z ≠ 0 by auto
   thus (inverse (f z) - inverse (f x)) / (z - x) =
        - (inverse (f z) * ((f z - f x) / (z - x)) * inverse (f x))
     using neq by (rule DERIV-inverse-lemma)
 next
```

```
    from der have (λz. (f z − f x) / (z − x)) −− x −−> D
      by (unfold DERIV-iff2)
    thus (λz. − (inverse (f z) ∗ ((f z − f x) / (z − x)) ∗ inverse (f x)))
          −− x −−> ?E
      by (intro LIM-mult LIM-inverse LIM-minus LIM-const lim-f neq)
  qed
qed

lemma DERIV-divide:
  ⟦DERIV f x :> D; DERIV g x :> E; g x ≠ 0⟧
   ⟹ DERIV (λx. f x / g x) x :> (D ∗ g x − f x ∗ E) / (g x ∗ g x)
apply (subgoal-tac f x ∗ − (inverse (g x) ∗ E ∗ inverse (g x)) +
        D ∗ inverse (g x) = (D ∗ g x − f x ∗ E) / (g x ∗ g x))
apply (erule subst)
apply (unfold divide-inverse)
apply (erule DERIV-mult')
apply (erule (1) DERIV-inverse')
apply (simp add: ring-distribs nonzero-inverse-mult-distrib)
apply (simp add: mult-ac)
done

lemma DERIV-power-Suc:
  fixes f :: 'a ⇒ 'a::{real-normed-field,recpower}
  assumes f: DERIV f x :> D
  shows DERIV (λx. f x ^ Suc n) x :> (1 + of-nat n) ∗ (D ∗ f x ^ n)
proof (induct n)
case 0
  show ?case by (simp add: f)
case (Suc k)
  from DERIV-mult' [OF f Suc] show ?case
    apply (simp only: of-nat-Suc ring-distribs mult-1-left)
    apply (simp only: power-Suc algebra-simps)
    done
qed

lemma DERIV-power:
  fixes f :: 'a ⇒ 'a::{real-normed-field,recpower}
  assumes f: DERIV f x :> D
  shows DERIV (λx. f x ^ n) x :> of-nat n ∗ (D ∗ f x ^ (n − Suc 0))
by (cases n, simp, simp add: DERIV-power-Suc f del: power-Suc)
```

Caratheodory formulation of derivative at a point

```
lemma CARAT-DERIV:
     (DERIV f x :> l) =
      (∃ g. (∀ z. f z − f x = g z ∗ (z−x)) & isCont g x & g x = l)
      (is ?lhs = ?rhs)
proof
  assume der: DERIV f x :> l
  show ∃ g. (∀ z. f z − f x = g z ∗ (z−x)) ∧ isCont g x ∧ g x = l
```

proof (*intro exI conjI*)
let *?g* = (%*z*. *if z* = *x then l else* (*f z* − *f x*) / (*z*−*x*))
show ∀ *z*. *f z* − *f x* = *?g z* ∗ (*z*−*x*) **by** *simp*
show *isCont ?g x* **using** *der*
by (*simp add*: *isCont-iff DERIV-iff diff-minus*
cong: *LIM-equal* [*rule-format*])
show *?g x* = *l* **by** *simp*
qed
next
assume *?rhs*
then obtain *g* **where**
(∀ *z*. *f z* − *f x* = *g z* ∗ (*z*−*x*)) **and** *isCont g x* **and** *g x* = *l* **by** *blast*
thus (*DERIV f x* :> *l*)
by (*auto simp add*: *isCont-iff DERIV-iff cong*: *LIM-cong*)
qed

lemma *DERIV-chain'*:
assumes *f*: *DERIV f x* :> *D*
assumes *g*: *DERIV g* (*f x*) :> *E*
shows *DERIV* (λ*x*. *g* (*f x*)) *x* :> *E* ∗ *D*
proof (*unfold DERIV-iff2*)
obtain *d* **where** *d*: ∀ *y*. *g y* − *g* (*f x*) = *d y* ∗ (*y* − *f x*)
and *cont-d*: *isCont d* (*f x*) **and** *dfx*: *d* (*f x*) = *E*
using *CARAT-DERIV* [*THEN iffD1*, *OF g*] **by** *fast*
from *f* **have** *f* −− *x* −−> *f x*
by (*rule DERIV-isCont* [*unfolded isCont-def*])
with *cont-d* **have** (λ*z*. *d* (*f z*)) −− *x* −−> *d* (*f x*)
by (*rule isCont-LIM-compose*)
hence (λ*z*. *d* (*f z*) ∗ ((*f z* − *f x*) / (*z* − *x*)))
−− *x* −−> *d* (*f x*) ∗ *D*
by (*rule LIM-mult* [*OF - f* [*unfolded DERIV-iff2*]])
thus (λ*z*. (*g* (*f z*) − *g* (*f x*)) / (*z* − *x*)) −− *x* −−> *E* ∗ *D*
by (*simp add*: *d dfx real-scaleR-def*)
qed

lemma *DERIV-cmult*:
DERIV f x :> *D* ==> *DERIV* (%*x*. *c* ∗ *f x*) *x* :> *c*∗*D*
by (*drule DERIV-mult'* [*OF DERIV-const*], *simp*)

lemma *DERIV-chain*: [| *DERIV f* (*g x*) :> *Da*; *DERIV g x* :> *Db* |] ==> *DERIV* (*f o g*) *x* :> *Da* ∗ *Db*
by (*drule* (*1*) *DERIV-chain'*, *simp add*: *o-def real-scaleR-def mult-commute*)

lemma *DERIV-chain2*: [| *DERIV f* (*g x*) :> *Da*; *DERIV g x* :> *Db* |] ==> *DERIV* (%*x*. *f* (*g x*)) *x* :> *Da* ∗ *Db*

by (*auto dest*: *DERIV-chain simp add*: *o-def*)

lemma *DERIV-cmult-Id* [*simp*]: *DERIV* (*op* ∗ *c*) *x* :> *c*
by (*cut-tac c* = *c* **and** *x* = *x* **in** *DERIV-ident* [*THEN DERIV-cmult*], *simp*)

lemma *DERIV-pow*: *DERIV* (%*x*. *x* ^ *n*) *x* :> *real n* ∗ (*x* ^ (*n* − *Suc 0*))
apply (*cut-tac DERIV-power* [*OF DERIV-ident*])
apply (*simp add*: *real-scaleR-def real-of-nat-def*)
done

Power of -1

lemma *DERIV-inverse*:
fixes *x* :: *'a*::{*real-normed-field*,*recpower*}
shows *x* ≠ *0* ==> *DERIV* (%*x*. *inverse*(*x*)) *x* :> (−(*inverse x* ^ *Suc* (*Suc 0*)))
by (*drule DERIV-inverse'* [*OF DERIV-ident*]) *simp*

Derivative of inverse

lemma *DERIV-inverse-fun*:
fixes *x* :: *'a*::{*real-normed-field*,*recpower*}
shows [| *DERIV f x* :> *d*; *f*(*x*) ≠ *0* |]
==> *DERIV* (%*x*. *inverse*(*f x*)) *x* :> (− (*d* ∗ *inverse*(*f*(*x*) ^ *Suc* (*Suc 0*))))
by (*drule* (*1*) *DERIV-inverse'*) (*simp add*: *mult-ac nonzero-inverse-mult-distrib*)

Derivative of quotient

lemma *DERIV-quotient*:
fixes *x* :: *'a*::{*real-normed-field*,*recpower*}
shows [| *DERIV f x* :> *d*; *DERIV g x* :> *e*; *g*(*x*) ≠ *0* |]
==> *DERIV* (%*y*. *f*(*y*) / (*g y*)) *x* :> (*d*∗*g*(*x*) − (*e*∗*f*(*x*))) / (*g*(*x*) ^ *Suc* (*Suc 0*))
by (*drule* (*2*) *DERIV-divide*) (*simp add*: *mult-commute*)

lemma *lemma-DERIV-subst*: [| *DERIV f x* :> *D*; *D* = *E* |] ==> *DERIV f x* :> *E*
by *auto*

60.2 Differentiability predicate

definition
differentiable :: [*'a*::*real-normed-field* ⇒ *'a*, *'a*] ⇒ *bool*
(**infixl** *differentiable 60*) **where**
f differentiable x = (∃ *D*. *DERIV f x* :> *D*)

lemma *differentiableE* [*elim?*]:
assumes *f differentiable x*
obtains *df* **where** *DERIV f x* :> *df*
using *prems* **unfolding** *differentiable-def* **..**

lemma *differentiableD*: *f differentiable x* ==> ∃ *D*. *DERIV f x* :> *D*

by (*simp add*: *differentiable-def*)

lemma *differentiableI*: *DERIV f x :> D ==> f differentiable x*
by (*force simp add*: *differentiable-def*)

lemma *differentiable-ident* [*simp*]: (λ*x*. *x*) *differentiable x*
by (*rule DERIV-ident* [*THEN differentiableI*])

lemma *differentiable-const* [*simp*]: (λ*z*. *a*) *differentiable x*
by (*rule DERIV-const* [*THEN differentiableI*])

lemma *differentiable-compose*:
assumes *f*: *f differentiable* (*g x*)
assumes *g*: *g differentiable x*
shows (λ*x*. *f* (*g x*)) *differentiable x*
proof −
from ‹*f differentiable* (*g x*)› **obtain** *df* **where** *DERIV f* (*g x*) :> *df* **..**
moreover
from ‹*g differentiable x*› **obtain** *dg* **where** *DERIV g x* :> *dg* **..**
ultimately
have *DERIV* (λ*x*. *f* (*g x*)) *x* :> *df* ∗ *dg* **by** (*rule DERIV-chain2*)
thus *?thesis* **by** (*rule differentiableI*)
qed

lemma *differentiable-sum* [*simp*]:
assumes *f differentiable x*
and *g differentiable x*
shows (λ*x*. *f x* + *g x*) *differentiable x*
proof −
from ‹*f differentiable x*› **obtain** *df* **where** *DERIV f x* :> *df* **..**
moreover
from ‹*g differentiable x*› **obtain** *dg* **where** *DERIV g x* :> *dg* **..**
ultimately
have *DERIV* (λ*x*. *f x* + *g x*) *x* :> *df* + *dg* **by** (*rule DERIV-add*)
thus *?thesis* **by** (*rule differentiableI*)
qed

lemma *differentiable-minus* [*simp*]:
assumes *f differentiable x*
shows (λ*x*. − *f x*) *differentiable x*
proof −
from ‹*f differentiable x*› **obtain** *df* **where** *DERIV f x* :> *df* **..**
hence *DERIV* (λ*x*. − *f x*) *x* :> − *df* **by** (*rule DERIV-minus*)
thus *?thesis* **by** (*rule differentiableI*)
qed

lemma *differentiable-diff* [*simp*]:
assumes *f differentiable x*
assumes *g differentiable x*

shows $(\lambda x.\ f\ x - g\ x)$ *differentiable* x
unfolding *diff-minus* **using** *prems* **by** *simp*

lemma *differentiable-mult* [*simp*]:
assumes f *differentiable* x
assumes g *differentiable* x
shows $(\lambda x.\ f\ x * g\ x)$ *differentiable* x
proof −
from ‹f *differentiable* x› **obtain** df **where** *DERIV* $f\ x$:> df **..**
moreover
from ‹g *differentiable* x› **obtain** dg **where** *DERIV* $g\ x$:> dg **..**
ultimately
have *DERIV* $(\lambda x.\ f\ x * g\ x)\ x$:> $df * g\ x + dg * f\ x$ **by** (*rule DERIV-mult*)
thus *?thesis* **by** (*rule differentiableI*)
qed

lemma *differentiable-inverse* [*simp*]:
assumes f *differentiable* x **and** $f\ x \neq 0$
shows $(\lambda x.\ inverse\ (f\ x))$ *differentiable* x
proof −
from ‹f *differentiable* x› **obtain** df **where** *DERIV* $f\ x$:> df **..**
hence *DERIV* $(\lambda x.\ inverse\ (f\ x))\ x$:> $-\ (inverse\ (f\ x) * df * inverse\ (f\ x))$
using ‹$f\ x \neq 0$› **by** (*rule DERIV-inverse'*)
thus *?thesis* **by** (*rule differentiableI*)
qed

lemma *differentiable-divide* [*simp*]:
assumes f *differentiable* x
assumes g *differentiable* x **and** $g\ x \neq 0$
shows $(\lambda x.\ f\ x\ /\ g\ x)$ *differentiable* x
unfolding *divide-inverse* **using** *prems* **by** *simp*

lemma *differentiable-power* [*simp*]:
fixes f :: $'a$::{*recpower*,*real-normed-field*} ⇒ $'a$
assumes f *differentiable* x
shows $(\lambda x.\ f\ x\ \hat{}\ n)$ *differentiable* x
by (*induct* n, *simp*, *simp add*: *prems*)

60.3 Nested Intervals and Bisection

Lemmas about nested intervals and proof by bisection (cf.Harrison). All considerably tidied by lcp.

lemma *lemma-f-mono-add* [*rule-format* (*no-asm*)]: $(\forall n.\ (f$::*nat*=>*real*$)\ n \leq f\ (Suc\ n)) \longrightarrow f\ m \leq f(m + no)$
apply (*induct no*)
apply (*auto intro*: *order-trans*)
done

lemma *f-inc-g-dec-Beq-f*: [| $\forall n.\ f(n) \leq f(Suc\ n)$;

```
        ∀ n. g(Suc n) ≤ g(n);
        ∀ n. f(n) ≤ g(n) |]
     ==> Bseq (f :: nat ⇒ real)
apply (rule-tac k = f 0 and K = g 0 in BseqI2, rule allI)
apply (induct-tac n)
apply (auto intro: order-trans)
apply (rule-tac y = g (Suc na) in order-trans)
apply (induct-tac [2] na)
apply (auto intro: order-trans)
done

lemma f-inc-g-dec-Beq-g: [| ∀ n. f(n) ≤ f(Suc n);
        ∀ n. g(Suc n) ≤ g(n);
        ∀ n. f(n) ≤ g(n) |]
     ==> Bseq (g :: nat ⇒ real)
apply (subst Bseq-minus-iff [symmetric])
apply (rule-tac g = %x. - (f x) in f-inc-g-dec-Beq-f)
apply auto
done

lemma f-inc-imp-le-lim:
  fixes f :: nat ⇒ real
  shows ⟦∀ n. f n ≤ f (Suc n); convergent f⟧ ⟹ f n ≤ lim f
apply (rule linorder-not-less [THEN iffD1])
apply (auto simp add: convergent-LIMSEQ-iff LIMSEQ-iff monoseq-Suc)
apply (drule real-less-sum-gt-zero)
apply (drule-tac x = f n + - lim f in spec, safe)
apply (drule-tac P = %na. no≤na --> ?Q na and x = no + n in spec, auto)
apply (subgoal-tac lim f ≤ f (no + n) )
apply (drule-tac no=no and m=n in lemma-f-mono-add)
apply (auto simp add: add-commute)
apply (induct-tac no)
apply simp
apply (auto intro: order-trans simp add: diff-minus abs-if)
done

lemma lim-uminus: convergent g ==> lim (%x. - g x) = - (lim g)
apply (rule LIMSEQ-minus [THEN limI])
apply (simp add: convergent-LIMSEQ-iff)
done

lemma g-dec-imp-lim-le:
  fixes g :: nat ⇒ real
  shows ⟦∀ n. g (Suc n) ≤ g(n); convergent g⟧ ⟹ lim g ≤ g n
apply (subgoal-tac - (g n) ≤ - (lim g) )
apply (cut-tac [2] f = %x. - (g x) in f-inc-imp-le-lim)
apply (auto simp add: lim-uminus convergent-minus-iff [symmetric])
done
```

lemma *lemma-nest*: [| $\forall n.\ f(n) \leq f(Suc\ n)$;
$\forall n.\ g(Suc\ n) \leq g(n)$;
$\forall n.\ f(n) \leq g(n)$ |]
==> $\exists l\ m :: real.\ l \leq m\ \&\ ((\forall n.\ f(n) \leq l)\ \&\ f\ ----->\ l)\ \&$
$((\forall n.\ m \leq g(n))\ \&\ g\ ----->\ m)$
apply (*subgoal-tac monoseq f & monoseq g*)
prefer *2* **apply** (*force simp add*: *LIMSEQ-iff monoseq-Suc*)
apply (*subgoal-tac Bseq f & Bseq g*)
prefer *2* **apply** (*blast intro*: *f-inc-g-dec-Beq-f f-inc-g-dec-Beq-g*)
apply (*auto dest*!: *Bseq-monoseq-convergent simp add*: *convergent-LIMSEQ-iff*)
apply (*rule-tac x = lim f* **in** *exI*)
apply (*rule-tac x = lim g* **in** *exI*)
apply (*auto intro*: *LIMSEQ-le*)
apply (*auto simp add*: *f-inc-imp-le-lim g-dec-imp-lim-le convergent-LIMSEQ-iff*)
done

lemma *lemma-nest-unique*: [| $\forall n.\ f(n) \leq f(Suc\ n)$;
$\forall n.\ g(Suc\ n) \leq g(n)$;
$\forall n.\ f(n) \leq g(n)$;
$(\%n.\ f(n) - g(n))\ ----->\ 0$ |]
==> $\exists l::real.\ ((\forall n.\ f(n) \leq l)\ \&\ f\ ----->\ l)\ \&$
$((\forall n.\ l \leq g(n))\ \&\ g\ ----->\ l)$
apply (*drule lemma-nest*, *auto*)
apply (*subgoal-tac l = m*)
apply (*drule-tac* [*2*] *X = f* **in** *LIMSEQ-diff*)
apply (*auto intro*: *LIMSEQ-unique*)
done

The universal quantifiers below are required for the declaration of *Bolzano-nest-unique* below.

lemma *Bolzano-bisect-le*:
$a \leq b$ ==> $\forall n.\ fst\ (Bolzano\text{-}bisect\ P\ a\ b\ n) \leq snd\ (Bolzano\text{-}bisect\ P\ a\ b\ n)$
apply (*rule allI*)
apply (*induct-tac n*)
apply (*auto simp add*: *Let-def split-def*)
done

lemma *Bolzano-bisect-fst-le-Suc*: $a \leq b$ ==>
$\forall n.\ fst(Bolzano\text{-}bisect\ P\ a\ b\ n) \leq fst\ (Bolzano\text{-}bisect\ P\ a\ b\ (Suc\ n))$
apply (*rule allI*)
apply (*induct-tac n*)
apply (*auto simp add*: *Bolzano-bisect-le Let-def split-def*)
done

lemma *Bolzano-bisect-Suc-le-snd*: $a \leq b$ ==>
$\forall n.\ snd(Bolzano\text{-}bisect\ P\ a\ b\ (Suc\ n)) \leq snd\ (Bolzano\text{-}bisect\ P\ a\ b\ n)$
apply (*rule allI*)
apply (*induct-tac n*)
apply (*auto simp add*: *Bolzano-bisect-le Let-def split-def*)

done

lemma *eq-divide-2-times-iff*: $((x::real) = y \ / \ (2 * z)) = (2 * x = y/z)$
apply (*auto*)
apply (*drule-tac f* = %*u*. (*1*/*2*) ∗*u* **in** *arg-cong*)
apply (*simp*)
done

lemma *Bolzano-bisect-diff*:
$a \leq b \Longrightarrow$
snd(*Bolzano-bisect P a b n*) − *fst*(*Bolzano-bisect P a b n*) =
(*b*−*a*) / (*2* ^ *n*)
apply (*induct n*)
apply (*auto simp add*: *eq-divide-2-times-iff add-divide-distrib Let-def split-def*)
done

lemmas *Bolzano-nest-unique* =
lemma-nest-unique
[*OF Bolzano-bisect-fst-le-Suc Bolzano-bisect-Suc-le-snd Bolzano-bisect-le*]

lemma *not-P-Bolzano-bisect*:
assumes *P*: !!*a b c*. [| *P*(*a*,*b*); *P*(*b*,*c*); $a \leq b$; $b \leq c$|] ==> *P*(*a*,*c*)
and *notP*: ~ *P*(*a*,*b*)
and *le*: $a \leq b$
shows ~ *P*(*fst*(*Bolzano-bisect P a b n*), *snd*(*Bolzano-bisect P a b n*))
proof (*induct n*)
case *0* **show** *?case* **using** *notP* **by** *simp*
next
case (*Suc n*)
thus *?case*
by (*auto simp del*: *surjective-pairing* [*symmetric*]
simp add: *Let-def split-def Bolzano-bisect-le* [*OF le*]
P [*of fst* (*Bolzano-bisect P a b n*) - *snd* (*Bolzano-bisect P a b n*)])
qed

lemma *not-P-Bolzano-bisect'*:
[| ∀ *a b c*. *P*(*a*,*b*) & *P*(*b*,*c*) & $a \leq b$ & $b \leq c$ −−> *P*(*a*,*c*);
~ *P*(*a*,*b*); $a \leq b$ |] ==>
∀ *n*. ~ *P*(*fst*(*Bolzano-bisect P a b n*), *snd*(*Bolzano-bisect P a b n*))
by (*blast elim*!: *not-P-Bolzano-bisect* [*THEN* [*2*] *rev-notE*])

lemma *lemma-BOLZANO*:
[| ∀ *a b c*. *P*(*a*,*b*) & *P*(*b*,*c*) & $a \leq b$ & $b \leq c$ −−> *P*(*a*,*c*);
∀ *x*. ∃ *d*::*real*. *0* < *d* &
(∀ *a b*. $a \leq x$ & $x \leq b$ & (*b*−*a*) < *d* −−> *P*(*a*,*b*));

```
      a ≤ b |]
   ==> P(a,b)
apply (rule Bolzano-nest-unique [where P1=P, THEN exE], assumption+)
apply (rule LIMSEQ-minus-cancel)
apply (simp (no-asm-simp) add: Bolzano-bisect-diff LIMSEQ-divide-realpow-zero)
apply (rule ccontr)
apply (drule not-P-Bolzano-bisect', assumption+)
apply (rename-tac l)
apply (drule-tac x = l in spec, clarify)
apply (simp add: LIMSEQ-def)
apply (drule-tac P = %r. 0<r --> ?Q r and x = d/2 in spec)
apply (drule-tac P = %r. 0<r --> ?Q r and x = d/2 in spec)
apply (drule real-less-half-sum, auto)
apply (drule-tac x = fst (Bolzano-bisect P a b (no + noa)) in spec)
apply (drule-tac x = snd (Bolzano-bisect P a b (no + noa)) in spec)
apply safe
apply (simp-all (no-asm-simp))
apply (rule-tac y = abs (fst (Bolzano-bisect P a b (no + noa)) - l) + abs (snd
(Bolzano-bisect P a b (no + noa)) - l) in order-le-less-trans)
apply (simp (no-asm-simp) add: abs-if)
apply (rule real-sum-of-halves [THEN subst])
apply (rule add-strict-mono)
apply (simp-all add: diff-minus [symmetric])
done


lemma lemma-BOLZANO2: ((∀ a b c. (a ≤ b & b ≤ c & P(a,b) & P(b,c)) --->
P(a,c)) &
      (∀ x. ∃ d::real. 0 < d &
            (∀ a b. a ≤ x & x ≤ b & (b-a) < d ---> P(a,b))))
   ---> (∀ a b. a ≤ b ---> P(a,b))
apply clarify
apply (blast intro: lemma-BOLZANO)
done
```

60.4 Intermediate Value Theorem

Prove Contrapositive by Bisection

```
lemma IVT: [| f(a::real) ≤ (y::real); y ≤ f(b);
      a ≤ b;
      (∀ x. a ≤ x & x ≤ b ---> isCont f x) |]
   ==> ∃ x. a ≤ x & x ≤ b & f(x) = y
apply (rule contrapos-pp, assumption)
apply (cut-tac P = % (u,v) . a ≤ u & u ≤ v & v ≤ b ---> ~ (f (u) ≤ y & y ≤
f (v)) in lemma-BOLZANO2)
apply safe
apply simp-all
apply (simp add: isCont-iff LIM-def)
apply (rule ccontr)
```

apply (*subgoal-tac* $a \leq x$ & $x \leq b$)
prefer *2*
apply *simp*
apply (*drule-tac* $P = \%d.\ 0<d \longrightarrow ?P\ d$ **and** $x = 1$ **in** *spec*, *arith*)
apply (*drule-tac* $x = x$ **in** *spec*)+
apply *simp*
apply (*drule-tac* $P = \%r.\ ?P\ r \longrightarrow (\exists s>0.\ ?Q\ r\ s)$ **and** $x = |y - f\,x|$ **in** *spec*)
apply *safe*
apply *simp*
apply (*drule-tac* $x = s$ **in** *spec*, *clarify*)
apply (*cut-tac* $x = f\,x$ **and** $y = y$ **in** *linorder-less-linear*, *safe*)
apply (*drule-tac* $x = ba - x$ **in** *spec*)
apply (*simp-all add*: *abs-if*)
apply (*drule-tac* $x = aa - x$ **in** *spec*)
apply (*case-tac* $x \leq aa$, *simp-all*)
done

lemma *IVT2*: [| $f(b::real) \leq (y::real)$; $y \leq f(a)$;
$a \leq b$;
$(\forall x.\ a \leq x\ \&\ x \leq b \longrightarrow isCont\ f\ x)$
|] ==> $\exists x.\ a \leq x\ \&\ x \leq b\ \&\ f(x) = y$
apply (*subgoal-tac* $-f\,a \leq -y\ \&\ -y \leq -f\,b$, *clarify*)
apply (*drule IVT* [**where** $f = \%x. -f\,x$], *assumption*)
apply (*auto intro*: *isCont-minus*)
done

lemma *IVT-objl*: $(f(a::real) \leq (y::real)\ \&\ y \leq f(b)\ \&\ a \leq b\ \&$
$(\forall x.\ a \leq x\ \&\ x \leq b \longrightarrow isCont\ f\ x))$
$\longrightarrow (\exists x.\ a \leq x\ \&\ x \leq b\ \&\ f(x) = y)$
apply (*blast intro*: *IVT*)
done

lemma *IVT2-objl*: $(f(b::real) \leq (y::real)\ \&\ y \leq f(a)\ \&\ a \leq b\ \&$
$(\forall x.\ a \leq x\ \&\ x \leq b \longrightarrow isCont\ f\ x))$
$\longrightarrow (\exists x.\ a \leq x\ \&\ x \leq b\ \&\ f(x) = y)$
apply (*blast intro*: *IVT2*)
done

60.5 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

lemma *isCont-bounded*:
[| $a \leq b$; $\forall x.\ a \leq x\ \&\ x \leq b \longrightarrow isCont\ f\ x$ |]
==> $\exists M::real.\ \forall x::real.\ a \leq x\ \&\ x \leq b \longrightarrow f(x) \leq M$
apply (*cut-tac* $P = \%\ (u,v)\ .\ a \leq u\ \&\ u \leq v\ \&\ v \leq b \longrightarrow (\exists M.\ \forall x.\ u \leq x\ \&$
$x \leq v \longrightarrow f\,x \leq M)$ **in** *lemma-BOLZANO2*)
apply *safe*
apply *simp-all*

apply (*rename-tac x xa ya M Ma*)
apply (*cut-tac x = M* **and** *y = Ma* **in** *linorder-linear*, *safe*)
apply (*rule-tac x = Ma* **in** *exI*, *clarify*)
apply (*cut-tac x = xb* **and** *y = xa* **in** *linorder-linear*, *force*)
apply (*rule-tac x = M* **in** *exI*, *clarify*)
apply (*cut-tac x = xb* **and** *y = xa* **in** *linorder-linear*, *force*)
apply (*case-tac a ≤ x & x ≤ b*)
apply (*rule-tac [2] x = 1* **in** *exI*)
prefer *2* **apply** *force*
apply (*simp add*: *LIM-def isCont-iff*)
apply (*drule-tac x = x* **in** *spec*, *auto*)
apply (*erule-tac V = ∀ M. ∃ x. a ≤ x & x ≤ b & ~ f x ≤ M* **in** *thin-rl*)
apply (*drule-tac x = 1* **in** *spec*, *auto*)
apply (*rule-tac x = s* **in** *exI*, *clarify*)
apply (*rule-tac x = |f x| + 1* **in** *exI*, *clarify*)
apply (*drule-tac x = xa−x* **in** *spec*)
apply (*auto simp add*: *abs-ge-self*)
done

Refine the above to existence of least upper bound

lemma *lemma-reals-complete*: ((∃ x. x ∈ S) & (∃ y. *isUb UNIV S* (y::*real*))) −−>
(∃ t. *isLub UNIV S t*)
by (*blast intro*: *reals-complete*)

lemma *isCont-has-Ub*: [| a ≤ b; ∀ x. a ≤ x & x ≤ b −−> *isCont f x* |]
==> ∃ M::*real*. (∀ x::*real*. a ≤ x & x ≤ b −−> f(x) ≤ M) &
(∀ N. N < M −−> (∃ x. a ≤ x & x ≤ b & N < f(x)))
apply (*cut-tac S = Collect* (%y. ∃ x. a ≤ x & x ≤ b & y = f x)
in *lemma-reals-complete*)
apply *auto*
apply (*drule isCont-bounded*, *assumption*)
apply (*auto simp add*: *isUb-def leastP-def isLub-def setge-def setle-def*)
apply (*rule exI*, *auto*)
apply (*auto dest*!: *spec simp add*: *linorder-not-less*)
done

Now show that it attains its upper bound

lemma *isCont-eq-Ub*:
assumes *le*: a ≤ b
and *con*: ∀ x::*real*. a ≤ x & x ≤ b −−> *isCont f x*
shows ∃ M::*real*. (∀ x. a ≤ x & x ≤ b −−> f(x) ≤ M) &
(∃ x. a ≤ x & x ≤ b & f(x) = M)
proof −
from *isCont-has-Ub* [*OF le con*]
obtain *M* **where** *M1*: ∀ x. a ≤ x ∧ x ≤ b ⟶ f x ≤ M
and *M2*: !!N. N<M ==> ∃ x. a ≤ x ∧ x ≤ b ∧ N < f x **by** *blast*
show *?thesis*
proof (*intro exI*, *intro conjI*)
show ∀ x. a ≤ x ∧ x ≤ b ⟶ f x ≤ M **by** (*rule M1*)

```
    show ∃x. a ≤ x ∧ x ≤ b ∧ f x = M
    proof (rule ccontr)
      assume ¬ (∃x. a ≤ x ∧ x ≤ b ∧ f x = M)
      with M1 have M3: ∀x. a ≤ x & x ≤ b --> f x < M
        by (fastsimp simp add: linorder-not-le [symmetric])
      hence ∀x. a ≤ x & x ≤ b --> isCont (%x. inverse (M - f x)) x
        by (auto simp add: isCont-inverse isCont-diff con)
      from isCont-bounded [OF le this]
     obtain k where k: !!x. a ≤ x & x ≤ b --> inverse (M - f x) ≤ k by auto
      have Minv: !!x. a ≤ x & x ≤ b --> 0 < inverse (M - f (x))
        by (simp add: M3 algebra-simps)
      have !!x. a ≤ x & x ≤ b --> inverse (M - f x) < k+1 using k
        by (auto intro: order-le-less-trans [of - k])
      with Minv
      have !!x. a ≤ x & x ≤ b --> inverse(k+1) < inverse(inverse(M - f x))
        by (intro strip less-imp-inverse-less, simp-all)
      hence invlt: !!x. a ≤ x & x ≤ b --> inverse(k+1) < M - f x
        by simp
      have M - inverse (k+1) < M using k [of a] Minv [of a] le
        by (simp, arith)
      from M2 [OF this]
      obtain x where ax: a ≤ x & x ≤ b & M - inverse(k+1) < f x ..
      thus False using invlt [of x] by force
    qed
  qed
qed
```

Same theorem for lower bound

```
lemma isCont-eq-Lb: [| a ≤ b; ∀x. a ≤ x & x ≤ b --> isCont f x |]
        ==> ∃M::real. (∀x::real. a ≤ x & x ≤ b --> M ≤ f(x)) &
                  (∃x. a ≤ x & x ≤ b & f(x) = M)
apply (subgoal-tac ∀x. a ≤ x & x ≤ b --> isCont (%x. - (f x)) x)
prefer 2 apply (blast intro: isCont-minus)
apply (drule-tac f = (%x. - (f x)) in isCont-eq-Ub)
apply safe
apply auto
done
```

Another version.

```
lemma isCont-Lb-Ub: [|a ≤ b; ∀x. a ≤ x & x ≤ b --> isCont f x |]
     ==> ∃L M::real. (∀x::real. a ≤ x & x ≤ b --> L ≤ f(x) & f(x) ≤ M) &
         (∀y. L ≤ y & y ≤ M --> (∃x. a ≤ x & x ≤ b & (f(x) = y)))
apply (frule isCont-eq-Lb)
apply (frule-tac [2] isCont-eq-Ub)
apply (assumption+, safe)
apply (rule-tac x = f x in exI)
apply (rule-tac x = f xa in exI, simp, safe)
apply (cut-tac x = x and y = xa in linorder-linear, safe)
apply (cut-tac f = f and a = x and b = xa and y = y in IVT-objl)
```

apply (*cut-tac* [*2*] $f = f$ **and** $a = xa$ **and** $b = x$ **and** $y = y$ **in** *IVT2-objl*, *safe*)
apply (*rule-tac* [*2*] $x = xb$ **in** *exI*)
apply (*rule-tac* [*4*] $x = xb$ **in** *exI*, *simp-all*)
done

60.6 Local extrema

If $(0::'a) < f'\ x$ then x is Locally Strictly Increasing At The Right

lemma *DERIV-left-inc*:
fixes f :: *real* => *real*
assumes *der*: *DERIV* $f\ x$:> l
and l: $0 < l$
shows $\exists\, d > 0.\ \forall\, h > 0.\ h < d$ --> $f(x) < f(x + h)$
proof −
from l *der* [*THEN DERIV-D*, *THEN LIM-D* [**where** $r = l$]]
have $\exists\, s > 0.\ (\forall\, z.\ z \neq 0 \wedge |z| < s \longrightarrow |(f(x+z) - f\ x) / z - l| < l)$
by (*simp add*: *diff-minus*)
then obtain s
where s: $0 < s$
and *all*: $!!z.\ z \neq 0 \wedge |z| < s \longrightarrow |(f(x+z) - f\ x) / z - l| < l$
by *auto*
thus *?thesis*
proof (*intro exI conjI strip*)
show $0<s$ **using** s **.**
fix h::*real*
assume $0 < h\ h < s$
with *all* [*of* h] **show** $f\ x < f\ (x+h)$
proof (*simp add*: *abs-if pos-less-divide-eq diff-minus* [*symmetric*]
split add: *split-if-asm*)
assume ~ $(f\ (x+h) - f\ x) / h < l$ **and** h: $0 < h$
with l
have $0 < (f\ (x+h) - f\ x) / h$ **by** *arith*
thus $f\ x < f\ (x+h)$
by (*simp add*: *pos-less-divide-eq h*)
qed
qed
qed

lemma *DERIV-left-dec*:
fixes f :: *real* => *real*
assumes *der*: *DERIV* $f\ x$:> l
and l: $l < 0$
shows $\exists\, d > 0.\ \forall\, h > 0.\ h < d$ --> $f(x) < f(x-h)$
proof −
from l *der* [*THEN DERIV-D*, *THEN LIM-D* [**where** $r = -l$]]
have $\exists\, s > 0.\ (\forall\, z.\ z \neq 0 \wedge |z| < s \longrightarrow |(f(x+z) - f\ x) / z - l| < -l)$
by (*simp add*: *diff-minus*)
then obtain s
where s: $0 < s$

```
        and all: !!z. z ≠ 0 ∧ |z| < s ⟶ |(f(x+z) − f x) / z − l| < −l
    by auto
  thus ?thesis
  proof (intro exI conjI strip)
    show 0<s using s .
    fix h::real
    assume 0 < h h < s
    with all [of −h] show f x < f (x−h)
    proof (simp add: abs-if pos-less-divide-eq diff-minus [symmetric]
    split add: split-if-asm)
      assume  − ((f (x−h) − f x) / h) < l and h: 0 < h
      with l
      have 0 < (f (x−h) − f x) / h by arith
      thus f x < f (x−h)
  by (simp add: pos-less-divide-eq h)
    qed
  qed
qed

lemma DERIV-local-max:
  fixes f :: real => real
  assumes der: DERIV f x :> l
      and d:   0 < d
      and le:  ∀ y. |x−y| < d −−> f(y) ≤ f(x)
  shows l = 0
proof (cases rule: linorder-cases [of l 0])
  case equal thus ?thesis .
next
  case less
  from DERIV-left-dec [OF der less]
  obtain d' where d': 0 < d'
          and lt: ∀ h > 0. h < d' ⟶ f x < f (x−h) by blast
  from real-lbound-gt-zero [OF d d']
  obtain e where 0 < e ∧ e < d ∧ e < d' ..
  with lt le [THEN spec [where x=x−e]]
  show ?thesis by (auto simp add: abs-if)
next
  case greater
  from DERIV-left-inc [OF der greater]
  obtain d' where d': 0 < d'
          and lt: ∀ h > 0. h < d' ⟶ f x < f (x + h) by blast
  from real-lbound-gt-zero [OF d d']
  obtain e where 0 < e ∧ e < d ∧ e < d' ..
  with lt le [THEN spec [where x=x+e]]
  show ?thesis by (auto simp add: abs-if)
qed
```

Similar theorem for a local minimum

```
lemma DERIV-local-min:
```

fixes f :: *real* => *real*
shows [| *DERIV f x* :> l; $0 < d$; $\forall y.\ |x-y| < d$ --> $f(x) \leq f(y)$ |] ==> $l = 0$
by (*drule DERIV-minus* [*THEN DERIV-local-max*], *auto*)

In particular, if a function is locally flat

lemma *DERIV-local-const*:
fixes f :: *real* => *real*
shows [| *DERIV f x* :> l; $0 < d$; $\forall y.\ |x-y| < d$ --> $f(x) = f(y)$ |] ==> $l = 0$
by (*auto dest*!: *DERIV-local-max*)

60.7 Rolle's Theorem

Lemma about introducing open ball in open interval

lemma *lemma-interval-lt*:
[| $a < x$; $x < b$ |]
==> $\exists d$::*real*. $0 < d$ & $(\forall y.\ |x-y| < d$ --> $a < y$ & $y < b)$

apply (*simp add*: *abs-less-iff*)
apply (*insert linorder-linear* [*of* $x-a$ $b-x$], *safe*)
apply (*rule-tac* $x = x-a$ **in** *exI*)
apply (*rule-tac* [2] $x = b-x$ **in** *exI*, *auto*)
done

lemma *lemma-interval*: [| $a < x$; $x < b$ |] ==>
$\exists d$::*real*. $0 < d$ & $(\forall y.\ |x-y| < d$ --> $a \leq y$ & $y \leq b)$
apply (*drule lemma-interval-lt*, *auto*)
apply (*auto intro*!: *exI*)
done

Rolle's Theorem. If f is defined and continuous on the closed interval $[a,b]$ and differentiable on the open interval (a,b), and $f\ a = f\ b$, then there exists $x0 \in (a,b)$ such that $f'\ x0 = (0::'a)$

theorem *Rolle*:
assumes *lt*: $a < b$
and *eq*: $f(a) = f(b)$
and *con*: $\forall x.\ a \leq x$ & $x \leq b$ --> *isCont f x*
and *dif* [*rule-format*]: $\forall x.\ a < x$ & $x < b$ --> *f differentiable x*
shows $\exists z$::*real*. $a < z$ & $z < b$ & *DERIV f z* :> 0
proof −
have *le*: $a \leq b$ **using** *lt* **by** *simp*
from *isCont-eq-Ub* [*OF le con*]
obtain x **where** *x-max*: $\forall z.\ a \leq z \wedge z \leq b \longrightarrow f\ z \leq f\ x$
and *alex*: $a \leq x$ **and** *xleb*: $x \leq b$
by *blast*
from *isCont-eq-Lb* [*OF le con*]
obtain x' **where** *x'-min*: $\forall z.\ a \leq z \wedge z \leq b \longrightarrow f\ x' \leq f\ z$

```
        and alex': a ≤ x' and x'leb: x' ≤ b
  by blast
show ?thesis
proof cases
  assume axb: a < x & x < b
    — f attains its maximum within the interval
  hence ax: a<x and xb: x<b by arith +
  from lemma-interval [OF ax xb]
  obtain d where d: 0<d and bound: ∀ y. |x−y| < d ⟶ a ≤ y ∧ y ≤ b
    by blast
  hence bound': ∀ y. |x−y| < d ⟶ f y ≤ f x using x-max
    by blast
  from differentiableD [OF dif [OF axb]]
  obtain l where der: DERIV f x :> l ..
  have l=0 by (rule DERIV-local-max [OF der d bound'])
    — the derivative at a local maximum is zero
  thus ?thesis using ax xb der by auto
next
  assume notaxb: ~ (a < x & x < b)
  hence xeqab: x=a | x=b using alex xleb by arith
  hence fb-eq-fx: f b = f x by (auto simp add: eq)
  show ?thesis
  proof cases
    assume ax'b: a < x' & x' < b
      — f attains its minimum within the interval
    hence ax': a<x' and x'b: x'<b by arith+
    from lemma-interval [OF ax' x'b]
    obtain d where d: 0<d and bound: ∀ y. |x'−y| < d ⟶ a ≤ y ∧ y ≤ b
by blast
    hence bound': ∀ y. |x'−y| < d ⟶ f x' ≤ f y using x'-min
by blast
    from differentiableD [OF dif [OF ax'b]]
    obtain l where der: DERIV f x' :> l ..
    have l=0 by (rule DERIV-local-min [OF der d bound'])
      — the derivative at a local minimum is zero
    thus ?thesis using ax' x'b der by auto
  next
    assume notax'b: ~ (a < x' & x' < b)
      — f is constant througout the interval
    hence x'eqab: x'=a | x'=b using alex' x'leb by arith
    hence fb-eq-fx': f b = f x' by (auto simp add: eq)
    from dense [OF lt]
    obtain r where ar: a < r and rb: r < b by blast
    from lemma-interval [OF ar rb]
    obtain d where d: 0<d and bound: ∀ y. |r−y| < d ⟶ a ≤ y ∧ y ≤ b
by blast
    have eq-fb: ∀ z. a ≤ z --> z ≤ b --> f z = f b
    proof (clarify)
      fix z::real
```

```
      assume az: a ≤ z and zb: z ≤ b
      show f z = f b
      proof (rule order-antisym)
        show f z ≤ f b by (simp add: fb-eq-fx x-max az zb)
        show f b ≤ f z by (simp add: fb-eq-fx' x'-min az zb)
      qed
    qed
    have bound': ∀ y. |r−y| < d ⟶ f r = f y
    proof (intro strip)
      fix y::real
      assume lt: |r−y| < d
      hence f y = f b by (simp add: eq-fb bound)
      thus f r = f y by (simp add: eq-fb ar rb order-less-imp-le)
    qed
    from differentiableD [OF dif [OF conjI [OF ar rb]]]
    obtain l where der: DERIV f r :> l ..
    have l=0 by (rule DERIV-local-const [OF der d bound'])
      — the derivative of a constant function is zero
    thus ?thesis using ar rb der by auto
  qed
 qed
qed
```

60.8 Mean Value Theorem

```
lemma lemma-MVT:
    f a − (f b − f a)/(b−a) * a = f b − (f b − f a)/(b−a) * (b::real)
proof cases
 assume a=b thus ?thesis by simp
next
 assume a≠b
 hence ba: b−a ≠ 0 by arith
 show ?thesis
   by (rule real-mult-left-cancel [OF ba, THEN iffD1],
       simp add: right-diff-distrib,
       simp add: left-diff-distrib)
qed

theorem MVT:
 assumes lt:  a < b
     and con: ∀ x. a ≤ x & x ≤ b −−> isCont f x
     and dif [rule-format]: ∀ x. a < x & x < b −−> f differentiable x
 shows ∃ l z::real. a < z & z < b & DERIV f z :> l &
                    (f(b) − f(a) = (b−a) * l)
proof −
 let ?F = %x. f x − ((f b − f a) / (b−a)) * x
 have contF: ∀ x. a ≤ x ∧ x ≤ b ⟶ isCont ?F x using con
   by (fast intro: isCont-diff isCont-const isCont-mult isCont-ident)
 have difF: ∀ x. a < x ∧ x < b ⟶ ?F differentiable x
```

```
  proof (clarify)
    fix x::real
    assume ax: a < x and xb: x < b
    from differentiableD [OF dif [OF conjI [OF ax xb]]]
    obtain l where der: DERIV f x :> l ..
    show ?F differentiable x
      by (rule differentiableI [where D = l - (f b - f a)/(b-a)],
          blast intro: DERIV-diff DERIV-cmult-Id der)
  qed
  from Rolle [where f = ?F, OF lt lemma-MVT contF difF]
  obtain z where az: a < z and zb: z < b and der: DERIV ?F z :> 0
    by blast
  have DERIV (%x. ((f b - f a)/(b-a)) * x) z :> (f b - f a)/(b-a)
    by (rule DERIV-cmult-Id)
  hence derF: DERIV (λx. ?F x + (f b - f a) / (b - a) * x) z
                :> 0 + (f b - f a) / (b - a)
    by (rule DERIV-add [OF der])
  show ?thesis
  proof (intro exI conjI)
    show a < z using az .
    show z < b using zb .
    show f b - f a = (b - a) * ((f b - f a)/(b-a)) by (simp)
    show DERIV f z :> ((f b - f a)/(b-a))  using derF by simp
  qed
qed

lemma MVT2:
     [| a < b; ∀ x. a ≤ x & x ≤ b --> DERIV f x :> f'(x) |]
      ==> ∃ z::real. a < z & z < b & (f b - f a = (b - a) * f'(z))
apply (drule MVT)
apply (blast intro: DERIV-isCont)
apply (force dest: order-less-imp-le simp add: differentiable-def)
apply (blast dest: DERIV-unique order-less-imp-le)
done
```

A function is constant if its derivative is 0 over an interval.

```
lemma DERIV-isconst-end:
  fixes f :: real => real
  shows [| a < b;
        ∀ x. a ≤ x & x ≤ b --> isCont f x;
        ∀ x. a < x & x < b --> DERIV f x :> 0 |]
       ==> f b = f a
apply (drule MVT, assumption)
apply (blast intro: differentiableI)
apply (auto dest!: DERIV-unique simp add: diff-eq-eq)
done

lemma DERIV-isconst1:
  fixes f :: real => real
```

shows [| $a < b$;
$\forall x.\ a \leq x\ \&\ x \leq b \longrightarrow isCont\ f\ x$;
$\forall x.\ a < x\ \&\ x < b \longrightarrow DERIV\ f\ x :> 0$ |]
$\Longrightarrow \forall x.\ a \leq x\ \&\ x \leq b \longrightarrow f\ x = f\ a$
apply *safe*
apply (*drule-tac* $x = a$ **in** *order-le-imp-less-or-eq*, *safe*)
apply (*drule-tac* $b = x$ **in** *DERIV-isconst-end*, *auto*)
done

lemma *DERIV-isconst2*:
fixes f :: *real* => *real*
shows [| $a < b$;
$\forall x.\ a \leq x\ \&\ x \leq b \longrightarrow isCont\ f\ x$;
$\forall x.\ a < x\ \&\ x < b \longrightarrow DERIV\ f\ x :> 0$;
$a \leq x$; $x \leq b$ |]
$\Longrightarrow f\ x = f\ a$
apply (*blast dest*: *DERIV-isconst1*)
done

lemma *DERIV-isconst3*: **fixes** $a\ b\ x\ y$ :: *real*
assumes $a < b$ **and** $x \in \{a <..< b\}$ **and** $y \in \{a <..< b\}$
assumes *derivable*: $\bigwedge x.\ x \in \{a <..< b\} \Longrightarrow DERIV\ f\ x :> 0$
shows $f\ x = f\ y$
proof (*cases* $x = y$)
case *False*
let *?a* = *min x y*
let *?b* = *max x y*

have $\forall z.\ ?a \leq z \wedge z \leq ?b \longrightarrow DERIV\ f\ z :> 0$
proof (*rule allI*, *rule impI*)
fix z :: *real* **assume** $?a \leq z \wedge z \leq ?b$
hence $a < z$ **and** $z < b$ **using** ‹$x \in \{a <..< b\}$› **and** ‹$y \in \{a <..< b\}$› **by** *auto*
hence $z \in \{a<..<b\}$ **by** *auto*
thus $DERIV\ f\ z :> 0$ **by** (*rule derivable*)
qed
hence *isCont*: $\forall z.\ ?a \leq z \wedge z \leq ?b \longrightarrow isCont\ f\ z$
and *DERIV*: $\forall z.\ ?a < z \wedge z < ?b \longrightarrow DERIV\ f\ z :> 0$ **using** *DERIV-isCont* **by** *auto*

have $?a < ?b$ **using** ‹$x \neq y$› **by** *auto*
from *DERIV-isconst2*[*OF this isCont DERIV*, *of x*] **and** *DERIV-isconst2*[*OF this isCont DERIV*, *of y*]
show *?thesis* **by** *auto*
qed *auto*

lemma *DERIV-isconst-all*:
fixes f :: *real* => *real*
shows $\forall x.\ DERIV\ f\ x :> 0 \Longrightarrow f(x) = f(y)$

apply (*rule linorder-cases* [*of x y*])
apply (*blast intro*: *sym DERIV-isCont DERIV-isconst-end*)+
done

lemma *DERIV-const-ratio-const*:
  fixes *f* :: *real* => *real*
  shows [|*a* ≠ *b*; ∀ *x*. *DERIV f x* :> *k* |] ==> (*f*(*b*) − *f*(*a*)) = (*b*−*a*) ∗ *k*
apply (*rule linorder-cases* [*of a b*], *auto*)
apply (*drule-tac* [!] *f* = *f* **in** *MVT*)
apply (*auto dest*: *DERIV-isCont DERIV-unique simp add*: *differentiable-def*)
apply (*auto dest*: *DERIV-unique simp add*: *ring-distribs diff-minus*)
done

lemma *DERIV-const-ratio-const2*:
  fixes *f* :: *real* => *real*
  shows [|*a* ≠ *b*; ∀ *x*. *DERIV f x* :> *k* |] ==> (*f*(*b*) − *f*(*a*))/(*b*−*a*) = *k*
apply (*rule-tac c1* = *b*−*a* **in** *real-mult-right-cancel* [*THEN iffD1*])
apply (*auto dest*!: *DERIV-const-ratio-const simp add*: *mult-assoc*)
done

lemma *real-average-minus-first* [*simp*]: ((*a* + *b*) /*2* − *a*) = (*b*−*a*)/(*2*::*real*)
by (*simp*)

lemma *real-average-minus-second* [*simp*]: ((*b* + *a*)/*2* − *a*) = (*b*−*a*)/(*2*::*real*)
by (*simp*)

Gallileo's "trick": average velocity = av. of end velocities

lemma *DERIV-const-average*:
  fixes *v* :: *real* => *real*
  assumes *neq*: *a* ≠ (*b*::*real*)
    and *der*: ∀ *x*. *DERIV v x* :> *k*
  shows *v* ((*a* + *b*)/*2*) = (*v a* + *v b*)/*2*
proof (*cases rule*: *linorder-cases* [*of a b*])
  case *equal* **with** *neq* **show** *?thesis* **by** *simp*
next
  case *less*
  have (*v b* − *v a*) / (*b* − *a*) = *k*
    by (*rule DERIV-const-ratio-const2* [*OF neq der*])
  hence (*b*−*a*) ∗ ((*v b* − *v a*) / (*b*−*a*)) = (*b*−*a*) ∗ *k* **by** *simp*
  moreover have (*v* ((*a* + *b*) / *2*) − *v a*) / ((*a* + *b*) / *2* − *a*) = *k*
    by (*rule DERIV-const-ratio-const2* [*OF - der*], *simp add*: *neq*)
  ultimately show *?thesis* **using** *neq* **by** *force*
next
  case *greater*
  have (*v b* − *v a*) / (*b* − *a*) = *k*
    by (*rule DERIV-const-ratio-const2* [*OF neq der*])
  hence (*b*−*a*) ∗ ((*v b* − *v a*) / (*b*−*a*)) = (*b*−*a*) ∗ *k* **by** *simp*
  moreover have (*v* ((*b* + *a*) / *2*) − *v a*) / ((*b* + *a*) / *2* − *a*) = *k*
    by (*rule DERIV-const-ratio-const2* [*OF - der*], *simp add*: *neq*)

ultimately show *?thesis* **using** *neq* **by** (*force simp add*: *add-commute*)
qed

60.9 Continuous injective functions

Dull lemma: an continuous injection on an interval must have a strict maximum at an end point, not in the middle.

lemma *lemma-isCont-inj*:
fixes f :: *real* $\Rightarrow$ *real*
assumes *d*: $0 < d$
and *inj* [*rule-format*]: $\forall z.\ |z-x| \leq d$ −−> $g(f\ z) = z$
and *cont*: $\forall z.\ |z-x| \leq d$ −−> *isCont f z*
shows $\exists z.\ |z-x| \leq d\ \&\ f\ x < f\ z$
proof (*rule ccontr*)
assume ~ $(\exists z.\ |z-x| \leq d\ \&\ f\ x < f\ z)$
hence *all* [*rule-format*]: $\forall z.\ |z - x| \leq d$ −−> $f\ z \leq f\ x$ **by** *auto*
show *False*
proof (*cases rule*: *linorder-le-cases* [*of* $f(x-d)$ $f(x+d)$])
case *le*
from *d cont all* [*of* $x+d$]
have *flef*: $f(x+d) \leq f\ x$
and *xlex*: $x - d \leq x$
and *cont'*: $\forall z.\ x - d \leq z \wedge z \leq x \longrightarrow$ *isCont f z*
by (*auto simp add*: *abs-if*)
from *IVT* [*OF le flef xlex cont'*]
obtain x' **where** $x-d \leq x'\ x' \leq x\ f\ x' = f(x+d)$ **by** *blast*
moreover
hence $g(f\ x') = g\ (f(x+d))$ **by** *simp*
ultimately show *False* **using** *d inj* [*of x'*] *inj* [*of* $x+d$]
by (*simp add*: *abs-le-iff*)
next
case *ge*
from *d cont all* [*of* $x-d$]
have *flef*: $f(x-d) \leq f\ x$
and *xlex*: $x \leq x+d$
and *cont'*: $\forall z.\ x \leq z \wedge z \leq x+d \longrightarrow$ *isCont f z*
by (*auto simp add*: *abs-if*)
from *IVT2* [*OF ge flef xlex cont'*]
obtain x' **where** $x \leq x'\ x' \leq x+d\ f\ x' = f(x-d)$ **by** *blast*
moreover
hence $g(f\ x') = g\ (f(x-d))$ **by** *simp*
ultimately show *False* **using** *d inj* [*of x'*] *inj* [*of* $x-d$]
by (*simp add*: *abs-le-iff*)
qed
qed

Similar version for lower bound.

lemma *lemma-isCont-inj2*:

fixes $f\ g :: real \Rightarrow real$
shows $[|0 < d; \forall z.\ |z-x| \leq d \longrightarrow g(f\ z) = z;$
$\forall z.\ |z-x| \leq d \longrightarrow isCont\ f\ z\ |]$
$\Longrightarrow \exists z.\ |z-x| \leq d\ \&\ f\ z < f\ x$
apply (*insert lemma-isCont-inj*
[**where** $f = \%x.\ -f\ x$ **and** $g = \%y.\ g(-y)$ **and** $x = x$ **and** $d = d$])
apply (*simp add*: *isCont-minus linorder-not-le*)
done

Show there's an interval surrounding $f\ x$ in $f[[x - d, x + d]]$.

lemma *isCont-inj-range*:
fixes $f :: real \Rightarrow real$
assumes *d*: $0 < d$
and *inj*: $\forall z.\ |z-x| \leq d \longrightarrow g(f\ z) = z$
and *cont*: $\forall z.\ |z-x| \leq d \longrightarrow isCont\ f\ z$
shows $\exists e{>}0.\ \forall y.\ |y - f\ x| \leq e \longrightarrow (\exists z.\ |z-x| \leq d\ \&\ f\ z = y)$
proof −
have $x-d \leq x+d\ \forall z.\ x-d \leq z \wedge z \leq x+d \longrightarrow isCont\ f\ z$ **using** *cont d*
by (*auto simp add*: *abs-le-iff*)
from *isCont-Lb-Ub* [*OF this*]
obtain *L M*
where *all1* [*rule-format*]: $\forall z.\ x-d \leq z \wedge z \leq x+d \longrightarrow L \leq f\ z \wedge f\ z \leq M$
and *all2* [*rule-format*]:
$\forall y.\ L \leq y \wedge y \leq M \longrightarrow (\exists z.\ x-d \leq z \wedge z \leq x+d \wedge f\ z = y)$
by *auto*
with *d* **have** $L \leq f\ x\ \&\ f\ x \leq M$ **by** *simp*
moreover have $L \neq f\ x$
proof −
from *lemma-isCont-inj2* [*OF d inj cont*]
obtain *u* **where** $|u - x| \leq d\ f\ u < f\ x$ **by** *auto*
thus *?thesis* **using** *all1* [*of u*] **by** *arith*
qed
moreover have $f\ x \neq M$
proof −
from *lemma-isCont-inj* [*OF d inj cont*]
obtain *u* **where** $|u - x| \leq d\ f\ x < f\ u$ **by** *auto*
thus *?thesis* **using** *all1* [*of u*] **by** *arith*
qed
ultimately have $L < f\ x\ \&\ f\ x < M$ **by** *arith*
hence $0 < f\ x - L\ 0 < M - f\ x$ **by** *arith+*
from *real-lbound-gt-zero* [*OF this*]
obtain *e* **where** *e*: $0 < e\ e < f\ x - L\ e < M - f\ x$ **by** *auto*
thus *?thesis*
proof (*intro exI conjI*)
show $0{<}e$ **using** *e*(*1*) **.**
show $\forall y.\ |y - f\ x| \leq e \longrightarrow (\exists z.\ |z - x| \leq d \wedge f\ z = y)$
proof (*intro strip*)
fix *y*::*real*
assume $|y - f\ x| \leq e$

```
    with e have L ≤ y ∧ y ≤ M by arith
    from all2 [OF this]
    obtain z where x − d ≤ z z ≤ x + d f z = y by blast
    thus ∃ z. |z − x| ≤ d ∧ f z = y
      by (force simp add: abs-le-iff)
  qed
 qed
qed
```

Continuity of inverse function

```
lemma isCont-inverse-function:
 fixes f g :: real ⇒ real
 assumes d: 0 < d
    and inj: ∀ z. |z−x| ≤ d −−> g(f z) = z
    and cont: ∀ z. |z−x| ≤ d −−> isCont f z
 shows isCont g (f x)
proof (simp add: isCont-iff LIM-eq)
 show ∀ r. 0 < r ⟶
       (∃ s>0. ∀ z. z≠0 ∧ |z| < s ⟶ |g(f x + z) − g(f x)| < r)
 proof (intro strip)
   fix r::real
   assume r: 0<r
   from real-lbound-gt-zero [OF r d]
   obtain e where e: 0 < e and e-lt: e < r ∧ e < d by blast
   with inj cont
   have e-simps: ∀ z. |z−x| ≤ e −−> g (f z) = z
                 ∀ z. |z−x| ≤ e −−> isCont f z   by auto
   from isCont-inj-range [OF e this]
   obtain e′ where e′: 0 < e′
      and all: ∀ y. |y − f x| ≤ e′ ⟶ (∃ z. |z − x| ≤ e ∧ f z = y)
       by blast
   show ∃ s>0. ∀ z. z≠0 ∧ |z| < s ⟶ |g(f x + z) − g(f x)| < r
   proof (intro exI conjI)
     show 0<e′ using e′ .
     show ∀ z. z ≠ 0 ∧ |z| < e′ ⟶ |g (f x + z) − g (f x)| < r
     proof (intro strip)
       fix z::real
       assume z: z ≠ 0 ∧ |z| < e′
       with e e-lt e-simps all [rule-format, of f x + z]
       show |g (f x + z) − g (f x)| < r by force
     qed
   qed
 qed
qed
```

Derivative of inverse function

```
lemma DERIV-inverse-function:
 fixes f g :: real ⇒ real
 assumes der: DERIV f (g x) :> D
```

assumes *neq*: $D \neq 0$
assumes *a*: $a < x$ **and** *b*: $x < b$
assumes *inj*: $\forall y.\ a < y \wedge y < b \longrightarrow f\ (g\ y) = y$
assumes *cont*: *isCont g x*
shows *DERIV g x :> inverse D*
unfolding *DERIV-iff2*
proof (*rule LIM-equal2*)
show $0 < min\ (x - a)\ (b - x)$
using *a b* **by** *arith*
next
fix *y*
assume *norm* $(y - x) < min\ (x - a)\ (b - x)$
hence $a < y$ **and** $y < b$
by (*simp-all add*: *abs-less-iff*)
thus $(g\ y - g\ x)\ /\ (y - x) =$
inverse $((f\ (g\ y) - x)\ /\ (g\ y - g\ x))$
by (*simp add*: *inj*)
next
have $(\lambda z.\ (f\ z - f\ (g\ x))\ /\ (z - g\ x)) -- g\ x --> D$
by (*rule der* [*unfolded DERIV-iff2*])
hence *1*: $(\lambda z.\ (f\ z - x)\ /\ (z - g\ x)) -- g\ x --> D$
using *inj a b* **by** *simp*
have *2*: $\exists\, d{>}0.\ \forall y.\ y \neq x \wedge norm\ (y - x) < d \longrightarrow g\ y \neq g\ x$
proof (*safe intro*!: *exI*)
show $0 < min\ (x - a)\ (b - x)$
using *a b* **by** *simp*
next
fix *y*
assume *norm* $(y - x) < min\ (x - a)\ (b - x)$
hence *y*: $a < y\ y < b$
by (*simp-all add*: *abs-less-iff*)
assume $g\ y = g\ x$
hence $f\ (g\ y) = f\ (g\ x)$ **by** *simp*
hence $y = x$ **using** *inj y a b* **by** *simp*
also assume $y \neq x$
finally show *False* **by** *simp*
qed
have $(\lambda y.\ (f\ (g\ y) - x)\ /\ (g\ y - g\ x)) -- x --> D$
using *cont 1 2* **by** (*rule isCont-LIM-compose2*)
thus $(\lambda y.\ inverse\ ((f\ (g\ y) - x)\ /\ (g\ y - g\ x)))$
$-- x --> inverse\ D$
using *neq* **by** (*rule LIM-inverse*)
qed

60.10 Generalized Mean Value Theorem

theorem *GMVT*:
fixes *a b* :: *real*
assumes *alb*: $a < b$

and *fc*: $\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x$
and *fd*: $\forall x.\ a < x \wedge x < b \longrightarrow f\ differentiable\ x$
and *gc*: $\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ g\ x$
and *gd*: $\forall x.\ a < x \wedge x < b \longrightarrow g\ differentiable\ x$
shows $\exists g'c\ f'c\ c.\ DERIV\ g\ c :> g'c \wedge DERIV\ f\ c :> f'c \wedge a < c \wedge c < b \wedge ((f\ b - f\ a) * g'c) = ((g\ b - g\ a) * f'c)$
proof −
let *?h* = $\lambda x.\ (f\ b - f\ a)*(g\ x) - (g\ b - g\ a)*(f\ x)$
from *prems* **have** $a < b$ **by** *simp*
moreover have $\forall x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ ?h\ x$
proof −
have $\forall x.\ a <= x \wedge x <= b \longrightarrow isCont\ (\lambda x.\ f\ b - f\ a)\ x$ **by** *simp*
with *gc* **have** $\forall x.\ a <= x \wedge x <= b \longrightarrow isCont\ (\lambda x.\ (f\ b - f\ a) * g\ x)\ x$
by (*auto intro*: *isCont-mult*)
moreover
have $\forall x.\ a <= x \wedge x <= b \longrightarrow isCont\ (\lambda x.\ g\ b - g\ a)\ x$ **by** *simp*
with *fc* **have** $\forall x.\ a <= x \wedge x <= b \longrightarrow isCont\ (\lambda x.\ (g\ b - g\ a) * f\ x)\ x$
by (*auto intro*: *isCont-mult*)
ultimately show *?thesis*
by (*fastsimp intro*: *isCont-diff*)
qed
moreover
have $\forall x.\ a < x \wedge x < b \longrightarrow ?h\ differentiable\ x$
proof −
have $\forall x.\ a < x \wedge x < b \longrightarrow (\lambda x.\ f\ b - f\ a)\ differentiable\ x$ **by** (*simp add*: *differentiable-const*)
with *gd* **have** $\forall x.\ a < x \wedge x < b \longrightarrow (\lambda x.\ (f\ b - f\ a) * g\ x)\ differentiable\ x$ **by** (*simp add*: *differentiable-mult*)
moreover
have $\forall x.\ a < x \wedge x < b \longrightarrow (\lambda x.\ g\ b - g\ a)\ differentiable\ x$ **by** (*simp add*: *differentiable-const*)
with *fd* **have** $\forall x.\ a < x \wedge x < b \longrightarrow (\lambda x.\ (g\ b - g\ a) * f\ x)\ differentiable\ x$ **by** (*simp add*: *differentiable-mult*)
ultimately show *?thesis* **by** (*simp add*: *differentiable-diff*)
qed
ultimately have $\exists l\ z.\ a < z \wedge z < b \wedge DERIV\ ?h\ z :> l \wedge ?h\ b - ?h\ a = (b - a) * l$ **by** (*rule MVT*)
then obtain *l* **where** *ldef*: $\exists z.\ a < z \wedge z < b \wedge DERIV\ ?h\ z :> l \wedge ?h\ b - ?h\ a = (b - a) * l$ **..**
then obtain *c* **where** *cdef*: $a < c \wedge c < b \wedge DERIV\ ?h\ c :> l \wedge ?h\ b - ?h\ a = (b - a) * l$ **..**

from *cdef* **have** *cint*: $a < c \wedge c < b$ **by** *auto*
with *gd* **have** *g differentiable c* **by** *simp*
hence $\exists D.\ DERIV\ g\ c :> D$ **by** (*rule differentiableD*)
then obtain $g'c$ **where** $g'cdef$: $DERIV\ g\ c :> g'c$ **..**

from *cdef* **have** $a < c \wedge c < b$ **by** *auto*
with *fd* **have** *f differentiable c* **by** *simp*

hence $\exists D.\ DERIV\ f\ c :> D$ **by** (*rule differentiableD*)
then obtain $f'c$ **where** *f'cdef*: $DERIV\ f\ c :> f'c$ **..**

from *cdef* **have** $DERIV\ ?h\ c :> l$ **by** *auto*
moreover
{
have $DERIV\ (\lambda x.\ (f\ b - f\ a) * g\ x)\ c :> g'c * (f\ b - f\ a)$
apply (*insert DERIV-const* [**where** $k=f\ b - f\ a$])
apply (*drule meta-spec* [*of* - *c*])
apply (*drule DERIV-mult* [*OF* - *g'cdef*])
by *simp*
moreover have $DERIV\ (\lambda x.\ (g\ b - g\ a) * f\ x)\ c :> f'c * (g\ b - g\ a)$
apply (*insert DERIV-const* [**where** $k=g\ b - g\ a$])
apply (*drule meta-spec* [*of* - *c*])
apply (*drule DERIV-mult* [*OF* - *f'cdef*])
by *simp*
ultimately have $DERIV\ ?h\ c :>\ g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a)$
by (*simp add*: *DERIV-diff*)
}
ultimately have *leq*: $l =\ g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a)$ **by** (*rule DERIV-unique*)

{
from *cdef* **have** $?h\ b - ?h\ a = (b - a) * l$ **by** *auto*
also with *leq* **have** $\ldots = (b - a) * (g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a))$ **by** *simp*
finally have $?h\ b - ?h\ a = (b - a) * (g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a))$ **by** *simp*
}
moreover
{
have $?h\ b - ?h\ a =$
$((f\ b)*(g\ b) - (f\ a)*(g\ b) - (g\ b)*(f\ b) + (g\ a)*(f\ b)) -$
$((f\ b)*(g\ a) - (f\ a)*(g\ a) - (g\ b)*(f\ a) + (g\ a)*(f\ a))$
by (*simp add*: *algebra-simps*)
hence $?h\ b - ?h\ a = 0$ **by** *auto*
}
ultimately have $(b - a) * (g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a)) = 0$ **by** *auto*
with *alb* **have** $g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a) = 0$ **by** *simp*
hence $g'c * (f\ b - f\ a) = f'c * (g\ b - g\ a)$ **by** *simp*
hence $(f\ b - f\ a) * g'c = (g\ b - g\ a) * f'c$ **by** (*simp add*: *mult-ac*)

with *g'cdef f'cdef cint* **show** *?thesis* **by** *auto*
qed

60.11 Theorems about Limits

lemma *isCont-inv-fun*:
fixes $f\ g :: real \Rightarrow real$

shows [| $0 < d$; $\forall z.\ |z - x| \leq d \longrightarrow g(f(z)) = z$;
$\forall z.\ |z - x| \leq d \longrightarrow isCont\ f\ z$ |]
$\Longrightarrow isCont\ g\ (f\ x)$
by (*rule isCont-inverse-function*)

lemma *isCont-inv-fun-inv*:
fixes $f\ g :: real \Rightarrow real$
shows [| $0 < d$;
$\forall z.\ |z - x| \leq d \longrightarrow g(f(z)) = z$;
$\forall z.\ |z - x| \leq d \longrightarrow isCont\ f\ z$ |]
$\Longrightarrow \exists e.\ 0 < e\ \&$
$(\forall y.\ 0 < |y - f(x)|\ \&\ |y - f(x)| < e \longrightarrow f(g(y)) = y)$
apply (*drule isCont-inj-range*)
prefer *2* **apply** (*assumption*, *assumption*, *auto*)
apply (*rule-tac* $x = e$ **in** *exI*, *auto*)
apply (*rotate-tac 2*)
apply (*drule-tac* $x = y$ **in** *spec*, *auto*)
done

Bartle/Sherbert: Introduction to Real Analysis, Theorem 4.2.9, p. 110

lemma *LIM-fun-gt-zero*:
[| f *–– c ––>* (l::*real*); $0 < l$ |]
$\Longrightarrow \exists r.\ 0 < r\ \&\ (\forall x{::}real.\ x \neq c\ \&\ |c - x| < r \longrightarrow 0 < f\ x)$
apply (*auto simp add*: *LIM-def*)
apply (*drule-tac* $x = l/2$ **in** *spec*, *safe*, *force*)
apply (*rule-tac* $x = s$ **in** *exI*)
apply (*auto simp only*: *abs-less-iff*)
done

lemma *LIM-fun-less-zero*:
[| f *–– c ––>* (l::*real*); $l < 0$ |]
$\Longrightarrow \exists r.\ 0 < r\ \&\ (\forall x{::}real.\ x \neq c\ \&\ |c - x| < r \longrightarrow f\ x < 0)$
apply (*auto simp add*: *LIM-def*)
apply (*drule-tac* $x = -l/2$ **in** *spec*, *safe*, *force*)
apply (*rule-tac* $x = s$ **in** *exI*)
apply (*auto simp only*: *abs-less-iff*)
done

lemma *LIM-fun-not-zero*:
[| f *–– c ––>* (l::*real*); $l \neq 0$ |]
$\Longrightarrow \exists r.\ 0 < r\ \&\ (\forall x{::}real.\ x \neq c\ \&\ |c - x| < r \longrightarrow f\ x \neq 0)$
apply (*cut-tac* $x = l$ **and** $y = 0$ **in** *linorder-less-linear*, *auto*)
apply (*drule LIM-fun-less-zero*)
apply (*drule-tac* [*3*] *LIM-fun-gt-zero*)
apply *force+*
done

end

61 Parity: Even and Odd for int and nat

theory *Parity*
imports *Main*
begin

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

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

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

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

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

instance ..

end

61.1 Even and odd are mutually exclusive

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

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

61.2 Behavior under integer arithmetic operations

declare *dvd-def*[*algebra*]
lemma *nat-even-iff-2-dvd*[*algebra*]: *even* (*x*::*nat*) ⟷ *2 dvd x*
  by (*presburger add*: *even-nat-def even-def*)
lemma *int-even-iff-2-dvd*[*algebra*]: *even* (*x*::*int*) ⟷ *2 dvd x*
  by *presburger*

lemma *even-times-anything*: *even* (*x*::*int*) ==> *even* (*x* * *y*)
  by *algebra*

lemma *anything-times-even*: *even* (*y*::*int*) ==> *even* (*x* * *y*) **by** *algebra*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

61.3 Equivalent definitions

lemma *two-times-even-div-two*: *even* (*x*::*int*) ==> *2* * (*x div 2*) = *x* **by** *presburger*

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

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

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

61.4 even and odd for nats

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

lemma *even-nat-product*[*presburger*, *algebra*]: *even*((*x*::*nat*) * *y*) = (*even x* | *even y*) **by** (*simp add*: *even-nat-def int-mult*)

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

lemma *even-nat-difference*[*presburger*, *algebra*]: *even* ((*x*::*nat*) − *y*) = (*x* < *y* | (*even x* & *even y*) | (*odd x* & *odd y*)) **by** *presburger*

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

lemma *even-nat-power*[*presburger*, *algebra*]: *even* ((*x*::*nat*) ^ *y*) = (*even x* & *0* < *y*) **by** (*simp add*: *even-nat-def int-power*)

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

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

61.5 Equivalent definitions

lemma *nat-lt-two-imp-zero-or-one*: (*x*::*nat*) < *Suc* (*Suc 0*) ==>

$x = 0 \mid x = Suc\ 0$ **by** *presburger*

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

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

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

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

lemma *even-nat-div-two-times-two*: $even\ (x::nat) \Longrightarrow$
$Suc\ (Suc\ 0) * (x\ div\ Suc\ (Suc\ 0)) = x$ **by** *presburger*

lemma *odd-nat-div-two-times-two-plus-one*: $odd\ (x::nat) \Longrightarrow$
$Suc(\ Suc\ (Suc\ 0) * (x\ div\ Suc\ (Suc\ 0))) = x$ **by** *presburger*

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

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

61.6 Parity and powers

lemma *minus-one-even-odd-power*:
$(even\ x \longrightarrow (-\ 1::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ x = 1)\ \&$
$(odd\ x \longrightarrow (-\ 1::'a)\ \hat{}\ x = -\ 1)$
apply (*induct x*)
apply (*rule conjI*)
apply *simp*
apply (*insert even-nat-zero*, *blast*)
apply (*simp add*: *power-Suc*)
done

lemma *minus-one-even-power* [*simp*]:
$even\ x \Longrightarrow (-\ 1::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ x = 1$
using *minus-one-even-odd-power* **by** *blast*

lemma *minus-one-odd-power* [*simp*]:
$odd\ x \Longrightarrow (-\ 1::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ x = -\ 1$
using *minus-one-even-odd-power* **by** *blast*

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

apply (*induct x*)
apply (*simp*, *simp add*: *power-Suc*)
done

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

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

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

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

lemma *zero-le-odd-power*: *odd n* ==>
(*0* <= (*x*::'*a*::{*recpower*,*ordered-idom*}) ^ *n*) = (*0* <= *x*)
apply (*auto simp*: *odd-nat-equiv-def2 power-Suc power-add zero-le-mult-iff*)
apply (*metis field-power-not-zero no-zero-divirors-neq0 order-antisym-conv zero-le-square*)
done

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

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

unfolding *order-less-le zero-le-power-eq* **by** *auto*

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

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

lemma *power-even-abs*: *even n* ==>
(*abs* (*x*::'*a*::{*recpower*,*ordered-idom*})) ^*n* = *x* ^*n*
apply (*subst power-abs* [*symmetric*])
apply (*simp add*: *zero-le-even-power*)
done

lemma *zero-less-power-nat-eq*[*presburger*]: (*0* < (*x*::*nat*) ^ *n*) = (*n* = *0* | *0* < *x*)
by (*induct n*) *auto*

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

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

lemma *power-mono-even*: **fixes** *x y* :: '*a* :: {*recpower*, *ordered-idom*}
assumes *even n* **and** |*x*| ≤ |*y*|
shows *x* ^*n* ≤ *y* ^*n*
proof −
have *0* ≤ |*x*| **by** *auto*
with ‹|*x*| ≤ |*y*|›
have |*x*| ^*n* ≤ |*y*| ^*n* **by** (*rule power-mono*)
thus *?thesis* **unfolding** *power-even-abs*[*OF* ‹*even n*›] **.**
qed

lemma *odd-pos*: *odd* (*n*::*nat*) ⟹ *0* < *n* **by** *presburger*

lemma *power-mono-odd*: **fixes** x y :: $'a$ :: {*recpower*, *ordered-idom*}
assumes *odd n* **and** $x \leq y$
shows $x\hat{}n \leq y\hat{}n$
proof (*cases* $y < 0$)
case *True* **with** ‹$x \leq y$› **have** $-y \leq -x$ **and** $0 \leq -y$ **by** *auto*
hence $(-y)\hat{}n \leq (-x)\hat{}n$ **by** (*rule power-mono*)
thus *?thesis* **unfolding** *power-minus-odd*[*OF* ‹*odd n*›] **by** *auto*
next
case *False*
show *?thesis*
proof (*cases* $x < 0$)
case *True* **hence** $n \neq 0$ **and** $x \leq 0$ **using** ‹*odd n*›[*THEN odd-pos*] **by** *auto*
hence $x\hat{}n \leq 0$ **unfolding** *power-le-zero-eq* **using** ‹*odd n*› **by** *auto*
moreover
from ‹$\neg$ $y < 0$› **have** $0 \leq y$ **by** *auto*
hence $0 \leq y\hat{}n$ **by** *auto*
ultimately show *?thesis* **by** *auto*
next
case *False* **hence** $0 \leq x$ **by** *auto*
with ‹$x \leq y$› **show** *?thesis* **using** *power-mono* **by** *auto*
qed
qed

61.7 General Lemmas About Division

lemma *Suc-times-mod-eq*: $1<k$ ==> *Suc* $(k * m)$ *mod* $k = 1$
apply (*induct m*)
apply (*simp-all add*: *mod-Suc*)
done

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

lemma [*simp*]: *n div k* $\leq$ (*Suc n*) *div k*
by (*simp add*: *div-le-mono*)

lemma *Suc-n-div-2-gt-zero* [*simp*]: $(0::nat) < n$ ==> $0 < (n + 1)$ *div* 2
by *arith*

lemma *div-2-gt-zero* [*simp*]: $(1::nat) < n$ ==> $0 < n$ *div* 2
by *arith*

lemma *mod-mult-self3* [*simp*]: $(k*n + m)$ *mod* $n = m$ *mod* $(n::nat)$
by (*simp add*: *mult-ac add-ac*)

lemma *mod-mult-self4* [*simp*]: *Suc* $(k*n + m)$ *mod* $n =$ *Suc* m *mod* n
proof −
have *Suc* $(k * n + m)$ *mod* $n = (k * n +$ *Suc* $m)$ *mod* n **by** *simp*
also have ... = *Suc* m *mod* n **by** (*rule mod-mult-self3*)

finally show *?thesis* **.**
qed

lemma *mod-Suc-eq-Suc-mod*: *Suc m mod n = Suc (m mod n) mod n*
apply (*subst mod-Suc* [*of m*])
apply (*subst mod-Suc* [*of m mod n*], *simp*)
done

61.8 More Even/Odd Results

lemma *even-mult-two-ex*: *even*(*n*) = (∃*m*::*nat*. *n* = *2*∗*m*) **by** *presburger*
lemma *odd-Suc-mult-two-ex*: *odd*(*n*) = (∃*m*. *n* = *Suc* (*2*∗*m*)) **by** *presburger*
lemma *even-add* [*simp*]: *even*(*m* + *n*::*nat*) = (*even m* = *even n*) **by** *presburger*

lemma *odd-add* [*simp*]: *odd*(*m* + *n*::*nat*) = (*odd m* ≠ *odd n*) **by** *presburger*

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

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

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

lemma *even-num-iff*: *0* < *n* ==> *even n* = (~ *even*(*n* − *1* :: *nat*)) **by** *presburger*
lemma *even-even-mod-4-iff*: *even* (*n*::*nat*) = *even* (*n mod 4*) **by** *presburger*

lemma *lemma-odd-mod-4-div-2*: *n mod 4* = (*3*::*nat*) ==> *odd*((*n* − *1*) *div 2*) **by** *presburger*

lemma *lemma-even-mod-4-div-2*: *n mod 4* = (*1*::*nat*) ==> *even* ((*n* − *1*) *div 2*)
by *presburger*

Simplify, when the exponent is a numeral

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

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

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

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

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

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

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

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

61.9 An Equivalence for $0 \leq a\hat{}n$

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

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

61.10 Miscellaneous

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

lemma [*presburger*]: $(Suc\ x)\ div\ Suc\ (Suc\ 0) = x\ div\ Suc\ (Suc\ 0) \longleftrightarrow even\ x$ **by** *presburger*
lemma [*presburger*]: $(Suc\ x)\ div\ Suc\ (Suc\ 0) = x\ div\ Suc\ (Suc\ 0) \longleftrightarrow even\ x$ **by** *presburger*

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

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

end

62 NthRoot: Nth Roots of Real Numbers

theory *NthRoot*
imports *Parity Deriv*
begin

62.1 Existence of Nth Root

Existence follows from the Intermediate Value Theorem

lemma *realpow-pos-nth*:
assumes *n*: *0* < *n*
assumes *a*: *0* < *a*
shows ∃ *r*>*0*. *r* ^ *n* = (*a*::*real*)
proof −
have ∃ *r*≥*0*. *r* ≤ (*max 1 a*) ∧ *r* ^ *n* = *a*
proof (*rule IVT*)
show *0* ^ *n* ≤ *a* **using** *n a* **by** (*simp add*: *power-0-left*)
show *0* ≤ *max 1 a* **by** *simp*
from *n* **have** *n1*: *1* ≤ *n* **by** *simp*
have *a* ≤ *max 1 a* ^ *1* **by** *simp*
also have *max 1 a* ^ *1* ≤ *max 1 a* ^ *n*
using *n1* **by** (*rule power-increasing*, *simp*)
finally show *a* ≤ *max 1 a* ^ *n* **.**
show ∀ *r*. *0* ≤ *r* ∧ *r* ≤ *max 1 a* ⟶ *isCont* (λ*x*. *x* ^ *n*) *r*
by (*simp add*: *isCont-power*)
qed
then obtain *r* **where** *r*: *0* ≤ *r* ∧ *r* ^ *n* = *a* **by** *fast*
with *n a* **have** *r* ≠ *0* **by** (*auto simp add*: *power-0-left*)
with *r* **have** *0* < *r* ∧ *r* ^ *n* = *a* **by** *simp*
thus *?thesis* **..**
qed

lemma *realpow-pos-nth2*: (*0*::*real*) < *a* ⟹ ∃ *r*>*0*. *r* ^ *Suc n* = *a*
by (*blast intro*: *realpow-pos-nth*)

Uniqueness of nth positive root

lemma *realpow-pos-nth-unique*:
⟦*0* < *n*; *0* < *a*⟧ ⟹ ∃!*r*. *0* < *r* ∧ *r* ^ *n* = (*a*::*real*)

apply (*auto intro*!: *realpow-pos-nth*)
apply (*rule-tac n=n* **in** *power-eq-imp-eq-base*, *simp-all*)
done

62.2 Nth Root

We define roots of negative reals such that *root n* (− *x*) = − *root n x*. This allows us to omit side conditions from many theorems.

definition
root :: [*nat*, *real*] ⇒ *real* **where**
root n x = (*if* 0 < *x then* (*THE u*. 0 < *u* ∧ *u* ^ *n* = *x*) *else*
if x < 0 *then* − (*THE u*. 0 < *u* ∧ *u* ^ *n* = − *x*) *else* 0)

lemma *real-root-zero* [*simp*]: *root n* 0 = 0
unfolding *root-def* **by** *simp*

lemma *real-root-minus*: 0 < *n* ⟹ *root n* (− *x*) = − *root n x*
unfolding *root-def* **by** *simp*

lemma *real-root-gt-zero*: ⟦0 < *n*; 0 < *x*⟧ ⟹ 0 < *root n x*
apply (*simp add*: *root-def*)
apply (*drule* (1) *realpow-pos-nth-unique*)
apply (*erule theI'* [*THEN conjunct1*])
done

lemma *real-root-pow-pos*:
⟦0 < *n*; 0 < *x*⟧ ⟹ *root n x* ^ *n* = *x*
apply (*simp add*: *root-def*)
apply (*drule* (1) *realpow-pos-nth-unique*)
apply (*erule theI'* [*THEN conjunct2*])
done

lemma *real-root-pow-pos2* [*simp*]:
⟦0 < *n*; 0 ≤ *x*⟧ ⟹ *root n x* ^ *n* = *x*
by (*auto simp add*: *order-le-less real-root-pow-pos*)

lemma *odd-real-root-pow*: *odd n* ⟹ *root n x* ^ *n* = *x*
apply (*rule-tac x=0* **and** *y=x* **in** *linorder-le-cases*)
apply (*erule* (1) *real-root-pow-pos2* [*OF odd-pos*])
apply (*subgoal-tac root n* (− *x*) ^ *n* = − *x*)
apply (*simp add*: *real-root-minus odd-pos*)
apply (*simp add*: *odd-pos*)
done

lemma *real-root-ge-zero*: ⟦0 < *n*; 0 ≤ *x*⟧ ⟹ 0 ≤ *root n x*
by (*auto simp add*: *order-le-less real-root-gt-zero*)

lemma *real-root-power-cancel*: ⟦0 < *n*; 0 ≤ *x*⟧ ⟹ *root n* (*x* ^ *n*) = *x*
apply (*subgoal-tac* 0 ≤ *x* ^ *n*)

apply (*subgoal-tac* $0 \leq root\ n\ (x\ \hat{}\ n)$)
apply (*subgoal-tac root* $n\ (x\ \hat{}\ n)\ \hat{}\ n = x\ \hat{}\ n$)
apply (*erule* (*3*) *power-eq-imp-eq-base*)
apply (*erule* (*1*) *real-root-pow-pos2*)
apply (*erule* (*1*) *real-root-ge-zero*)
apply (*erule zero-le-power*)
done

lemma *odd-real-root-power-cancel*: $odd\ n \Longrightarrow root\ n\ (x\ \hat{}\ n) = x$
apply (*rule-tac* $x=0$ **and** $y=x$ **in** *linorder-le-cases*)
apply (*erule* (*1*) *real-root-power-cancel* [*OF odd-pos*])
apply (*subgoal-tac root* $n\ ((-\ x)\ \hat{}\ n) = -\ x$)
apply (*simp add*: *real-root-minus odd-pos*)
apply (*erule real-root-power-cancel* [*OF odd-pos*], *simp*)
done

lemma *real-root-pos-unique*:
$[\![0 < n;\ 0 \leq y;\ y\ \hat{}\ n = x]\!] \Longrightarrow root\ n\ x = y$
by (*erule subst*, *rule real-root-power-cancel*)

lemma *odd-real-root-unique*:
$[\![odd\ n;\ y\ \hat{}\ n = x]\!] \Longrightarrow root\ n\ x = y$
by (*erule subst*, *rule odd-real-root-power-cancel*)

lemma *real-root-one* [*simp*]: $0 < n \Longrightarrow root\ n\ 1 = 1$
by (*simp add*: *real-root-pos-unique*)

Root function is strictly monotonic, hence injective

lemma *real-root-less-mono-lemma*:
$[\![0 < n;\ 0 \leq x;\ x < y]\!] \Longrightarrow root\ n\ x < root\ n\ y$
apply (*subgoal-tac* $0 \leq y$)
apply (*subgoal-tac root* $n\ x\ \hat{}\ n < root\ n\ y\ \hat{}\ n$)
apply (*erule power-less-imp-less-base*)
apply (*erule* (*1*) *real-root-ge-zero*)
apply *simp*
apply *simp*
done

lemma *real-root-less-mono*: $[\![0 < n;\ x < y]\!] \Longrightarrow root\ n\ x < root\ n\ y$
apply (*cases* $0 \leq x$)
apply (*erule* (*2*) *real-root-less-mono-lemma*)
apply (*cases* $0 \leq y$)
apply (*rule-tac* $y=0$ **in** *order-less-le-trans*)
apply (*subgoal-tac* $0 < root\ n\ (-\ x)$)
apply (*simp add*: *real-root-minus*)
apply (*simp add*: *real-root-gt-zero*)
apply (*simp add*: *real-root-ge-zero*)
apply (*subgoal-tac root* $n\ (-\ y) < root\ n\ (-\ x)$)
apply (*simp add*: *real-root-minus*)

apply (*simp add*: *real-root-less-mono-lemma*)
done

lemma *real-root-le-mono*: ⟦*0 < n*; *x ≤ y*⟧ ⟹ *root n x ≤ root n y*
by (*auto simp add*: *order-le-less real-root-less-mono*)

lemma *real-root-less-iff* [*simp*]:
0 < n ⟹ (*root n x < root n y*) = (*x < y*)
apply (*cases x < y*)
apply (*simp add*: *real-root-less-mono*)
apply (*simp add*: *linorder-not-less real-root-le-mono*)
done

lemma *real-root-le-iff* [*simp*]:
0 < n ⟹ (*root n x ≤ root n y*) = (*x ≤ y*)
apply (*cases x ≤ y*)
apply (*simp add*: *real-root-le-mono*)
apply (*simp add*: *linorder-not-le real-root-less-mono*)
done

lemma *real-root-eq-iff* [*simp*]:
0 < n ⟹ (*root n x = root n y*) = (*x = y*)
by (*simp add*: *order-eq-iff*)

lemmas *real-root-gt-0-iff* [*simp*] = *real-root-less-iff* [**where** *x=0*, *simplified*]
lemmas *real-root-lt-0-iff* [*simp*] = *real-root-less-iff* [**where** *y=0*, *simplified*]
lemmas *real-root-ge-0-iff* [*simp*] = *real-root-le-iff* [**where** *x=0*, *simplified*]
lemmas *real-root-le-0-iff* [*simp*] = *real-root-le-iff* [**where** *y=0*, *simplified*]
lemmas *real-root-eq-0-iff* [*simp*] = *real-root-eq-iff* [**where** *y=0*, *simplified*]

lemma *real-root-gt-1-iff* [*simp*]: *0 < n* ⟹ (*1 < root n y*) = (*1 < y*)
by (*insert real-root-less-iff* [**where** *x=1*], *simp*)

lemma *real-root-lt-1-iff* [*simp*]: *0 < n* ⟹ (*root n x < 1*) = (*x < 1*)
by (*insert real-root-less-iff* [**where** *y=1*], *simp*)

lemma *real-root-ge-1-iff* [*simp*]: *0 < n* ⟹ (*1 ≤ root n y*) = (*1 ≤ y*)
by (*insert real-root-le-iff* [**where** *x=1*], *simp*)

lemma *real-root-le-1-iff* [*simp*]: *0 < n* ⟹ (*root n x ≤ 1*) = (*x ≤ 1*)
by (*insert real-root-le-iff* [**where** *y=1*], *simp*)

lemma *real-root-eq-1-iff* [*simp*]: *0 < n* ⟹ (*root n x = 1*) = (*x = 1*)
by (*insert real-root-eq-iff* [**where** *y=1*], *simp*)

Roots of roots

lemma *real-root-Suc-0* [*simp*]: *root* (*Suc 0*) *x* = *x*
by (*simp add*: *odd-real-root-unique*)

lemma *real-root-pos-mult-exp*:
$[\![0 < m;\ 0 < n;\ 0 < x]\!] \Longrightarrow root\ (m * n)\ x = root\ m\ (root\ n\ x)$
by (*rule real-root-pos-unique*, *simp-all add*: *power-mult*)

lemma *real-root-mult-exp*:
$[\![0 < m;\ 0 < n]\!] \Longrightarrow root\ (m * n)\ x = root\ m\ (root\ n\ x)$
apply (*rule linorder-cases* [**where** *x*=*x* **and** *y*=*0*])
apply (*subgoal-tac root* $(m * n)\ (-x) = root\ m\ (root\ n\ (-x))$)
apply (*simp add*: *real-root-minus*)
apply (*simp-all add*: *real-root-pos-mult-exp*)
done

lemma *real-root-commute*:
$[\![0 < m;\ 0 < n]\!] \Longrightarrow root\ m\ (root\ n\ x) = root\ n\ (root\ m\ x)$
by (*simp add*: *real-root-mult-exp* [*symmetric*] *mult-commute*)

Monotonicity in first argument

lemma *real-root-strict-decreasing*:
$[\![0 < n;\ n < N;\ 1 < x]\!] \Longrightarrow root\ N\ x < root\ n\ x$
apply (*subgoal-tac root n* $(root\ N\ x)\ \hat{}\ n < root\ N\ (root\ n\ x)\ \hat{}\ N$, *simp*)
apply (*simp add*: *real-root-commute power-strict-increasing*
del: *real-root-pow-pos2*)
done

lemma *real-root-strict-increasing*:
$[\![0 < n;\ n < N;\ 0 < x;\ x < 1]\!] \Longrightarrow root\ n\ x < root\ N\ x$
apply (*subgoal-tac root N* $(root\ n\ x)\ \hat{}\ N < root\ n\ (root\ N\ x)\ \hat{}\ n$, *simp*)
apply (*simp add*: *real-root-commute power-strict-decreasing*
del: *real-root-pow-pos2*)
done

lemma *real-root-decreasing*:
$[\![0 < n;\ n < N;\ 1 \leq x]\!] \Longrightarrow root\ N\ x \leq root\ n\ x$
by (*auto simp add*: *order-le-less real-root-strict-decreasing*)

lemma *real-root-increasing*:
$[\![0 < n;\ n < N;\ 0 \leq x;\ x \leq 1]\!] \Longrightarrow root\ n\ x \leq root\ N\ x$
by (*auto simp add*: *order-le-less real-root-strict-increasing*)

Roots of multiplication and division

lemma *real-root-mult-lemma*:
$[\![0 < n;\ 0 \leq x;\ 0 \leq y]\!] \Longrightarrow root\ n\ (x * y) = root\ n\ x * root\ n\ y$
by (*simp add*: *real-root-pos-unique mult-nonneg-nonneg power-mult-distrib*)

lemma *real-root-inverse-lemma*:
$[\![0 < n;\ 0 \leq x]\!] \Longrightarrow root\ n\ (inverse\ x) = inverse\ (root\ n\ x)$
by (*simp add*: *real-root-pos-unique power-inverse* [*symmetric*])

lemma *real-root-mult*:

assumes *n*: *0 < n*
shows *root n (x * y) = root n x * root n y*
proof (*rule linorder-le-cases*, *rule-tac* [!] *linorder-le-cases*)
assume *0 ≤ x* **and** *0 ≤ y*
thus *?thesis* **by** (*rule real-root-mult-lemma* [*OF n*])
next
assume *0 ≤ x* **and** *y ≤ 0*
hence *0 ≤ x* **and** *0 ≤ - y* **by** *simp-all*
hence *root n (x * - y) = root n x * root n (- y)*
by (*rule real-root-mult-lemma* [*OF n*])
thus *?thesis* **by** (*simp add*: *real-root-minus* [*OF n*])
next
assume *x ≤ 0* **and** *0 ≤ y*
hence *0 ≤ - x* **and** *0 ≤ y* **by** *simp-all*
hence *root n (- x * y) = root n (- x) * root n y*
by (*rule real-root-mult-lemma* [*OF n*])
thus *?thesis* **by** (*simp add*: *real-root-minus* [*OF n*])
next
assume *x ≤ 0* **and** *y ≤ 0*
hence *0 ≤ - x* **and** *0 ≤ - y* **by** *simp-all*
hence *root n (- x * - y) = root n (- x) * root n (- y)*
by (*rule real-root-mult-lemma* [*OF n*])
thus *?thesis* **by** (*simp add*: *real-root-minus* [*OF n*])
qed

lemma *real-root-inverse*:
assumes *n*: *0 < n*
shows *root n (inverse x) = inverse (root n x)*
proof (*rule linorder-le-cases*)
assume *0 ≤ x*
thus *?thesis* **by** (*rule real-root-inverse-lemma* [*OF n*])
next
assume *x ≤ 0*
hence *0 ≤ - x* **by** *simp*
hence *root n (inverse (- x)) = inverse (root n (- x))*
by (*rule real-root-inverse-lemma* [*OF n*])
thus *?thesis* **by** (*simp add*: *real-root-minus* [*OF n*])
qed

lemma *real-root-divide*:
0 < n ⟹ root n (x / y) = root n x / root n y
by (*simp add*: *divide-inverse real-root-mult real-root-inverse*)

lemma *real-root-power*:
0 < n ⟹ root n (x ^ k) = root n x ^ k
by (*induct k*, *simp-all add*: *real-root-mult*)

lemma *real-root-abs*: *0 < n ⟹ root n |x| = |root n x|*
by (*simp add*: *abs-if real-root-minus*)

Continuity and derivatives

lemma *isCont-root-pos*:
assumes *n*: $0 < n$
assumes *x*: $0 < x$
shows *isCont* (*root n*) *x*
proof −
have *isCont* (*root n*) (*root n x* ^ *n*)
proof (*rule isCont-inverse-function* [**where** $f{=}\lambda a.\ a \hat{}\ n$])
show $0 <$ *root n x* **using** *n x* **by** *simp*
show $\forall z.\ |z -$ *root n x*$| \leq$ *root n x* $\longrightarrow$ *root n* $(z \hat{}\ n) = z$
by (*simp add*: *abs-le-iff real-root-power-cancel n*)
show $\forall z.\ |z -$ *root n x*$| \leq$ *root n x* $\longrightarrow$ *isCont* $(\lambda a.\ a \hat{}\ n)\ z$
by (*simp add*: *isCont-power*)
qed
thus *?thesis* **using** *n x* **by** *simp*
qed

lemma *isCont-root-neg*:
$[\![0 < n; x < 0]\!] \Longrightarrow$ *isCont* (*root n*) *x*
apply (*subgoal-tac isCont* $(\lambda x.\ -$ *root n* $(-\ x))\ x$)
apply (*simp add*: *real-root-minus*)
apply (*rule isCont-o2* [*OF isCont-minus* [*OF isCont-ident*]])
apply (*simp add*: *isCont-minus isCont-root-pos*)
done

lemma *isCont-root-zero*:
$0 < n \Longrightarrow$ *isCont* (*root n*) *0*
unfolding *isCont-def*
apply (*rule LIM-I*)
apply (*rule-tac x*=*r* ^ *n* **in** *exI*, *safe*)
apply (*simp*)
apply (*simp add*: *real-root-abs* [*symmetric*])
apply (*rule-tac n*=*n* **in** *power-less-imp-less-base*, *simp-all*)
done

lemma *isCont-real-root*: $0 < n \Longrightarrow$ *isCont* (*root n*) *x*
apply (*rule-tac x*=*x* **and** *y*=*0* **in** *linorder-cases*)
apply (*simp-all add*: *isCont-root-pos isCont-root-neg isCont-root-zero*)
done

lemma *DERIV-real-root*:
assumes *n*: $0 < n$
assumes *x*: $0 < x$
shows *DERIV* (*root n*) *x* :> *inverse* (*real n* ∗ *root n x* ^ $(n -$ *Suc 0*$)$)
proof (*rule DERIV-inverse-function*)
show $0 < x$ **using** *x* **.**
show $x < x + 1$ **by** *simp*
show $\forall y.\ 0 < y \wedge y < x + 1 \longrightarrow$ *root n y* ^ *n* $= y$
using *n* **by** *simp*

show *DERIV* $(\lambda x.\ x\ \hat{}\ n)$ $(root\ n\ x) :> real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0)$
by (*rule DERIV-pow*)
show $real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0) \neq 0$
using *n x* **by** *simp*
show *isCont* $(root\ n)\ x$
using *n* **by** (*rule isCont-real-root*)
qed

lemma *DERIV-odd-real-root*:
assumes *n*: *odd n*
assumes *x*: $x \neq 0$
shows *DERIV* $(root\ n)\ x :> inverse\ (real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0))$
proof (*rule DERIV-inverse-function*)
show $x - 1 < x$ **by** *simp*
show $x < x + 1$ **by** *simp*
show $\forall y.\ x - 1 < y \wedge y < x + 1 \longrightarrow root\ n\ y\ \hat{}\ n = y$
using *n* **by** (*simp add*: *odd-real-root-pow*)
show *DERIV* $(\lambda x.\ x\ \hat{}\ n)$ $(root\ n\ x) :> real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0)$
by (*rule DERIV-pow*)
show $real\ n * root\ n\ x\ \hat{}\ (n - Suc\ 0) \neq 0$
using *odd-pos* [*OF n*] *x* **by** *simp*
show *isCont* $(root\ n)\ x$
using *odd-pos* [*OF n*] **by** (*rule isCont-real-root*)
qed

62.3 Square Root

definition
$sqrt :: real \Rightarrow real$ **where**
$sqrt = root\ 2$

lemma *pos2*: $0 < (2::nat)$ **by** *simp*

lemma *real-sqrt-unique*: $[\![y^2 = x;\ 0 \leq y]\!] \Longrightarrow sqrt\ x = y$
unfolding *sqrt-def* **by** (*rule real-root-pos-unique* [*OF pos2*])

lemma *real-sqrt-abs* [*simp*]: $sqrt\ (x^2) = |x|$
apply (*rule real-sqrt-unique*)
apply (*rule power2-abs*)
apply (*rule abs-ge-zero*)
done

lemma *real-sqrt-pow2* [*simp*]: $0 \leq x \Longrightarrow (sqrt\ x)^2 = x$
unfolding *sqrt-def* **by** (*rule real-root-pow-pos2* [*OF pos2*])

lemma *real-sqrt-pow2-iff* [*simp*]: $((sqrt\ x)^2 = x) = (0 \leq x)$
apply (*rule iffI*)
apply (*erule subst*)
apply (*rule zero-le-power2*)

apply (*erule real-sqrt-pow2*)
done

lemma *real-sqrt-zero* [*simp*]: *sqrt 0 = 0*
unfolding *sqrt-def* **by** (*rule real-root-zero*)

lemma *real-sqrt-one* [*simp*]: *sqrt 1 = 1*
unfolding *sqrt-def* **by** (*rule real-root-one* [*OF pos2*])

lemma *real-sqrt-minus*: *sqrt* (− *x*) = − *sqrt x*
unfolding *sqrt-def* **by** (*rule real-root-minus* [*OF pos2*])

lemma *real-sqrt-mult*: *sqrt* (*x* ∗ *y*) = *sqrt x* ∗ *sqrt y*
unfolding *sqrt-def* **by** (*rule real-root-mult* [*OF pos2*])

lemma *real-sqrt-inverse*: *sqrt* (*inverse x*) = *inverse* (*sqrt x*)
unfolding *sqrt-def* **by** (*rule real-root-inverse* [*OF pos2*])

lemma *real-sqrt-divide*: *sqrt* (*x* / *y*) = *sqrt x* / *sqrt y*
unfolding *sqrt-def* **by** (*rule real-root-divide* [*OF pos2*])

lemma *real-sqrt-power*: *sqrt* (*x* ^ *k*) = *sqrt x* ^ *k*
unfolding *sqrt-def* **by** (*rule real-root-power* [*OF pos2*])

lemma *real-sqrt-gt-zero*: *0 < x* ⟹ *0 < sqrt x*
unfolding *sqrt-def* **by** (*rule real-root-gt-zero* [*OF pos2*])

lemma *real-sqrt-ge-zero*: *0* ≤ *x* ⟹ *0* ≤ *sqrt x*
unfolding *sqrt-def* **by** (*rule real-root-ge-zero* [*OF pos2*])

lemma *real-sqrt-less-mono*: *x < y* ⟹ *sqrt x < sqrt y*
unfolding *sqrt-def* **by** (*rule real-root-less-mono* [*OF pos2*])

lemma *real-sqrt-le-mono*: *x* ≤ *y* ⟹ *sqrt x* ≤ *sqrt y*
unfolding *sqrt-def* **by** (*rule real-root-le-mono* [*OF pos2*])

lemma *real-sqrt-less-iff* [*simp*]: (*sqrt x < sqrt y*) = (*x < y*)
unfolding *sqrt-def* **by** (*rule real-root-less-iff* [*OF pos2*])

lemma *real-sqrt-le-iff* [*simp*]: (*sqrt x* ≤ *sqrt y*) = (*x* ≤ *y*)
unfolding *sqrt-def* **by** (*rule real-root-le-iff* [*OF pos2*])

lemma *real-sqrt-eq-iff* [*simp*]: (*sqrt x = sqrt y*) = (*x = y*)
unfolding *sqrt-def* **by** (*rule real-root-eq-iff* [*OF pos2*])

lemmas *real-sqrt-gt-0-iff* [*simp*] = *real-sqrt-less-iff* [**where** *x=0*, *simplified*]
lemmas *real-sqrt-lt-0-iff* [*simp*] = *real-sqrt-less-iff* [**where** *y=0*, *simplified*]
lemmas *real-sqrt-ge-0-iff* [*simp*] = *real-sqrt-le-iff* [**where** *x=0*, *simplified*]
lemmas *real-sqrt-le-0-iff* [*simp*] = *real-sqrt-le-iff* [**where** *y=0*, *simplified*]

lemmas *real-sqrt-eq-0-iff* [*simp*] = *real-sqrt-eq-iff* [**where** *y=0*, *simplified*]

lemmas *real-sqrt-gt-1-iff* [*simp*] = *real-sqrt-less-iff* [**where** *x=1*, *simplified*]
lemmas *real-sqrt-lt-1-iff* [*simp*] = *real-sqrt-less-iff* [**where** *y=1*, *simplified*]
lemmas *real-sqrt-ge-1-iff* [*simp*] = *real-sqrt-le-iff* [**where** *x=1*, *simplified*]
lemmas *real-sqrt-le-1-iff* [*simp*] = *real-sqrt-le-iff* [**where** *y=1*, *simplified*]
lemmas *real-sqrt-eq-1-iff* [*simp*] = *real-sqrt-eq-iff* [**where** *y=1*, *simplified*]

lemma *isCont-real-sqrt*: *isCont sqrt x*
unfolding *sqrt-def* **by** (*rule isCont-real-root* [*OF pos2*])

lemma *DERIV-real-sqrt*:
  0 < x ⟹ *DERIV sqrt x :> inverse (sqrt x) / 2*
unfolding *sqrt-def* **by** (*rule DERIV-real-root* [*OF pos2*, *simplified*])

lemma *not-real-square-gt-zero* [*simp*]: (~ (*0*::*real*) < *x*∗*x*) = (*x* = *0*)
apply *auto*
apply (*cut-tac x = x* **and** *y = 0* **in** *linorder-less-linear*)
apply (*simp add*: *zero-less-mult-iff*)
done

lemma *real-sqrt-abs2* [*simp*]: *sqrt*(*x*∗*x*) = |*x*|
apply (*subst power2-eq-square* [*symmetric*])
apply (*rule real-sqrt-abs*)
done

lemma *real-sqrt-pow2-gt-zero*: *0 < x ==> 0 < (sqrt x)*^{2}
by *simp*

lemma *real-sqrt-not-eq-zero*: *0 < x ==> sqrt x* ≠ *0*
by *simp*

lemma *real-inv-sqrt-pow2*: *0 < x ==> inverse* (*sqrt*(*x*)) ^ *2 = inverse x*
by (*simp add*: *power-inverse* [*symmetric*])

lemma *real-sqrt-eq-zero-cancel*: [| *0* ≤ *x*; *sqrt*(*x*) = *0*|] ==> *x = 0*
by *simp*

lemma *real-sqrt-ge-one*: *1* ≤ *x ==> 1* ≤ *sqrt x*
by *simp*

lemma *real-sqrt-two-gt-zero* [*simp*]: *0 < sqrt 2*
by *simp*

lemma *real-sqrt-two-ge-zero* [*simp*]: *0* ≤ *sqrt 2*
by *simp*

lemma *real-sqrt-two-gt-one* [*simp*]: *1 < sqrt 2*
by *simp*

lemma *sqrt-divide-self-eq*:
  assumes *nneg*: $0 \leq x$
  shows *sqrt x / x = inverse (sqrt x)*
proof *cases*
  assume *x=0* **thus** *?thesis* **by** *simp*
next
  assume *nz*: $x \neq 0$
  hence *pos*: $0<x$ **using** *nneg* **by** *arith*
  show *?thesis*
  proof (*rule right-inverse-eq* [*THEN iffD1*, *THEN sym*])
    show *sqrt x / x ≠ 0* **by** (*simp add*: *divide-inverse nneg nz*)
    show *inverse (sqrt x) / (sqrt x / x) = 1*
      by (*simp add*: *divide-inverse mult-assoc* [*symmetric*]
                power2-eq-square [*symmetric*] *real-inv-sqrt-pow2 pos nz*)
  qed
qed

lemma *real-divide-square-eq* [*simp*]: *(((r::real) * a) / (r * r)) = a / r*
apply (*simp add*: *divide-inverse*)
apply (*case-tac r=0*)
apply (*auto simp add*: *mult-ac*)
done

lemma *lemma-real-divide-sqrt-less*: *0 < u ==> u / sqrt 2 < u*
by (*simp add*: *divide-less-eq mult-compare-simps*)

lemma *four-x-squared*:
  fixes *x::real*
  shows $4 * x^2 = (2 * x)^2$
by (*simp add*: *power2-eq-square*)

62.4 Square Root of Sum of Squares

lemma *real-sqrt-mult-self-sum-ge-zero* [*simp*]: $0 \leq sqrt(x*x + y*y)$
by (*rule real-sqrt-ge-zero* [*OF sum-squares-ge-zero*])

lemma *real-sqrt-sum-squares-ge-zero* [*simp*]: $0 \leq sqrt\ (x^2 + y^2)$
by *simp*

declare *real-sqrt-sum-squares-ge-zero* [*THEN abs-of-nonneg*, *simp*]

lemma *real-sqrt-sum-squares-mult-ge-zero* [*simp*]:
    $0 \leq sqrt\ ((x^2 + y^2)*(xa^2 + ya^2))$
by (*auto intro*!: *real-sqrt-ge-zero simp add*: *zero-le-mult-iff*)

lemma *real-sqrt-sum-squares-mult-squared-eq* [*simp*]:
    $sqrt\ ((x^2 + y^2) * (xa^2 + ya^2))\ \hat{}\ 2 = (x^2 + y^2) * (xa^2 + ya^2)$
by (*auto simp add*: *zero-le-mult-iff*)

lemma *real-sqrt-sum-squares-eq-cancel*: $sqrt\ (x^2 + y^2) = x \Longrightarrow y = 0$
by (*drule-tac* $f = \%x.\ x^2$ **in** *arg-cong*, *simp*)

lemma *real-sqrt-sum-squares-eq-cancel2*: $sqrt\ (x^2 + y^2) = y \Longrightarrow x = 0$
by (*drule-tac* $f = \%x.\ x^2$ **in** *arg-cong*, *simp*)

lemma *real-sqrt-sum-squares-ge1* [*simp*]: $x \leq sqrt\ (x^2 + y^2)$
by (*rule power2-le-imp-le*, *simp-all*)

lemma *real-sqrt-sum-squares-ge2* [*simp*]: $y \leq sqrt\ (x^2 + y^2)$
by (*rule power2-le-imp-le*, *simp-all*)

lemma *real-sqrt-ge-abs1* [*simp*]: $|x| \leq sqrt\ (x^2 + y^2)$
by (*rule power2-le-imp-le*, *simp-all*)

lemma *real-sqrt-ge-abs2* [*simp*]: $|y| \leq sqrt\ (x^2 + y^2)$
by (*rule power2-le-imp-le*, *simp-all*)

lemma *le-real-sqrt-sumsq* [*simp*]: $x \leq sqrt\ (x * x + y * y)$
by (*simp add*: *power2-eq-square* [*symmetric*])

lemma *power2-sum*:
  fixes $x\ y :: 'a::\{number\text{-}ring,recpower\}$
  shows $(x + y)^2 = x^2 + y^2 + 2 * x * y$
by (*simp add*: *ring-distribs power2-eq-square*)

lemma *power2-diff*:
  fixes $x\ y :: 'a::\{number\text{-}ring,recpower\}$
  shows $(x - y)^2 = x^2 + y^2 - 2 * x * y$
by (*simp add*: *ring-distribs power2-eq-square*)

lemma *real-sqrt-sum-squares-triangle-ineq*:
  $sqrt\ ((a + c)^2 + (b + d)^2) \leq sqrt\ (a^2 + b^2) + sqrt\ (c^2 + d^2)$
apply (*rule power2-le-imp-le*, *simp*)
apply (*simp add*: *power2-sum*)
apply (*simp only*: *mult-assoc right-distrib* [*symmetric*])
apply (*rule mult-left-mono*)
apply (*rule power2-le-imp-le*)
apply (*simp add*: *power2-sum power-mult-distrib*)
apply (*simp add*: *ring-distribs*)
apply (*subgoal-tac* $0 \leq b^2 * c^2 + a^2 * d^2 - 2 * (a * c) * (b * d)$, *simp*)
apply (*rule-tac* $b=(a * d - b * c)^2$ **in** *ord-le-eq-trans*)
apply (*rule zero-le-power2*)
apply (*simp add*: *power2-diff power-mult-distrib*)
apply (*simp add*: *mult-nonneg-nonneg*)
apply *simp*
apply (*simp add*: *add-increasing*)
done

lemma *real-sqrt-sum-squares-less*:
⟦|*x*| < *u* / *sqrt 2*; |*y*| < *u* / *sqrt 2*⟧ ⟹ *sqrt* (*x*² + *y*²) < *u*
apply (*rule power2-less-imp-less*, *simp*)
apply (*drule power-strict-mono* [*OF - abs-ge-zero pos2*])
apply (*drule power-strict-mono* [*OF - abs-ge-zero pos2*])
apply (*simp add*: *power-divide*)
apply (*drule order-le-less-trans* [*OF abs-ge-zero*])
apply (*simp add*: *zero-less-divide-iff*)
done

Needed for the infinitely close relation over the nonstandard complex numbers

lemma *lemma-sqrt-hcomplex-capprox*:
[| *0* < *u*; *x* < *u*/*2*; *y* < *u*/*2*; *0* ≤ *x*; *0* ≤ *y* |] ==> *sqrt* (*x*² + *y*²) < *u*
apply (*rule-tac y* = *u*/*sqrt 2* **in** *order-le-less-trans*)
apply (*erule-tac* [*2*] *lemma-real-divide-sqrt-less*)
apply (*rule power2-le-imp-le*)
apply (*auto simp add*: *real-0-le-divide-iff power-divide*)
apply (*rule-tac t* = *u*² **in** *real-sum-of-halves* [*THEN subst*])
apply (*rule add-mono*)
apply (*auto simp add*: *four-x-squared intro*: *power-mono*)
done

Legacy theorem names:

lemmas *real-root-pos2* = *real-root-power-cancel*
lemmas *real-root-pos-pos* = *real-root-gt-zero* [*THEN order-less-imp-le*]
lemmas *real-root-pos-pos-le* = *real-root-ge-zero*
lemmas *real-sqrt-mult-distrib* = *real-sqrt-mult*
lemmas *real-sqrt-mult-distrib2* = *real-sqrt-mult*
lemmas *real-sqrt-eq-zero-cancel-iff* = *real-sqrt-eq-0-iff*

lemma *real-root-pos*: *0* < *x* ⟹ *root* (*Suc n*) (*x* ^ (*Suc n*)) = *x*
by (*rule real-root-power-cancel* [*OF zero-less-Suc order-less-imp-le*])

end

63 Transcendental: Power Series, Transcendental Functions etc.

theory *Transcendental*
imports *Fact Series Deriv NthRoot*
begin

63.1 Properties of Power Series

lemma *lemma-realpow-diff*:
fixes $y :: 'a::recpower$
shows $p \leq n \Longrightarrow y \mathbin{\hat{}} (Suc\ n - p) = (y \mathbin{\hat{}} (n - p)) * y$
proof −
assume $p \leq n$
hence $Suc\ n - p = Suc\ (n - p)$ **by** (*rule Suc-diff-le*)
thus *?thesis* **by** (*simp add*: *power-commutes*)
qed

lemma *lemma-realpow-diff-sumr*:
fixes $y :: 'a::\{recpower,comm\text{-}semiring\text{-}0\}$ **shows**
$(\sum p{=}0..{<}Suc\ n.\ (x \mathbin{\hat{}} p) * y \mathbin{\hat{}} (Suc\ n - p)) =$
$y * (\sum p{=}0..{<}Suc\ n.\ (x \mathbin{\hat{}} p) * y \mathbin{\hat{}} (n - p))$
by (*simp add*: *setsum-right-distrib lemma-realpow-diff mult-ac*
del: *setsum-op-ivl-Suc cong*: *strong-setsum-cong*)

lemma *lemma-realpow-diff-sumr2*:
fixes $y :: 'a::\{recpower,comm\text{-}ring\}$ **shows**
$x \mathbin{\hat{}} (Suc\ n) - y \mathbin{\hat{}} (Suc\ n) =$
$(x - y) * (\sum p{=}0..{<}Suc\ n.\ (x \mathbin{\hat{}} p) * y \mathbin{\hat{}} (n - p))$
apply (*induct n*, *simp*)
apply (*simp del*: *setsum-op-ivl-Suc*)
apply (*subst setsum-op-ivl-Suc*)
apply (*subst lemma-realpow-diff-sumr*)
apply (*simp add*: *right-distrib del*: *setsum-op-ivl-Suc*)
apply (*subst mult-left-commute* [**where** $a{=}x - y$])
apply (*erule subst*)
apply (*simp add*: *algebra-simps*)
done

lemma *lemma-realpow-rev-sumr*:
$(\sum p{=}0..{<}Suc\ n.\ (x \mathbin{\hat{}} p) * (y \mathbin{\hat{}} (n - p))) =$
$(\sum p{=}0..{<}Suc\ n.\ (x \mathbin{\hat{}} (n - p)) * (y \mathbin{\hat{}} p))$
apply (*rule setsum-reindex-cong* [**where** $f{=}\lambda i.\ n - i$])
apply (*rule inj-onI*, *simp*)
apply *auto*
apply (*rule-tac* $x{=}n - x$ **in** *image-eqI*, *simp*, *simp*)
done

Power series has a 'circle' of convergence, i.e. if it sums for x, then it sums absolutely for z with $|z| < |x|$.

lemma *powser-insidea*:
fixes $x\ z :: 'a::\{real\text{-}normed\text{-}field,banach,recpower\}$
assumes *1*: $summable\ (\lambda n.\ f\ n * x \mathbin{\hat{}} n)$
assumes *2*: $norm\ z < norm\ x$
shows $summable\ (\lambda n.\ norm\ (f\ n * z \mathbin{\hat{}} n))$
proof −
from *2* **have** *x-neq-0*: $x \neq 0$ **by** *clarsimp*

```
  from 1 have (λn. f n * x ^ n) ----> 0
    by (rule summable-LIMSEQ-zero)
  hence convergent (λn. f n * x ^ n)
    by (rule convergentI)
  hence Cauchy (λn. f n * x ^ n)
    by (simp add: Cauchy-convergent-iff)
  hence Bseq (λn. f n * x ^ n)
    by (rule Cauchy-Bseq)
  then obtain K where 3: 0 < K and 4: ∀ n. norm (f n * x ^ n) ≤ K
    by (simp add: Bseq-def, safe)
  have ∃ N. ∀ n≥N. norm (norm (f n * z ^ n)) ≤
                   K * norm (z ^ n) * inverse (norm (x ^ n))
  proof (intro exI allI impI)
    fix n::nat assume 0 ≤ n
    have norm (norm (f n * z ^ n)) * norm (x ^ n) =
          norm (f n * x ^ n) * norm (z ^ n)
      by (simp add: norm-mult abs-mult)
    also have ... ≤ K * norm (z ^ n)
      by (simp only: mult-right-mono 4 norm-ge-zero)
    also have ... = K * norm (z ^ n) * (inverse (norm (x ^ n)) * norm (x ^ n))
      by (simp add: x-neq-0)
    also have ... = K * norm (z ^ n) * inverse (norm (x ^ n)) * norm (x ^ n)
      by (simp only: mult-assoc)
    finally show norm (norm (f n * z ^ n)) ≤
                 K * norm (z ^ n) * inverse (norm (x ^ n))
      by (simp add: mult-le-cancel-right x-neq-0)
  qed
  moreover have summable (λn. K * norm (z ^ n) * inverse (norm (x ^ n)))
  proof -
    from 2 have norm (norm (z * inverse x)) < 1
      using x-neq-0
      by (simp add: nonzero-norm-divide divide-inverse [symmetric])
    hence summable (λn. norm (z * inverse x) ^ n)
      by (rule summable-geometric)
    hence summable (λn. K * norm (z * inverse x) ^ n)
      by (rule summable-mult)
    thus summable (λn. K * norm (z ^ n) * inverse (norm (x ^ n)))
      using x-neq-0
      by (simp add: norm-mult nonzero-norm-inverse power-mult-distrib
                    power-inverse norm-power mult-assoc)
  qed
  ultimately show summable (λn. norm (f n * z ^ n))
    by (rule summable-comparison-test)
qed

lemma powser-inside:
  fixes f :: nat ⇒ 'a::{real-normed-field,banach,recpower} shows
    [| summable (%n. f(n) * (x ^ n)); norm z < norm x |]
     ==> summable (%n. f(n) * (z ^ n))
```

by (*rule powser-insidea* [*THEN summable-norm-cancel*])

lemma *sum-split-even-odd*: **fixes** $f :: nat \Rightarrow real$ **shows**
$(\sum\ i = 0\ ..<\ 2 * n.\ if\ even\ i\ then\ f\ i\ else\ g\ i) =$
$(\sum\ i = 0\ ..<\ n.\ f\ (2 * i)) + (\sum\ i = 0\ ..<\ n.\ g\ (2 * i + 1))$
proof (*induct n*)
case (*Suc n*)
have $(\sum\ i = 0\ ..<\ 2 * Suc\ n.\ if\ even\ i\ then\ f\ i\ else\ g\ i) =$
$(\sum\ i = 0\ ..<\ n.\ f\ (2 * i)) + (\sum\ i = 0\ ..<\ n.\ g\ (2 * i + 1)) + (f\ (2 * n) + g\ (2 * n + 1))$
using *Suc.hyps* **unfolding** *One-nat-def* **by** *auto*
also have $\ldots = (\sum\ i = 0\ ..<\ Suc\ n.\ f\ (2 * i)) + (\sum\ i = 0\ ..<\ Suc\ n.\ g\ (2 * i + 1))$ **by** *auto*
finally show *?case* **.**
qed *auto*

lemma *sums-if'*: **fixes** $g :: nat \Rightarrow real$ **assumes** *g sums x*
shows $(\lambda\ n.\ if\ even\ n\ then\ 0\ else\ g\ ((n - 1)\ div\ 2))\ sums\ x$
unfolding *sums-def*
proof (*rule LIMSEQ-I*)
fix $r :: real$ **assume** $0 < r$
from ‹*g sums x*›[*unfolded sums-def*, *THEN LIMSEQ-D*, *OF this*]
obtain *no* **where** *no-eq*: $\bigwedge\ n.\ n \geq no \Longrightarrow (norm\ (setsum\ g\ \{\ 0..<n\ \} - x) < r)$ **by** *blast*

let *?SUM* $= \lambda\ m.\ \sum\ i = 0\ ..<\ m.\ if\ even\ i\ then\ 0\ else\ g\ ((i - 1)\ div\ 2)$
{ fix *m* **assume** $m \geq 2 * no$ **hence** $m\ div\ 2 \geq no$ **by** *auto*
have *sum-eq*: $?SUM\ (2 * (m\ div\ 2)) = setsum\ g\ \{\ 0\ ..<\ m\ div\ 2\ \}$
using *sum-split-even-odd* **by** *auto*
hence $(norm\ (?SUM\ (2 * (m\ div\ 2)) - x) < r)$ **using** *no-eq* **unfolding** *sum-eq* **using** ‹$m\ div\ 2 \geq no$› **by** *auto*
moreover
have $?SUM\ (2 * (m\ div\ 2)) = ?SUM\ m$
proof (*cases even m*)
case *True* **show** *?thesis* **unfolding** *even-nat-div-two-times-two*[*OF True*, *unfolded numeral-2-eq-2*[*symmetric*]] **..**
next
case *False* **hence** *even* (*Suc m*) **by** *auto*
from *even-nat-div-two-times-two*[*OF this*, *unfolded numeral-2-eq-2*[*symmetric*]] *odd-nat-plus-one-div-two*[*OF False*, *unfolded numeral-2-eq-2*[*symmetric*]]
have *eq*: $Suc\ (2 * (m\ div\ 2)) = m$ **by** *auto*
hence $even\ (2 * (m\ div\ 2))$ **using** ‹*odd m*› **by** *auto*
have $?SUM\ m = ?SUM\ (Suc\ (2 * (m\ div\ 2)))$ **unfolding** *eq* **..**
also have $\ldots = ?SUM\ (2 * (m\ div\ 2))$ **using** ‹$even\ (2 * (m\ div\ 2))$› **by** *auto*
finally show *?thesis* **by** *auto*
qed
ultimately have $(norm\ (?SUM\ m - x) < r)$ **by** *auto*
}
thus $\exists\ no.\ \forall\ m \geq no.\ norm\ (?SUM\ m - x) < r$ **by** *blast*

qed

lemma *sums-if*: **fixes** $g :: nat \Rightarrow real$ **assumes** g *sums* x **and** f *sums* y
shows $(\lambda\ n.$ *if even* n *then* $f\ (n\ div\ 2)$ *else* $g\ ((n - 1)\ div\ 2))$ *sums* $(x + y)$
proof −
let *?s* $= \lambda\ n.$ *if even* n *then* 0 *else* $f\ ((n - 1)\ div\ 2)$
{ **fix** $B\ T\ E$ **have** $(if\ B\ then\ (0 :: real)\ else\ E) + (if\ B\ then\ T\ else\ 0) = (if\ B\ then\ T\ else\ E)$
by $(cases\ B)$ *auto* } **note** *if-sum* = *this*
have *g-sums*: $(\lambda\ n.$ *if even* n *then* 0 *else* $g\ ((n - 1)\ div\ 2))$ *sums* x **using** *sums-if'*[*OF* ‹*g sums x*›] .
{
have *?s* 0 = 0 **by** *auto*
have *Suc-m1*: $\bigwedge\ n.$ *Suc* $n - 1 = n$ **by** *auto*
{ **fix** $B\ T\ E$ **have** $(if\ \neg\ B\ then\ T\ else\ E) = (if\ B\ then\ E\ else\ T)$ **by** *auto* }
note *if-eq* = *this*

have *?s sums y* **using** *sums-if'*[*OF* ‹*f sums y*›] .
from *this*[*unfolded sums-def*, *THEN LIMSEQ-Suc*]
have $(\lambda\ n.$ *if even* n *then* $f\ (n\ div\ 2)$ *else* 0) *sums y*
unfolding *sums-def setsum-shift-lb-Suc0-0-upt*[**where** *f*=*?s*, *OF* ‹*?s* 0 = 0›, *symmetric*]
image-Suc-atLeastLessThan[*symmetric*] *setsum-reindex*[*OF inj-Suc*, *unfolded comp-def*]
even-nat-Suc Suc-m1 if-eq .
} **from** *sums-add*[*OF g-sums this*]
show *?thesis* **unfolding** *if-sum* .
qed

63.2 Alternating series test / Leibniz formula

lemma *sums-alternating-upper-lower*:
fixes $a :: nat \Rightarrow real$
assumes *mono*: $\bigwedge n.\ a\ (Suc\ n) \leq a\ n$ **and** *a-pos*: $\bigwedge n.\ 0 \leq a\ n$ **and** a ----> 0
shows $\exists\ l.\ ((\forall\ n.\ (\sum i{=}0..{<}2{*}n.\ -1\hat{}i{*}a\ i) \leq l) \wedge (\lambda\ n.\ \sum i{=}0..{<}2{*}n.\ -1\hat{}i{*}a\ i)$ ----> $l) \wedge$
$((\forall\ n.\ l \leq (\sum i{=}0..{<}2{*}n + 1.\ -1\hat{}i{*}a\ i)) \wedge (\lambda\ n.\ \sum i{=}0..{<}2{*}n + 1.\ -1\hat{}i{*}a\ i)$ ----> $l)$
(**is** $\exists\ l.\ ((\forall\ n.$ *?f* $n \leq l) \wedge$ -) $\wedge\ ((\forall\ n.\ l \leq$ *?g* $n) \wedge$ -))
proof −
have *fg-diff*: $\bigwedge n.$ *?f* $n -$ *?g* $n = -\ a\ (2 * n)$ **unfolding** *One-nat-def* **by** *auto*

have $\forall\ n.$ *?f* $n \leq$ *?f* $(Suc\ n)$
proof fix n **show** *?f* $n \leq$ *?f* $(Suc\ n)$ **using** *mono*[*of* 2∗*n*] **by** *auto* **qed**
moreover
have $\forall\ n.$ *?g* $(Suc\ n) \leq$ *?g* n
proof fix n **show** *?g* $(Suc\ n) \leq$ *?g* n **using** *mono*[*of Suc* (2∗*n*)]
unfolding *One-nat-def* **by** *auto* **qed**

```
  moreover
  have ∀ n. ?f n ≤ ?g n
  proof fix n show ?f n ≤ ?g n using fg-diff a-pos
    unfolding One-nat-def by auto qed
  moreover
  have (λ n. ?f n − ?g n) ----> 0 unfolding fg-diff
  proof (rule LIMSEQ-I)
    fix r :: real assume 0 < r
    with ‹a ----> 0›[THEN LIMSEQ-D]
    obtain N where ⋀ n. n ≥ N ⟹ norm (a n − 0) < r by auto
    hence ∀ n ≥ N. norm (− a (2 * n) − 0) < r by auto
    thus ∃ N. ∀ n ≥ N. norm (− a (2 * n) − 0) < r by auto
  qed
  ultimately
  show ?thesis by (rule lemma-nest-unique)
qed

lemma summable-Leibniz': fixes a :: nat ⇒ real
  assumes a-zero: a ----> 0 and a-pos: ⋀ n. 0 ≤ a n
  and a-monotone: ⋀ n. a (Suc n) ≤ a n
  shows summable: summable (λ n. (−1) ^n * a n)
  and ⋀n. (∑ i=0..<2*n. (−1) ^i*a i) ≤ (∑ i. (−1) ^i*a i)
  and (λn. ∑ i=0..<2*n. (−1) ^i*a i) ----> (∑ i. (−1) ^i*a i)
  and ⋀n. (∑ i. (−1) ^i*a i) ≤ (∑ i=0..<2*n+1. (−1) ^i*a i)
  and (λn. ∑ i=0..<2*n+1. (−1) ^i*a i) ----> (∑ i. (−1) ^i*a i)
proof −
  let ?S n = (−1) ^n * a n
  let ?P n = ∑ i=0..<n. ?S i
  let ?f n = ?P (2 * n)
  let ?g n = ?P (2 * n + 1)
  obtain l :: real where below-l: ∀ n. ?f n ≤ l and ?f ----> l and above-l: ∀
n. l ≤ ?g n and ?g ----> l
    using sums-alternating-upper-lower[OF a-monotone a-pos a-zero] by blast

  let ?Sa = λ m. ∑ n = 0..<m. ?S n
  have ?Sa ----> l
  proof (rule LIMSEQ-I)
    fix r :: real assume 0 < r

    with ‹?f ----> l›[THEN LIMSEQ-D]
    obtain f-no where f: ⋀ n. n ≥ f-no ⟹ norm (?f n − l) < r by auto

    from ‹0 < r› ‹?g ----> l›[THEN LIMSEQ-D]
    obtain g-no where g: ⋀ n. n ≥ g-no ⟹ norm (?g n − l) < r by auto

    { fix n :: nat
      assume n ≥ (max (2 * f-no) (2 * g-no)) hence n ≥ 2 * f-no and n ≥ 2 *
g-no by auto
      have norm (?Sa n − l) < r
```

```
      proof (cases even n)
        case True from even-nat-div-two-times-two[OF this]
       have n-eq: 2 * (n div 2) = n unfolding numeral-2-eq-2[symmetric] by auto
        with ‹n ≥ 2 * f-no› have n div 2 ≥ f-no by auto
        from f[OF this]
        show ?thesis unfolding n-eq atLeastLessThanSuc-atLeastAtMost .
      next
        case False hence even (n − 1) using even-num-iff odd-pos by auto
        from even-nat-div-two-times-two[OF this]
      have n-eq: 2 * ((n − 1) div 2) = n − 1 unfolding numeral-2-eq-2[symmetric]
by auto
        hence range-eq: n − 1 + 1 = n using odd-pos[OF False] by auto

        from n-eq ‹n ≥ 2 * g-no› have (n − 1) div 2 ≥ g-no by auto
        from g[OF this]
       show ?thesis unfolding n-eq atLeastLessThanSuc-atLeastAtMost range-eq .
      qed
    }
    thus ∃ no. ∀ n ≥ no. norm (?Sa n − l) < r by blast
  qed
  hence sums-l: (λi. (−1) ^ i * a i) sums l unfolding sums-def atLeastLessThanSuc-atLeastAtMost[symmetric]
.
  thus summable ?S using summable-def by auto

  have l = suminf ?S using sums-unique[OF sums-l] .

  { fix n show suminf ?S ≤ ?g n unfolding sums-unique[OF sums-l, symmetric]
using above-l by auto }
  { fix n show ?f n ≤ suminf ?S unfolding sums-unique[OF sums-l, symmetric]
using below-l by auto }
  show ?g ----> suminf ?S using ‹?g ----> l› ‹l = suminf ?S› by auto
  show ?f ----> suminf ?S using ‹?f ----> l› ‹l = suminf ?S› by auto
qed

theorem summable-Leibniz: fixes a :: nat ⇒ real
  assumes a-zero: a ----> 0 and monoseq a
  shows summable (λ n. (−1) ^ n * a n) (is ?summable)
   and 0 < a 0 ⟶ (∀ n. (∑ i. −1 ^ i * a i) ∈ { ∑ i=0..<2*n. −1 ^ i * a i ..
∑ i=0..<2*n+1. −1 ^ i * a i}) (is ?pos)
   and a 0 < 0 ⟶ (∀ n. (∑ i. −1 ^ i * a i) ∈ { ∑ i=0..<2*n+1. −1 ^ i * a i ..
∑ i=0..<2*n. −1 ^ i * a i}) (is ?neg)
  and (λn. ∑ i=0..<2*n. −1 ^ i * a i) ----> (∑ i. −1 ^ i * a i) (is ?f)
  and (λn. ∑ i=0..<2*n+1. −1 ^ i * a i) ----> (∑ i. −1 ^ i * a i) (is ?g)
proof −
  have ?summable ∧ ?pos ∧ ?neg ∧ ?f ∧ ?g
  proof (cases (∀ n. 0 ≤ a n) ∧ (∀ m. ∀ n≥m. a n ≤ a m))
    case True
    hence ord: ⋀n m. m ≤ n ⟹ a n ≤ a m and ge0: ⋀ n. 0 ≤ a n by auto
    { fix n have a (Suc n) ≤ a n using ord[where n=Suc n and m=n] by auto
```

}
note *leibniz* = *summable-Leibniz'*[*OF* ‹*a* −−−−> *0*› *ge0*] **and** *mono* = *this*
from *leibniz*[*OF mono*]
show *?thesis* **using** ‹*0* ≤ *a 0*› **by** *auto*
next
let *?a* = λ *n*. − *a n*
case *False*
with *monoseq-le*[*OF* ‹*monoseq a*› ‹*a* −−−−> *0*›]
have (∀ *n*. *a n* ≤ *0*) ∧ (∀ *m*. ∀ *n*≥*m*. *a m* ≤ *a n*) **by** *auto*
hence *ord*: ⋀*n m*. *m* ≤ *n* ⟹ *?a n* ≤ *?a m* **and** *ge0*: ⋀ *n*. *0* ≤ *?a n* **by** *auto*
{ fix *n* **have** *?a* (*Suc n*) ≤ *?a n* **using** *ord*[**where** *n*=*Suc n* **and** *m*=*n*] **by** *auto* **}**
note *monotone* = *this*
note *leibniz* = *summable-Leibniz'*[*OF - ge0*, *of* λ*x*. *x*, *OF LIMSEQ-minus*[*OF* ‹*a* −−−−> *0*›, *unfolded minus-zero*] *monotone*]
have *summable* (λ *n*. (−*1*) ^*n* ∗ *?a n*) **using** *leibniz*(*1*) **by** *auto*
then obtain *l* **where** (λ *n*. (−*1*) ^*n* ∗ *?a n*) *sums l* **unfolding** *summable-def* **by** *auto*
from *this*[*THEN sums-minus*]
have (λ *n*. (−*1*) ^*n* ∗ *a n*) *sums* −*l* **by** *auto*
hence *?summable* **unfolding** *summable-def* **by** *auto*
moreover
have ⋀ *a b* :: *real*. | − *a* − − *b* | = |*a* − *b*| **unfolding** *minus-diff-minus* **by** *auto*

from *suminf-minus*[*OF leibniz*(*1*), *unfolded mult-minus-right minus-minus*]
have *move-minus*: (∑ *n*. − (−*1* ^ *n* ∗ *a n*)) = − (∑ *n*. −*1* ^ *n* ∗ *a n*) **by** *auto*

have *?pos* **using** ‹*0* ≤ *?a 0*› **by** *auto*
moreover have *?neg* **using** *leibniz*(*2*,*4*) **unfolding** *mult-minus-right setsum-negf move-minus neg-le-iff-le* **by** *auto*
moreover have *?f* **and** *?g* **using** *leibniz*(*3*,*5*)[*unfolded mult-minus-right setsum-negf move-minus*, *THEN LIMSEQ-minus-cancel*] **by** *auto*
ultimately show *?thesis* **by** *auto*
qed
from *this*[*THEN conjunct1*] *this*[*THEN conjunct2*, *THEN conjunct1*] *this*[*THEN conjunct2*, *THEN conjunct2*, *THEN conjunct1*] *this*[*THEN conjunct2*, *THEN conjunct2*, *THEN conjunct2*, *THEN conjunct1*]
this[*THEN conjunct2*, *THEN conjunct2*, *THEN conjunct2*, *THEN conjunct2*]
show *?summable* **and** *?pos* **and** *?neg* **and** *?f* **and** *?g* **.**
qed

63.3 Term-by-Term Differentiability of Power Series

definition
diffs :: (*nat* => *'a*::*ring-1*) => *nat* => *'a* **where**
diffs c = (%*n*. *of-nat* (*Suc n*) ∗ *c*(*Suc n*))

Lemma about distributing negation over it

lemma *diffs-minus*: *diffs* (%*n*. − *c n*) = (%*n*. − *diffs c n*)
by (*simp add*: *diffs-def*)

lemma *sums-Suc-imp*:
assumes *f*: *f 0* = *0*
shows (λ*n*. *f* (*Suc n*)) *sums s* ⟹ (λ*n*. *f n*) *sums s*
unfolding *sums-def*
apply (*rule LIMSEQ-imp-Suc*)
apply (*subst setsum-shift-lb-Suc0-0-upt* [**where** *f*=*f*, *OF f*, *symmetric*])
apply (*simp only*: *setsum-shift-bounds-Suc-ivl*)
done

lemma *diffs-equiv*:
summable (%*n*. (*diffs c*)(*n*) ∗ (*x* ^ *n*)) ==>
(%*n*. *of-nat n* ∗ *c*(*n*) ∗ (*x* ^ (*n* − *Suc 0*))) *sums*
(∑ *n*. (*diffs c*)(*n*) ∗ (*x* ^ *n*))
unfolding *diffs-def*
apply (*drule summable-sums*)
apply (*rule sums-Suc-imp*, *simp-all*)
done

lemma *lemma-termdiff1*:
fixes *z* :: *'a* :: {*recpower*,*comm-ring*} **shows**
(∑ *p*=*0*..<*m*. (((*z* + *h*) ^ (*m* − *p*)) ∗ (*z* ^ *p*)) − (*z* ^ *m*)) =
(∑ *p*=*0*..<*m*. (*z* ^ *p*) ∗ (((*z* + *h*) ^ (*m* − *p*)) − (*z* ^ (*m* − *p*))))
by(*auto simp add*: *algebra-simps power-add* [*symmetric*] *cong*: *strong-setsum-cong*)

lemma *sumr-diff-mult-const2*:
setsum f {*0*..<*n*} − *of-nat n* ∗ (*r*::*'a*::*ring-1*) = (∑ *i* = *0*..<*n*. *f i* − *r*)
by (*simp add*: *setsum-subtractf*)

lemma *lemma-termdiff2*:
fixes *h* :: *'a* :: {*recpower*,*field*}
assumes *h*: *h* ≠ *0* **shows**
((*z* + *h*) ^ *n* − *z* ^ *n*) / *h* − *of-nat n* ∗ *z* ^ (*n* − *Suc 0*) =
h ∗ (∑ *p*=*0*..< *n* − *Suc 0*. ∑ *q*=*0*..< *n* − *Suc 0* − *p*.
(*z* + *h*) ^ *q* ∗ *z* ^ (*n* − *2* − *q*)) (**is** *?lhs* = *?rhs*)
apply (*subgoal-tac h* ∗ *?lhs* = *h* ∗ *?rhs*, *simp add*: *h*)
apply (*simp add*: *right-diff-distrib diff-divide-distrib h*)
apply (*simp add*: *mult-assoc* [*symmetric*])
apply (*cases n*, *simp*)
apply (*simp add*: *lemma-realpow-diff-sumr2 h*
right-diff-distrib [*symmetric*] *mult-assoc*
del: *power-Suc setsum-op-ivl-Suc of-nat-Suc*)
apply (*subst lemma-realpow-rev-sumr*)
apply (*subst sumr-diff-mult-const2*)
apply *simp*
apply (*simp only*: *lemma-termdiff1 setsum-right-distrib*)
apply (*rule setsum-cong* [*OF refl*])

apply (*simp add*: *diff-minus* [*symmetric*] *less-iff-Suc-add*)
apply (*clarify*)
apply (*simp add*: *setsum-right-distrib lemma-realpow-diff-sumr2 mult-ac*
del: *setsum-op-ivl-Suc power-Suc*)
apply (*subst mult-assoc* [*symmetric*], *subst power-add* [*symmetric*])
apply (*simp add*: *mult-ac*)
done

lemma *real-setsum-nat-ivl-bounded2*:
fixes K :: $'a$::*ordered-semidom*
assumes *f*: $\bigwedge p$::*nat*. $p < n \Longrightarrow f\ p \leq K$
assumes *K*: $0 \leq K$
shows *setsum f* $\{0..<n-k\} \leq$ *of-nat n* $*$ K
apply (*rule order-trans* [*OF setsum-mono*])
apply (*rule f*, *simp*)
apply (*simp add*: *mult-right-mono K*)
done

lemma *lemma-termdiff3*:
fixes $h\ z$ :: $'a$::{*real-normed-field*,*recpower*}
assumes *1*: $h \neq 0$
assumes *2*: *norm* $z \leq K$
assumes *3*: *norm* $(z + h) \leq K$
shows *norm* $(((z + h)\ \hat{}\ n - z\ \hat{}\ n)\ /\ h - \textit{of-nat}\ n * z\ \hat{}\ (n - \textit{Suc}\ 0))$
$\leq$ *of-nat n* $*$ *of-nat* $(n - \textit{Suc}\ 0) * K\ \hat{}\ (n - 2) *$ *norm h*
proof −
have *norm* $(((z + h)\ \hat{}\ n - z\ \hat{}\ n)\ /\ h - \textit{of-nat}\ n * z\ \hat{}\ (n - \textit{Suc}\ 0)) =$
norm $(\sum p = 0..<n - \textit{Suc}\ 0.\ \sum q = 0..<n - \textit{Suc}\ 0 - p.$
$(z + h)\ \hat{}\ q * z\ \hat{}\ (n - 2 - q)) *$ *norm h*
apply (*subst lemma-termdiff2* [*OF 1*])
apply (*subst norm-mult*)
apply (*rule mult-commute*)
done
also have ... $\leq$ *of-nat n* $*$ (*of-nat* $(n - \textit{Suc}\ 0) * K\ \hat{}\ (n - 2)) *$ *norm h*
proof (*rule mult-right-mono* [*OF - norm-ge-zero*])
from *norm-ge-zero 2* **have** *K*: $0 \leq K$ **by** (*rule order-trans*)
have *le-Kn*: $\bigwedge i\ j\ n.\ i + j = n \Longrightarrow$ *norm* $((z + h)\ \hat{}\ i * z\ \hat{}\ j) \leq K\ \hat{}\ n$
apply (*erule subst*)
apply (*simp only*: *norm-mult norm-power power-add*)
apply (*intro mult-mono power-mono 2 3 norm-ge-zero zero-le-power K*)
done
show *norm* $(\sum p = 0..<n - \textit{Suc}\ 0.\ \sum q = 0..<n - \textit{Suc}\ 0 - p.$
$(z + h)\ \hat{}\ q * z\ \hat{}\ (n - 2 - q))$
$\leq$ *of-nat n* $*$ (*of-nat* $(n - \textit{Suc}\ 0) * K\ \hat{}\ (n - 2))$
apply (*intro*
order-trans [*OF norm-setsum*]
real-setsum-nat-ivl-bounded2
mult-nonneg-nonneg
zero-le-imp-of-nat

```
        zero-le-power K)
      apply (rule le-Kn, simp)
      done
  qed
  also have ... = of-nat n * of-nat (n - Suc 0) * K ^ (n - 2) * norm h
    by (simp only: mult-assoc)
  finally show ?thesis .
qed

lemma lemma-termdiff4:
  fixes f :: 'a::{real-normed-field,recpower} ⇒
             'b::real-normed-vector
  assumes k: 0 < (k::real)
  assumes le: ⋀h. [[h ≠ 0; norm h < k]] ⟹ norm (f h) ≤ K * norm h
  shows f -- 0 --> 0
unfolding LIM-def diff-0-right
proof (safe)
  let ?h = of-real (k / 2)::'a
  have ?h ≠ 0 and norm ?h < k using k by simp-all
  hence norm (f ?h) ≤ K * norm ?h by (rule le)
  hence 0 ≤ K * norm ?h by (rule order-trans [OF norm-ge-zero])
  hence zero-le-K: 0 ≤ K using k by (simp add: zero-le-mult-iff)

  fix r::real assume r: 0 < r
  show ∃s. 0 < s ∧ (∀x. x ≠ 0 ∧ norm x < s ⟶ norm (f x) < r)
  proof (cases)
    assume K = 0
    with k r le have 0 < k ∧ (∀x. x ≠ 0 ∧ norm x < k ⟶ norm (f x) < r)
      by simp
    thus ∃s. 0 < s ∧ (∀x. x ≠ 0 ∧ norm x < s ⟶ norm (f x) < r) ..
  next
    assume K-neq-zero: K ≠ 0
    with zero-le-K have K: 0 < K by simp
    show ∃s. 0 < s ∧ (∀x. x ≠ 0 ∧ norm x < s ⟶ norm (f x) < r)
    proof (rule exI, safe)
      from k r K show 0 < min k (r * inverse K / 2)
        by (simp add: mult-pos-pos positive-imp-inverse-positive)
    next
      fix x::'a
      assume x1: x ≠ 0 and x2: norm x < min k (r * inverse K / 2)
      from x2 have x3: norm x < k and x4: norm x < r * inverse K / 2
        by simp-all
      from x1 x3 le have norm (f x) ≤ K * norm x by simp
      also from x4 K have K * norm x < K * (r * inverse K / 2)
        by (rule mult-strict-left-mono)
      also have ... = r / 2
        using K-neq-zero by simp
      also have r / 2 < r
        using r by simp
```

```
    finally show norm (f x) < r .
  qed
 qed
qed

lemma lemma-termdiff5:
 fixes g :: 'a::{recpower,real-normed-field} ⇒
            nat ⇒ 'b::banach
 assumes k: 0 < (k::real)
 assumes f: summable f
 assumes le: ⋀h n. ⟦h ≠ 0; norm h < k⟧ ⟹ norm (g h n) ≤ f n * norm h
 shows (λh. suminf (g h)) -- 0 --> 0
proof (rule lemma-termdiff4 [OF k])
 fix h::'a assume h ≠ 0 and norm h < k
 hence A: ∀ n. norm (g h n) ≤ f n * norm h
   by (simp add: le)
 hence ∃ N. ∀ n≥N. norm (norm (g h n)) ≤ f n * norm h
   by simp
 moreover from f have B: summable (λn. f n * norm h)
   by (rule summable-mult2)
 ultimately have C: summable (λn. norm (g h n))
   by (rule summable-comparison-test)
 hence norm (suminf (g h)) ≤ (∑ n. norm (g h n))
   by (rule summable-norm)
 also from A C B have (∑ n. norm (g h n)) ≤ (∑ n. f n * norm h)
   by (rule summable-le)
 also from f have (∑ n. f n * norm h) = suminf f * norm h
   by (rule suminf-mult2 [symmetric])
 finally show norm (suminf (g h)) ≤ suminf f * norm h .
qed
```

FIXME: Long proofs

```
lemma termdiffs-aux:
 fixes x :: 'a::{recpower,real-normed-field,banach}
 assumes 1: summable (λn. diffs (diffs c) n * K ^ n)
 assumes 2: norm x < norm K
 shows (λh. ∑ n. c n * (((x + h) ^ n - x ^ n) / h
          - of-nat n * x ^ (n - Suc 0))) -- 0 --> 0
proof -
 from dense [OF 2]
 obtain r where r1: norm x < r and r2: r < norm K by fast
 from norm-ge-zero r1 have r: 0 < r
   by (rule order-le-less-trans)
 hence r-neq-0: r ≠ 0 by simp
 show ?thesis
 proof (rule lemma-termdiff5)
   show 0 < r - norm x using r1 by simp
 next
   from r r2 have norm (of-real r::'a) < norm K
```

by *simp*
with *1* **have** *summable* (λ*n*. *norm* (*diffs* (*diffs c*) *n* ∗ (*of-real r* ^ *n*)))
by (*rule powser-insidea*)
hence *summable* (λ*n*. *diffs* (*diffs* (λ*n*. *norm* (*c n*))) *n* ∗ *r* ^ *n*)
using *r*
by (*simp add*: *diffs-def norm-mult norm-power del*: *of-nat-Suc*)
hence *summable* (λ*n*. *of-nat n* ∗ *diffs* (λ*n*. *norm* (*c n*)) *n* ∗ *r* ^ (*n* − *Suc 0*))
by (*rule diffs-equiv* [*THEN sums-summable*])
also have (λ*n*. *of-nat n* ∗ *diffs* (λ*n*. *norm* (*c n*)) *n* ∗ *r* ^ (*n* − *Suc 0*))
= (λ*n*. *diffs* (%*m*. *of-nat* (*m* − *Suc 0*) ∗ *norm* (*c m*) ∗ *inverse r*) *n* ∗ (*r* ^ *n*))
apply (*rule ext*)
apply (*simp add*: *diffs-def*)
apply (*case-tac n*, *simp-all add*: *r-neq-0*)
done
finally have *summable*
(λ*n*. *of-nat n* ∗ (*of-nat* (*n* − *Suc 0*) ∗ *norm* (*c n*) ∗ *inverse r*) ∗ *r* ^ (*n* − *Suc 0*))
by (*rule diffs-equiv* [*THEN sums-summable*])
also have
(λ*n*. *of-nat n* ∗ (*of-nat* (*n* − *Suc 0*) ∗ *norm* (*c n*) ∗ *inverse r*) ∗
r ^ (*n* − *Suc 0*)) =
(λ*n*. *norm* (*c n*) ∗ *of-nat n* ∗ *of-nat* (*n* − *Suc 0*) ∗ *r* ^ (*n* − *2*))
apply (*rule ext*)
apply (*case-tac n*, *simp*)
apply (*case-tac nat*, *simp*)
apply (*simp add*: *r-neq-0*)
done
finally show
summable (λ*n*. *norm* (*c n*) ∗ *of-nat n* ∗ *of-nat* (*n* − *Suc 0*) ∗ *r* ^ (*n* − *2*)) .
next
fix *h*::*'a* **and** *n*::*nat*
assume *h*: *h* ≠ *0*
assume *norm h* < *r* − *norm x*
hence *norm x* + *norm h* < *r* **by** *simp*
with *norm-triangle-ineq* **have** *xh*: *norm* (*x* + *h*) < *r*
by (*rule order-le-less-trans*)
show *norm* (*c n* ∗ (((*x* + *h*) ^ *n* − *x* ^ *n*) / *h* − *of-nat n* ∗ *x* ^ (*n* − *Suc 0*)))
≤ *norm* (*c n*) ∗ *of-nat n* ∗ *of-nat* (*n* − *Suc 0*) ∗ *r* ^ (*n* − *2*) ∗ *norm h*
apply (*simp only*: *norm-mult mult-assoc*)
apply (*rule mult-left-mono* [*OF - norm-ge-zero*])
apply (*simp* (*no-asm*) *add*: *mult-assoc* [*symmetric*])
apply (*rule lemma-termdiff3*)
apply (*rule h*)
apply (*rule r1* [*THEN order-less-imp-le*])
apply (*rule xh* [*THEN order-less-imp-le*])
done
qed
qed

lemma *termdiffs*:
fixes $K\ x :: 'a::\{recpower,real\text{-}normed\text{-}field,banach\}$
assumes *1*: $summable\ (\lambda n.\ c\ n * K\ \hat{}\ n)$
assumes *2*: $summable\ (\lambda n.\ (diffs\ c)\ n * K\ \hat{}\ n)$
assumes *3*: $summable\ (\lambda n.\ (diffs\ (diffs\ c))\ n * K\ \hat{}\ n)$
assumes *4*: $norm\ x < norm\ K$
shows $DERIV\ (\lambda x.\ \sum n.\ c\ n * x\ \hat{}\ n)\ x :> (\sum n.\ (diffs\ c)\ n * x\ \hat{}\ n)$
unfolding *deriv-def*
proof (*rule LIM-zero-cancel*)
show $(\lambda h.\ (suminf\ (\lambda n.\ c\ n * (x + h)\ \hat{}\ n) - suminf\ (\lambda n.\ c\ n * x\ \hat{}\ n))\ /\ h$
$- suminf\ (\lambda n.\ diffs\ c\ n * x\ \hat{}\ n)) -- 0 --> 0$
proof (*rule LIM-equal2*)
show $0 < norm\ K - norm\ x$ **using** *4* **by** (*simp add*: *less-diff-eq*)
next
fix $h :: 'a$
assume $h \neq 0$
assume $norm\ (h - 0) < norm\ K - norm\ x$
hence $norm\ x + norm\ h < norm\ K$ **by** *simp*
hence *5*: $norm\ (x + h) < norm\ K$
by (*rule norm-triangle-ineq* [*THEN order-le-less-trans*])
have *A*: $summable\ (\lambda n.\ c\ n * x\ \hat{}\ n)$
by (*rule powser-inside* [*OF 1 4*])
have *B*: $summable\ (\lambda n.\ c\ n * (x + h)\ \hat{}\ n)$
by (*rule powser-inside* [*OF 1 5*])
have *C*: $summable\ (\lambda n.\ diffs\ c\ n * x\ \hat{}\ n)$
by (*rule powser-inside* [*OF 2 4*])
show $((\sum n.\ c\ n * (x + h)\ \hat{}\ n) - (\sum n.\ c\ n * x\ \hat{}\ n))\ /\ h$
$- (\sum n.\ diffs\ c\ n * x\ \hat{}\ n) =$
$(\sum n.\ c\ n * (((x + h)\ \hat{}\ n - x\ \hat{}\ n)\ /\ h - of\text{-}nat\ n * x\ \hat{}\ (n - Suc\ 0)))$
apply (*subst sums-unique* [*OF diffs-equiv* [*OF C*]])
apply (*subst suminf-diff* [*OF B A*])
apply (*subst suminf-divide* [*symmetric*])
apply (*rule summable-diff* [*OF B A*])
apply (*subst suminf-diff*)
apply (*rule summable-divide*)
apply (*rule summable-diff* [*OF B A*])
apply (*rule sums-summable* [*OF diffs-equiv* [*OF C*]])
apply (*rule arg-cong* [**where** *f*=*suminf*], *rule ext*)
apply (*simp add*: *algebra-simps*)
done
next
show $(\lambda h.\ \sum n.\ c\ n * (((x + h)\ \hat{}\ n - x\ \hat{}\ n)\ /\ h -$
$of\text{-}nat\ n * x\ \hat{}\ (n - Suc\ 0))) -- 0 --> 0$
by (*rule termdiffs-aux* [*OF 3 4*])
qed
qed

63.4 Some properties of factorials

lemma *real-of-nat-fact-not-zero* [*simp*]: *real* (*fact n*) ≠ *0*
by *auto*

lemma *real-of-nat-fact-gt-zero* [*simp*]: *0* < *real*(*fact n*)
by *auto*

lemma *real-of-nat-fact-ge-zero* [*simp*]: *0* ≤ *real*(*fact n*)
by *simp*

lemma *inv-real-of-nat-fact-gt-zero* [*simp*]: *0* < *inverse* (*real* (*fact n*))
by (*auto simp add*: *positive-imp-inverse-positive*)

lemma *inv-real-of-nat-fact-ge-zero* [*simp*]: *0* ≤ *inverse* (*real* (*fact n*))
by (*auto intro*: *order-less-imp-le*)

63.5 Derivability of power series

lemma *DERIV-series'*: **fixes** *f* :: *real* ⇒ *nat* ⇒ *real*
assumes *DERIV-f*: ⋀ *n*. *DERIV* (λ *x*. *f x n*) *x0* :> (*f' x0 n*)
and *allf-summable*: ⋀ *x*. *x* ∈ {*a* <..< *b*} ⟹ *summable* (*f x*) **and** *x0-in-I*: *x0* ∈ {*a* <..< *b*}
and *summable* (*f' x0*)
and *summable L* **and** *L-def*: ⋀ *n x y*. ⟦ *x* ∈ { *a* <..< *b*} ; *y* ∈ { *a* <..< *b*} ⟧ ⟹ | *f x n* − *f y n* | ≤ *L n* ∗ | *x* − *y* |
shows *DERIV* (λ *x*. *suminf* (*f x*)) *x0* :> (*suminf* (*f' x0*))
unfolding *deriv-def*
proof (*rule LIM-I*)
fix *r* :: *real* **assume** *0* < *r* **hence** *0* < *r*/*3* **by** *auto*

obtain *N-L* **where** *N-L*: ⋀ *n*. *N-L* ≤ *n* ⟹ | ∑ *i*. *L* (*i* + *n*) | < *r*/*3*
using *suminf-exist-split*[*OF* ‹*0* < *r*/*3*› ‹*summable L*›] **by** *auto*

obtain *N-f'* **where** *N-f'*: ⋀ *n*. *N-f'* ≤ *n* ⟹ | ∑ *i*. *f' x0* (*i* + *n*) | < *r*/*3*
using *suminf-exist-split*[*OF* ‹*0* < *r*/*3*› ‹*summable* (*f' x0*)›] **by** *auto*

let *?N* = *Suc* (*max N-L N-f'*)
have | ∑ *i*. *f' x0* (*i* + *?N*) | < *r*/*3* (**is** *?f'-part* < *r*/*3*) **and**
L-estimate: | ∑ *i*. *L* (*i* + *?N*) | < *r*/*3* **using** *N-L*[*of ?N*] **and** *N-f'* [*of ?N*] **by** *auto*

let *?diff i x* = (*f* (*x0* + *x*) *i* − *f x0 i*) / *x*

let *?r* = *r* / (*3* ∗ *real ?N*)
have *0* < *3* ∗ *real ?N* **by** *auto*
from *divide-pos-pos*[*OF* ‹*0* < *r*› *this*]
have *0* < *?r* **.**

let *?s n* = *SOME s*. *0* < *s* ∧ (∀ *x*. *x* ≠ *0* ∧ | *x* | < *s* ⟶ | *?diff n x* − *f' x0 n*

$| < ?r)$
def $S' \equiv Min\ (?s\ `\ \{\ 0\ ..<\ ?N\ \})$

have $0 < S'$ **unfolding** *S'-def*
proof (*rule iffD2*[*OF Min-gr-iff*])
show $\forall\ x \in (?s\ `\ \{\ 0\ ..<\ ?N\ \}).\ 0 < x$
proof (*rule ballI*)
fix x **assume** $x \in ?s\ `\ \{0..<?N\}$
then obtain n **where** $x = ?s\ n$ **and** $n \in \{0..<?N\}$ **using** *image-iff*[*THEN iffD1*] **by** *blast*
from *DERIV-D*[*OF DERIV-f*[**where** *n=n*], *THEN LIM-D*, *OF* ‹$0 < ?r$›, *unfolded real-norm-def*]
obtain s **where** *s-bound*: $0 < s \wedge (\forall x.\ x \neq 0 \wedge |x| < s \longrightarrow |?diff\ n\ x - f'\ x0\ n| < ?r)$ **by** *auto*
have $0 < ?s\ n$ **by** (*rule someI2*[**where** *a=s*], *auto simp add*: *s-bound*)
thus $0 < x$ **unfolding** ‹$x = ?s\ n$› **.**
qed
qed *auto*

def $S \equiv min\ (min\ (x0 - a)\ (b - x0))\ S'$
hence $0 < S$ **and** *S-a*: $S \leq x0 - a$ **and** *S-b*: $S \leq b - x0$ **and** $S \leq S'$ **using** *x0-in-I* **and** ‹$0 < S'$›
by *auto*

{ fix x **assume** $x \neq 0$ **and** $| x | < S$
hence *x-in-I*: $x0 + x \in \{\ a\ <..<\ b\ \}$ **using** *S-a S-b* **by** *auto*

note *diff-smbl* = *summable-diff*[*OF allf-summable*[*OF x-in-I*] *allf-summable*[*OF x0-in-I*]]
note *div-smbl* = *summable-divide*[*OF diff-smbl*]
note *all-smbl* = *summable-diff*[*OF div-smbl* ‹*summable* $(f'\ x0)$›]
note *ign* = *summable-ignore-initial-segment*[**where** *k=?N*]
note *diff-shft-smbl* = *summable-diff*[*OF ign*[*OF allf-summable*[*OF x-in-I*]] *ign*[*OF allf-summable*[*OF x0-in-I*]]]
note *div-shft-smbl* = *summable-divide*[*OF diff-shft-smbl*]
note *all-shft-smbl* = *summable-diff*[*OF div-smbl ign*[*OF* ‹*summable* $(f'\ x0)$›]]

{ fix n
have $| ?diff\ (n + ?N)\ x | \leq L\ (n + ?N) * | (x0 + x) - x0 | / | x |$
using *divide-right-mono*[*OF L-def*[*OF x-in-I x0-in-I*] *abs-ge-zero*] **unfolding** *abs-divide* **.**
hence $| (|\ ?diff\ (n + ?N)\ x\ |) | \leq L\ (n + ?N)$ **using** ‹$x \neq 0$› **by** *auto*
} note *L-ge* = *summable-le2*[*OF allI*[*OF this*] *ign*[*OF* ‹*summable L*›]]
from *order-trans*[*OF summable-rabs*[*OF conjunct1*[*OF L-ge*]] *L-ge*[*THEN conjunct2*]]
have $| \sum\ i.\ ?diff\ (i + ?N)\ x | \leq (\sum\ i.\ L\ (i + ?N))$ **.**
hence $| \sum\ i.\ ?diff\ (i + ?N)\ x | \leq r\ /\ 3$ (**is** *?L-part* $\leq r/3$) **using** *L-estimate* **by** *auto*

have $|\sum n \in \{\ 0\ ..<\ ?N\}.\ ?diff\ n\ x - f'\ x0\ n\ | \leq (\sum n \in \{\ 0\ ..<\ ?N\}.\ |?diff\ n\ x - f'\ x0\ n\ |)$ **..**
also have $\ldots < (\sum n \in \{\ 0\ ..<\ ?N\}.\ ?r)$
proof (*rule setsum-strict-mono*)
fix n **assume** $n \in \{\ 0\ ..<\ ?N\}$
have $|\ x\ | < S$ **using** ‹$|\ x\ | < S$› **.**
also have $S \leq S'$ **using** ‹$S \leq S'$› **.**
also have $S' \leq ?s\ n$ **unfolding** S'*-def*
proof (*rule Min-le-iff*[*THEN iffD2*])
have $?s\ n \in (?s\ `\ \{0..<?N\}) \wedge ?s\ n \leq ?s\ n$ **using** ‹$n \in \{\ 0\ ..<\ ?N\}$› **by** *auto*
thus $\exists\ a \in (?s\ `\ \{0..<?N\}).\ a \leq ?s\ n$ **by** *blast*
qed *auto*
finally have $|\ x\ | < ?s\ n$ **.**

from *DERIV-D*[*OF DERIV-f*[**where** n=n], *THEN LIM-D*, *OF* ‹$0 < ?r$›, *unfolded real-norm-def diff-0-right*, *unfolded some-eq-ex*[*symmetric*], *THEN conjunct2*]
have $\forall x.\ x \neq 0 \wedge |x| < ?s\ n \longrightarrow |?diff\ n\ x - f'\ x0\ n| < ?r$ **.**
with ‹$x \neq 0$› **and** ‹$|x| < ?s\ n$›
show $|?diff\ n\ x - f'\ x0\ n| < ?r$ **by** *blast*
qed *auto*
also have $\ldots =$ *of-nat* (*card* $\{0\ ..<\ ?N\}) * ?r$ **by** (*rule setsum-constant*)
also have $\ldots =$ *real* $?N * ?r$ **unfolding** *real-eq-of-nat* **by** *auto*
also have $\ldots = r/3$ **by** *auto*
finally have $|\sum n \in \{\ 0\ ..<\ ?N\}.\ ?diff\ n\ x - f'\ x0\ n\ | < r\ /\ 3$ (**is** *?diff-part* $< r\ /\ 3$) **.**

from *suminf-diff*[*OF allf-summable*[*OF x-in-I*] *allf-summable*[*OF x0-in-I*]]
have $|\ (suminf\ (f\ (x0 + x)) - (suminf\ (f\ x0)))\ /\ x - suminf\ (f'\ x0)\ | =$
$|\ \sum n.\ ?diff\ n\ x - f'\ x0\ n\ |$ **unfolding** *suminf-diff*[*OF div-smbl* ‹*summable* $(f'\ x0)$›, *symmetric*] **using** *suminf-divide*[*OF diff-smbl*, *symmetric*] **by** *auto*
also have $\ldots \leq ?diff\text{-}part + |\ (\sum n.\ ?diff\ (n + ?N)\ x) - (\sum\ n.\ f'\ x0\ (n + ?N))\ |$ **unfolding** *suminf-split-initial-segment*[*OF all-smbl*, **where** k=$?N$] **unfolding** *suminf-diff*[*OF div-shft-smbl ign*[*OF* ‹*summable* $(f'\ x0)$›]] **by** (*rule abs-triangle-ineq*)
also have $\ldots \leq ?diff\text{-}part + ?L\text{-}part + ?f'\text{-}part$ **using** *abs-triangle-ineq4* **by** *auto*
also have $\ldots < r\ /3 + r/3 + r/3$
using ‹*?diff-part* $< r/3$› ‹*?L-part* $\leq r/3$› **and** ‹*?f'-part* $< r/3$› **by** *auto*
finally have $|\ (suminf\ (f\ (x0 + x)) - (suminf\ (f\ x0)))\ /\ x - suminf\ (f'\ x0)\ | < r$
by *auto*
} thus $\exists\ s > 0.\ \forall\ x.\ x \neq 0 \wedge norm\ (x - 0) < s \longrightarrow$
$norm\ (((\sum n.\ f\ (x0 + x)\ n) - (\sum n.\ f\ x0\ n))\ /\ x - (\sum n.\ f'\ x0\ n)) < r$ **using** ‹$0 < S$›
unfolding *real-norm-def diff-0-right* **by** *blast*
qed

lemma *DERIV-power-series'*: **fixes** $f :: nat \Rightarrow real$
assumes *converges*: $\bigwedge x.\ x \in \{-R <..< R\} \Longrightarrow summable\ (\lambda\ n.\ f\ n * real\ (Suc\ n) * x\hat{}n)$
and *x0-in-I*: $x0 \in \{-R <..< R\}$ **and** $0 < R$
shows $DERIV\ (\lambda\ x.\ (\sum\ n.\ f\ n * x\hat{}(Suc\ n)))\ x0 :> (\sum\ n.\ f\ n * real\ (Suc\ n) * x0\hat{}n)$
(**is** $DERIV\ (\lambda\ x.\ (suminf\ (?f\ x)))\ x0 :> (suminf\ (?f'\ x0))$)
proof −
{ fix R' **assume** $0 < R'$ **and** $R' < R$ **and** $-R' < x0$ **and** $x0 < R'$
hence $x0 \in \{-R' <..< R'\}$ **and** $R' \in \{-R <..< R\}$ **and** $x0 \in \{-R <..< R\}$ **by** *auto*
have $DERIV\ (\lambda\ x.\ (suminf\ (?f\ x)))\ x0 :> (suminf\ (?f'\ x0))$
proof (*rule DERIV-series'*)
show $summable\ (\lambda\ n.\ |f\ n * real\ (Suc\ n) * R'\hat{}n|)$
proof −
have $(R' + R) / 2 < R$ **and** $0 < (R' + R) / 2$ **using** ‹$0 < R'$› ‹$0 < R$› ‹$R' < R$› **by** *auto*
hence *in-Rball*: $(R' + R) / 2 \in \{-R <..< R\}$ **using** ‹$R' < R$› **by** *auto*
have $norm\ R' < norm\ ((R' + R) / 2)$ **using** ‹$0 < R'$› ‹$0 < R$› ‹$R' < R$› **by** *auto*
from *powser-insidea*[*OF converges*[*OF in-Rball*] *this*] **show** *?thesis* **by** *auto*
qed
{ fix $n\ x\ y$ **assume** $x \in \{-R' <..< R'\}$ **and** $y \in \{-R' <..< R'\}$
show $|?f\ x\ n - ?f\ y\ n| \leq |f\ n * real\ (Suc\ n) * R'\hat{}n| * |x-y|$
proof −
have $|f\ n * x\ \hat{}\ (Suc\ n) - f\ n * y\ \hat{}\ (Suc\ n)| = (|f\ n| * |x-y|) * |\sum p = 0..<Suc\ n.\ x\ \hat{}\ p * y\ \hat{}\ (n - p)|$
unfolding *right-diff-distrib*[*symmetric*] *lemma-realpow-diff-sumr2 abs-mult* **by** *auto*
also have $\ldots \leq (|f\ n| * |x-y|) * (|real\ (Suc\ n)| * |R'\ \hat{}\ n|)$
proof (*rule mult-left-mono*)
have $|\sum p = 0..<Suc\ n.\ x\ \hat{}\ p * y\ \hat{}\ (n - p)| \leq (\sum p = 0..<Suc\ n.\ |x\ \hat{}\ p * y\ \hat{}\ (n - p)|)$ **by** (*rule setsum-abs*)
also have $\ldots \leq (\sum p = 0..<Suc\ n.\ R'\ \hat{}\ n)$
proof (*rule setsum-mono*)
fix p **assume** $p \in \{0..<Suc\ n\}$ **hence** $p \leq n$ **by** *auto*
{ fix n **fix** $x :: real$ **assume** $x \in \{-R'<..<R'\}$
hence $|x| \leq R'$ **by** *auto*
hence $|x\hat{}n| \leq R'\hat{}n$ **unfolding** *power-abs* **by** (*rule power-mono*, *auto*)
} from *mult-mono*[*OF this*[*OF* ‹$x \in \{-R'<..<R'\}$›, *of p*] *this*[*OF* ‹$y \in \{-R'<..<R'\}$›, *of n−p*]] ‹$0 < R'$›
have $|x\hat{}p * y\hat{}(n-p)| \leq R'\hat{}p * R'\hat{}(n-p)$ **unfolding** *abs-mult* **by** *auto*
thus $|x\hat{}p * y\hat{}(n-p)| \leq R'\hat{}n$ **unfolding** *power-add*[*symmetric*] **using** ‹$p \leq n$› **by** *auto*
qed
also have $\ldots = real\ (Suc\ n) * R'\ \hat{}\ n$ **unfolding** *setsum-constant card-atLeastLessThan real-of-nat-def* **by** *auto*
finally show $|\sum p = 0..<Suc\ n.\ x\ \hat{}\ p * y\ \hat{}\ (n - p)| \leq |real\ (Suc\ n)| * |R'\ \hat{}\ n|$ **unfolding** *abs-real-of-nat-cancel abs-of-nonneg*[*OF zero-le-power*[*OF*

less-imp-le[*OF* ‹*0* < *R'*›]]] .
show *0* ≤ |*f n*| * |*x* − *y*| **unfolding** *abs-mult*[*symmetric*] **by** *auto*
qed
also have ... = |*f n* * *real* (*Suc n*) * *R'* ^ *n*| * |*x* − *y*| **unfolding** *abs-mult real-mult-assoc*[*symmetric*] **by** *algebra*
finally show *?thesis* .
qed }
{ fix *n*
from *DERIV-pow*[*of Suc n x0*, *THEN DERIV-cmult*[**where** *c*=*f n*]]
show *DERIV* (λ *x*. *?f x n*) *x0* :> (*?f' x0 n*) **unfolding** *real-mult-assoc* **by** *auto* }
{ fix *x* **assume** *x* ∈ {−*R'* <..< *R'*} **hence** *R'* ∈ {−*R* <..< *R*} **and** *norm x* < *norm R'* **using** *assms* ‹*R'* < *R*› **by** *auto*
have *summable* (λ *n*. *f n* * *x*^*n*)
proof (*rule summable-le2*[*THEN conjunct1*, *OF - powser-insidea*[*OF converges*[*OF* ‹*R'* ∈ {−*R* <..< *R*}›] ‹*norm x* < *norm R'*›]], *rule allI*)
fix *n*
have *le*: |*f n*| * *1* ≤ |*f n*| * *real* (*Suc n*) **by** (*rule mult-left-mono*, *auto*)
show |*f n* * *x* ^ *n*| ≤ *norm* (*f n* * *real* (*Suc n*) * *x* ^ *n*) **unfolding** *real-norm-def abs-mult*
by (*rule mult-right-mono*, *auto simp add*: *le*[*unfolded mult-1-right*])
qed
from *this*[*THEN summable-mult2*[**where** *c*=*x*], *unfolded real-mult-assoc*, *unfolded real-mult-commute*]
show *summable* (*?f x*) **by** *auto* }
show *summable* (*?f' x0*) **using** *converges*[*OF* ‹*x0* ∈ {−*R* <..< *R*}›] .
show *x0* ∈ {−*R'* <..< *R'*} **using** ‹*x0* ∈ {−*R'* <..< *R'*}› .
qed
} **note** *for-subinterval* = *this*
let *?R* = (*R* + |*x0*|) / *2*
have |*x0*| < *?R* **using** *assms* **by** *auto*
hence − *?R* < *x0*
proof (*cases x0* < *0*)
case *True*
hence − *x0* < *?R* **using** ‹|*x0*| < *?R*› **by** *auto*
thus *?thesis* **unfolding** *neg-less-iff-less*[*symmetric*, *of* − *x0*] **by** *auto*
next
case *False*
have − *?R* < *0* **using** *assms* **by** *auto*
also have ... ≤ *x0* **using** *False* **by** *auto*
finally show *?thesis* .
qed
hence *0* < *?R* *?R* < *R* − *?R* < *x0* **and** *x0* < *?R* **using** *assms* **by** *auto*
from *for-subinterval*[*OF this*]
show *?thesis* .
qed

63.6 Exponential Function

definition
exp :: *'a* ⇒ *'a*::{*recpower*,*real-normed-field*,*banach*} **where**
exp x = (∑ *n*. *x* ^ *n* /$_R$ *real* (*fact n*))

lemma *summable-exp-generic*:
fixes *x* :: *'a*::{*real-normed-algebra-1*,*recpower*,*banach*}
defines *S-def*: *S* ≡ λ*n*. *x* ^ *n* /$_R$ *real* (*fact n*)
shows *summable S*
proof −
have *S-Suc*: ⋀*n*. *S* (*Suc n*) = (*x* ∗ *S n*) /$_R$ *real* (*Suc n*)
unfolding *S-def* **by** (*simp del*: *mult-Suc*)
obtain *r* :: *real* **where** *r0*: *0* < *r* **and** *r1*: *r* < *1*
using *dense* [*OF zero-less-one*] **by** *fast*
obtain *N* :: *nat* **where** *N*: *norm x* < *real N* ∗ *r*
using *reals-Archimedean3* [*OF r0*] **by** *fast*
from *r1* **show** *?thesis*
proof (*rule ratio-test* [*rule-format*])
fix *n* :: *nat*
assume *n*: *N* ≤ *n*
have *norm x* ≤ *real N* ∗ *r*
using *N* **by** (*rule order-less-imp-le*)
also have *real N* ∗ *r* ≤ *real* (*Suc n*) ∗ *r*
using *r0 n* **by** (*simp add*: *mult-right-mono*)
finally have *norm x* ∗ *norm* (*S n*) ≤ *real* (*Suc n*) ∗ *r* ∗ *norm* (*S n*)
using *norm-ge-zero* **by** (*rule mult-right-mono*)
hence *norm* (*x* ∗ *S n*) ≤ *real* (*Suc n*) ∗ *r* ∗ *norm* (*S n*)
by (*rule order-trans* [*OF norm-mult-ineq*])
hence *norm* (*x* ∗ *S n*) / *real* (*Suc n*) ≤ *r* ∗ *norm* (*S n*)
by (*simp add*: *pos-divide-le-eq mult-ac*)
thus *norm* (*S* (*Suc n*)) ≤ *r* ∗ *norm* (*S n*)
by (*simp add*: *S-Suc norm-scaleR inverse-eq-divide*)
qed
qed

lemma *summable-norm-exp*:
fixes *x* :: *'a*::{*real-normed-algebra-1*,*recpower*,*banach*}
shows *summable* (λ*n*. *norm* (*x* ^ *n* /$_R$ *real* (*fact n*)))
proof (*rule summable-norm-comparison-test* [*OF exI*, *rule-format*])
show *summable* (λ*n*. *norm x* ^ *n* /$_R$ *real* (*fact n*))
by (*rule summable-exp-generic*)
next
fix *n* **show** *norm* (*x* ^ *n* /$_R$ *real* (*fact n*)) ≤ *norm x* ^ *n* /$_R$ *real* (*fact n*)
by (*simp add*: *norm-scaleR norm-power-ineq*)
qed

lemma *summable-exp*: *summable* (%*n*. *inverse* (*real* (*fact n*)) ∗ *x* ^ *n*)
by (*insert summable-exp-generic* [**where** *x*=*x*], *simp*)

lemma *exp-converges*: $(\lambda n.\ x\ \hat{}\ n\ /_R\ real\ (fact\ n))\ sums\ exp\ x$
unfolding *exp-def* **by** (*rule summable-exp-generic* [*THEN summable-sums*])

lemma *exp-fdiffs*:
$diffs\ (\%n.\ inverse(real\ (fact\ n)))\ =\ (\%n.\ inverse(real\ (fact\ n)))$
by (*simp add*: *diffs-def mult-assoc* [*symmetric*] *real-of-nat-def of-nat-mult*
del: *mult-Suc of-nat-Suc*)

lemma *diffs-of-real*: $diffs\ (\lambda n.\ of\text{-}real\ (f\ n))\ =\ (\lambda n.\ of\text{-}real\ (diffs\ f\ n))$
by (*simp add*: *diffs-def*)

lemma *lemma-exp-ext*: $exp\ =\ (\lambda x.\ \sum n.\ x\ \hat{}\ n\ /_R\ real\ (fact\ n))$
by (*auto intro*!: *ext simp add*: *exp-def*)

lemma *DERIV-exp* [*simp*]: *DERIV exp x* :> $exp(x)$
apply (*simp add*: *exp-def*)
apply (*subst lemma-exp-ext*)
apply (*subgoal-tac DERIV* $(\lambda u.\ \sum n.\ of\text{-}real\ (inverse\ (real\ (fact\ n)))\ *\ u\ \hat{}\ n)\ x$
:> $(\sum n.\ diffs\ (\lambda n.\ of\text{-}real\ (inverse\ (real\ (fact\ n))))\ n\ *\ x\ \hat{}\ n)$)
apply (*rule-tac* [*2*] $K\ =\ of\text{-}real\ (1\ +\ norm\ x)$ **in** *termdiffs*)
apply (*simp-all only*: *diffs-of-real scaleR-conv-of-real exp-fdiffs*)
apply (*rule exp-converges* [*THEN sums-summable*, *unfolded scaleR-conv-of-real*])+
apply (*simp del*: *of-real-add*)
done

lemma *isCont-exp* [*simp*]: *isCont exp x*
by (*rule DERIV-exp* [*THEN DERIV-isCont*])

63.6.1 Properties of the Exponential Function

lemma *powser-zero*:
fixes $f\ ::\ nat\ \Rightarrow\ 'a$::{*real-normed-algebra-1*,*recpower*}
shows $(\sum n.\ f\ n\ *\ 0\ \hat{}\ n)\ =\ f\ 0$
proof −
have $(\sum n\ =\ 0..<1.\ f\ n\ *\ 0\ \hat{}\ n)\ =\ (\sum n.\ f\ n\ *\ 0\ \hat{}\ n)$
by (*rule sums-unique* [*OF series-zero*], *simp add*: *power-0-left*)
thus *?thesis* **unfolding** *One-nat-def* **by** *simp*
qed

lemma *exp-zero* [*simp*]: $exp\ 0\ =\ 1$
unfolding *exp-def* **by** (*simp add*: *scaleR-conv-of-real powser-zero*)

lemma *setsum-cl-ivl-Suc2*:
$(\sum i{=}m..Suc\ n.\ f\ i)\ =\ (if\ Suc\ n\ <\ m\ then\ 0\ else\ f\ m\ +\ (\sum i{=}m..n.\ f\ (Suc\ i)))$
by (*simp add*: *setsum-head-Suc setsum-shift-bounds-cl-Suc-ivl*
del: *setsum-cl-ivl-Suc*)

lemma *exp-series-add*:

fixes $x\ y :: 'a::\{real\text{-}field, recpower\}$
defines *S-def*: $S \equiv \lambda x\ n.\ x\ \hat{}\ n\ /_R\ real\ (fact\ n)$
shows $S\ (x + y)\ n = (\sum i{=}0..n.\ S\ x\ i * S\ y\ (n - i))$
proof (*induct n*)
case *0*
show *?case*
unfolding *S-def* **by** *simp*
next
case (*Suc n*)
have *S-Suc*: $\bigwedge x\ n.\ S\ x\ (Suc\ n) = (x * S\ x\ n)\ /_R\ real\ (Suc\ n)$
unfolding *S-def* **by** (*simp del*: *mult-Suc*)
hence *times-S*: $\bigwedge x\ n.\ x * S\ x\ n = real\ (Suc\ n) *_R S\ x\ (Suc\ n)$
by *simp*

have $real\ (Suc\ n) *_R S\ (x + y)\ (Suc\ n) = (x + y) * S\ (x + y)\ n$
by (*simp only*: *times-S*)
also have $\ldots = (x + y) * (\sum i{=}0..n.\ S\ x\ i * S\ y\ (n{-}i))$
by (*simp only*: *Suc*)
also have $\ldots = x * (\sum i{=}0..n.\ S\ x\ i * S\ y\ (n{-}i))$
$+\ y * (\sum i{=}0..n.\ S\ x\ i * S\ y\ (n{-}i))$
by (*rule left-distrib*)
also have $\ldots = (\sum i{=}0..n.\ (x * S\ x\ i) * S\ y\ (n{-}i))$
$+\ (\sum i{=}0..n.\ S\ x\ i * (y * S\ y\ (n{-}i)))$
by (*simp only*: *setsum-right-distrib mult-ac*)
also have $\ldots = (\sum i{=}0..n.\ real\ (Suc\ i) *_R (S\ x\ (Suc\ i) * S\ y\ (n{-}i)))$
$+\ (\sum i{=}0..n.\ real\ (Suc\ n{-}i) *_R (S\ x\ i * S\ y\ (Suc\ n{-}i)))$
by (*simp add*: *times-S Suc-diff-le*)
also have $(\sum i{=}0..n.\ real\ (Suc\ i) *_R (S\ x\ (Suc\ i) * S\ y\ (n{-}i))) =$
$(\sum i{=}0..Suc\ n.\ real\ i *_R (S\ x\ i * S\ y\ (Suc\ n{-}i)))$
by (*subst setsum-cl-ivl-Suc2*, *simp*)
also have $(\sum i{=}0..n.\ real\ (Suc\ n{-}i) *_R (S\ x\ i * S\ y\ (Suc\ n{-}i))) =$
$(\sum i{=}0..Suc\ n.\ real\ (Suc\ n{-}i) *_R (S\ x\ i * S\ y\ (Suc\ n{-}i)))$
by (*subst setsum-cl-ivl-Suc*, *simp*)
also have $(\sum i{=}0..Suc\ n.\ real\ i *_R (S\ x\ i * S\ y\ (Suc\ n{-}i))) +$
$(\sum i{=}0..Suc\ n.\ real\ (Suc\ n{-}i) *_R (S\ x\ i * S\ y\ (Suc\ n{-}i))) =$
$(\sum i{=}0..Suc\ n.\ real\ (Suc\ n) *_R (S\ x\ i * S\ y\ (Suc\ n{-}i)))$
by (*simp only*: *setsum-addf* [*symmetric*] *scaleR-left-distrib* [*symmetric*]
real-of-nat-add [*symmetric*], *simp*)
also have $\ldots = real\ (Suc\ n) *_R (\sum i{=}0..Suc\ n.\ S\ x\ i * S\ y\ (Suc\ n{-}i))$
by (*simp only*: *scaleR-right.setsum*)
finally show
$S\ (x + y)\ (Suc\ n) = (\sum i{=}0..Suc\ n.\ S\ x\ i * S\ y\ (Suc\ n - i))$
by (*simp add*: *scaleR-cancel-left del*: *setsum-cl-ivl-Suc*)
qed

lemma *exp-add*: $exp\ (x + y) = exp\ x * exp\ y$
unfolding *exp-def*
by (*simp only*: *Cauchy-product summable-norm-exp exp-series-add*)

lemma *mult-exp-exp*: *exp x ∗ exp y = exp (x + y)*
by (*rule exp-add* [*symmetric*])

lemma *exp-of-real*: *exp (of-real x) = of-real (exp x)*
unfolding *exp-def*
apply (*subst of-real.suminf*)
apply (*rule summable-exp-generic*)
apply (*simp add*: *scaleR-conv-of-real*)
done

lemma *exp-not-eq-zero* [*simp*]: *exp x ≠ 0*
proof
  have *exp x ∗ exp (− x) = 1* **by** (*simp add*: *mult-exp-exp*)
  also assume *exp x = 0*
  finally show *False* **by** *simp*
qed

lemma *exp-minus*: *exp (− x) = inverse (exp x)*
by (*rule inverse-unique* [*symmetric*], *simp add*: *mult-exp-exp*)

lemma *exp-diff*: *exp (x − y) = exp x / exp y*
  unfolding *diff-minus divide-inverse*
  by (*simp add*: *exp-add exp-minus*)

63.6.2 Properties of the Exponential Function on Reals

Comparisons of *exp x* with zero.

Proof: because every exponential can be seen as a square.

lemma *exp-ge-zero* [*simp*]: *0 ≤ exp (x*::*real)*
proof −
  have *0 ≤ exp (x/2) ∗ exp (x/2)* **by** *simp*
  thus *?thesis* **by** (*simp add*: *exp-add* [*symmetric*])
qed

lemma *exp-gt-zero* [*simp*]: *0 < exp (x*::*real)*
by (*simp add*: *order-less-le*)

lemma *not-exp-less-zero* [*simp*]: ¬ *exp (x*::*real) < 0*
by (*simp add*: *not-less*)

lemma *not-exp-le-zero* [*simp*]: ¬ *exp (x*::*real) ≤ 0*
by (*simp add*: *not-le*)

lemma *abs-exp-cancel* [*simp*]: *|exp x*::*real| = exp x*
by *simp*

lemma *exp-real-of-nat-mult*: *exp(real n ∗ x) = exp(x) ^ n*
apply (*induct n*)

apply (*auto simp add*: *real-of-nat-Suc right-distrib exp-add mult-commute*)
done

Strict monotonicity of exponential.

lemma *exp-ge-add-one-self-aux*: *0* ≤ (*x*::*real*) ==> (*1* + *x*) ≤ *exp*(*x*)
apply (*drule order-le-imp-less-or-eq*, *auto*)
apply (*simp add*: *exp-def*)
apply (*rule real-le-trans*)
apply (*rule-tac* [*2*] *n* = *2* **and** *f* = (%*n*. *inverse* (*real* (*fact n*)) ∗ *x* ^ *n*) **in** *series-pos-le*)
apply (*auto intro*: *summable-exp simp add*: *numeral-2-eq-2 zero-le-mult-iff*)
done

lemma *exp-gt-one*: *0* < (*x*::*real*) ⟹ *1* < *exp x*
proof −
  assume *x*: *0* < *x*
  hence *1* < *1* + *x* **by** *simp*
  also from *x* **have** *1* + *x* ≤ *exp x*
    by (*simp add*: *exp-ge-add-one-self-aux*)
  finally show *?thesis* **.**
qed

lemma *exp-less-mono*:
  fixes *x y* :: *real*
  assumes *x* < *y* **shows** *exp x* < *exp y*
proof −
  from ‹*x* < *y*› **have** *0* < *y* − *x* **by** *simp*
  hence *1* < *exp* (*y* − *x*) **by** (*rule exp-gt-one*)
  hence *1* < *exp y* / *exp x* **by** (*simp only*: *exp-diff*)
  thus *exp x* < *exp y* **by** *simp*
qed

lemma *exp-less-cancel*: *exp* (*x*::*real*) < *exp y* ==> *x* < *y*
apply (*simp add*: *linorder-not-le* [*symmetric*])
apply (*auto simp add*: *order-le-less exp-less-mono*)
done

lemma *exp-less-cancel-iff* [*iff*]: *exp* (*x*::*real*) < *exp y* ⟷ *x* < *y*
by (*auto intro*: *exp-less-mono exp-less-cancel*)

lemma *exp-le-cancel-iff* [*iff*]: *exp* (*x*::*real*) ≤ *exp y* ⟷ *x* ≤ *y*
by (*auto simp add*: *linorder-not-less* [*symmetric*])

lemma *exp-inj-iff* [*iff*]: *exp* (*x*::*real*) = *exp y* ⟷ *x* = *y*
by (*simp add*: *order-eq-iff*)

Comparisons of *exp x* with one.

lemma *one-less-exp-iff* [*simp*]: *1* < *exp* (*x*::*real*) ⟷ *0* < *x*
  using *exp-less-cancel-iff* [**where** *x*=*0* **and** *y*=*x*] **by** *simp*

lemma *exp-less-one-iff* [*simp*]: *exp* (*x*::*real*) < *1* ⟷ *x* < *0*
using *exp-less-cancel-iff* [**where** *x*=*x* **and** *y*=*0*] **by** *simp*

lemma *one-le-exp-iff* [*simp*]: *1* ≤ *exp* (*x*::*real*) ⟷ *0* ≤ *x*
using *exp-le-cancel-iff* [**where** *x*=*0* **and** *y*=*x*] **by** *simp*

lemma *exp-le-one-iff* [*simp*]: *exp* (*x*::*real*) ≤ *1* ⟷ *x* ≤ *0*
using *exp-le-cancel-iff* [**where** *x*=*x* **and** *y*=*0*] **by** *simp*

lemma *exp-eq-one-iff* [*simp*]: *exp* (*x*::*real*) = *1* ⟷ *x* = *0*
using *exp-inj-iff* [**where** *x*=*x* **and** *y*=*0*] **by** *simp*

lemma *lemma-exp-total*: *1* ≤ *y* ==> ∃ *x*. *0* ≤ *x* & *x* ≤ *y* − *1* & *exp*(*x*::*real*) = *y*
apply (*rule IVT*)
apply (*auto intro*: *isCont-exp simp add*: *le-diff-eq*)
apply (*subgoal-tac 1* + (*y* − *1*) ≤ *exp* (*y* − *1*))
apply *simp*
apply (*rule exp-ge-add-one-self-aux*, *simp*)
done

lemma *exp-total*: *0* < (*y*::*real*) ==> ∃ *x*. *exp x* = *y*
apply (*rule-tac x* = *1* **and** *y* = *y* **in** *linorder-cases*)
apply (*drule order-less-imp-le* [*THEN lemma-exp-total*])
apply (*rule-tac* [*2*] *x* = *0* **in** *exI*)
apply (*frule-tac* [*3*] *real-inverse-gt-one*)
apply (*drule-tac* [*4*] *order-less-imp-le* [*THEN lemma-exp-total*], *auto*)
apply (*rule-tac x* = −*x* **in** *exI*)
apply (*simp add*: *exp-minus*)
done

63.7 Natural Logarithm

definition
ln :: *real* => *real* **where**
ln x = (*THE u*. *exp u* = *x*)

lemma *ln-exp* [*simp*]: *ln* (*exp x*) = *x*
by (*simp add*: *ln-def*)

lemma *exp-ln* [*simp*]: *0* < *x* ⟹ *exp* (*ln x*) = *x*
by (*auto dest*: *exp-total*)

lemma *exp-ln-iff* [*simp*]: *exp* (*ln x*) = *x* ⟷ *0* < *x*
apply (*rule iffI*)
apply (*erule subst*, *rule exp-gt-zero*)
apply (*erule exp-ln*)
done

lemma *ln-unique*: $exp\ y = x \Longrightarrow ln\ x = y$
by (*erule subst*, *rule ln-exp*)

lemma *ln-one* [*simp*]: $ln\ 1 = 0$
by (*rule ln-unique*, *simp*)

lemma *ln-mult*: $[\![0 < x;\ 0 < y]\!] \Longrightarrow ln\ (x * y) = ln\ x + ln\ y$
by (*rule ln-unique*, *simp add*: *exp-add*)

lemma *ln-inverse*: $0 < x \Longrightarrow ln\ (inverse\ x) = -\ ln\ x$
by (*rule ln-unique*, *simp add*: *exp-minus*)

lemma *ln-div*: $[\![0 < x;\ 0 < y]\!] \Longrightarrow ln\ (x\ /\ y) = ln\ x - ln\ y$
by (*rule ln-unique*, *simp add*: *exp-diff*)

lemma *ln-realpow*: $0 < x \Longrightarrow ln\ (x\ \hat{}\ n) = real\ n * ln\ x$
by (*rule ln-unique*, *simp add*: *exp-real-of-nat-mult*)

lemma *ln-less-cancel-iff* [*simp*]: $[\![0 < x;\ 0 < y]\!] \Longrightarrow ln\ x < ln\ y \longleftrightarrow x < y$
by (*subst exp-less-cancel-iff* [*symmetric*], *simp*)

lemma *ln-le-cancel-iff* [*simp*]: $[\![0 < x;\ 0 < y]\!] \Longrightarrow ln\ x \leq ln\ y \longleftrightarrow x \leq y$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *ln-inj-iff* [*simp*]: $[\![0 < x;\ 0 < y]\!] \Longrightarrow ln\ x = ln\ y \longleftrightarrow x = y$
by (*simp add*: *order-eq-iff*)

lemma *ln-add-one-self-le-self* [*simp*]: $0 \leq x \Longrightarrow ln\ (1 + x) \leq x$
apply (*rule exp-le-cancel-iff* [*THEN iffD1*])
apply (*simp add*: *exp-ge-add-one-self-aux*)
done

lemma *ln-less-self* [*simp*]: $0 < x \Longrightarrow ln\ x < x$
by (*rule order-less-le-trans* [**where** *y*=$ln\ (1 + x)$]) *simp-all*

lemma *ln-ge-zero* [*simp*]:
  assumes *x*: $1 \leq x$ **shows** $0 \leq ln\ x$
proof −
  have $0 < x$ **using** *x* **by** *arith*
  hence $exp\ 0 \leq exp\ (ln\ x)$
    by (*simp add*: *x*)
  thus *?thesis* **by** (*simp only*: *exp-le-cancel-iff*)
qed

lemma *ln-ge-zero-imp-ge-one*:
  assumes *ln*: $0 \leq ln\ x$
    and *x*: $0 < x$
  shows $1 \leq x$
proof −

from *ln* **have** *ln 1 ≤ ln x* **by** *simp*
thus *?thesis* **by** (*simp add*: *x del*: *ln-one*)
qed

lemma *ln-ge-zero-iff* [*simp*]: *0 < x ==> (0 ≤ ln x) = (1 ≤ x)*
by (*blast intro*: *ln-ge-zero ln-ge-zero-imp-ge-one*)

lemma *ln-less-zero-iff* [*simp*]: *0 < x ==> (ln x < 0) = (x < 1)*
by (*insert ln-ge-zero-iff* [*of x*], *arith*)

lemma *ln-gt-zero*:
assumes *x*: *1 < x* **shows** *0 < ln x*
proof −
have *0 < x* **using** *x* **by** *arith*
hence *exp 0 < exp* (*ln x*) **by** (*simp add*: *x*)
thus *?thesis* **by** (*simp only*: *exp-less-cancel-iff*)
qed

lemma *ln-gt-zero-imp-gt-one*:
assumes *ln*: *0 < ln x*
and *x*: *0 < x*
shows *1 < x*
proof −
from *ln* **have** *ln 1 < ln x* **by** *simp*
thus *?thesis* **by** (*simp add*: *x del*: *ln-one*)
qed

lemma *ln-gt-zero-iff* [*simp*]: *0 < x ==> (0 < ln x) = (1 < x)*
by (*blast intro*: *ln-gt-zero ln-gt-zero-imp-gt-one*)

lemma *ln-eq-zero-iff* [*simp*]: *0 < x ==> (ln x = 0) = (x = 1)*
by (*insert ln-less-zero-iff* [*of x*] *ln-gt-zero-iff* [*of x*], *arith*)

lemma *ln-less-zero*: [| *0 < x*; *x < 1* |] *==> ln x < 0*
by *simp*

lemma *exp-ln-eq*: *exp u = x ==> ln x = u*
by *auto*

lemma *isCont-ln*: *0 < x ⟹ isCont ln x*
apply (*subgoal-tac isCont ln* (*exp* (*ln x*)), *simp*)
apply (*rule isCont-inverse-function* [**where** *f=exp*], *simp-all*)
done

lemma *DERIV-ln*: *0 < x ⟹ DERIV ln x :> inverse x*
apply (*rule DERIV-inverse-function* [**where** *f=exp* **and** *a=0* **and** *b=x+1*])
apply (*erule lemma-DERIV-subst* [*OF DERIV-exp exp-ln*])
apply (*simp-all add*: *abs-if isCont-ln*)
done

lemma *ln-series*: **assumes** $0 < x$ **and** $x < 2$
shows $ln\ x = (\sum\ n.\ (-1)\ \hat{}\ n * (1\ /\ real\ (n + 1)) * (x - 1)\ \hat{}\ (Suc\ n))$ (**is** $ln\ x = suminf\ (?f\ (x - 1))$)
proof −
let $?f'\ x\ n = (-1)\ \hat{}\ n * (x - 1)\ \hat{}\ n$

have $ln\ x - suminf\ (?f\ (x - 1)) = ln\ 1 - suminf\ (?f\ (1 - 1))$
proof (*rule DERIV-isconst3*[**where** $x=x$])
fix $x :: real$ **assume** $x \in \{0 <..< 2\}$ **hence** $0 < x$ **and** $x < 2$ **by** *auto*
have $norm\ (1 - x) < 1$ **using** ‹$0 < x$› **and** ‹$x < 2$› **by** *auto*
have $1\ /\ x = 1\ /\ (1 - (1 - x))$ **by** *auto*
also have $\ldots = (\sum\ n.\ (1 - x)\ \hat{}\ n)$ **using** *geometric-sums*[*OF* ‹$norm\ (1 - x) < 1$›] **by** (*rule sums-unique*)
also have $\ldots = suminf\ (?f'\ x)$ **unfolding** *power-mult-distrib*[*symmetric*] **by** (*rule arg-cong*[**where** $f=suminf$], *rule arg-cong*[**where** $f=op\ \hat{}$], *auto*)
finally have *DERIV ln x* :> $suminf\ (?f'\ x)$ **using** *DERIV-ln*[*OF* ‹$0 < x$›] **unfolding** *real-divide-def* **by** *auto*
moreover
have *repos*: $\bigwedge\ h\ x :: real.\ h - 1 + x = h + x - 1$ **by** *auto*
have *DERIV* $(\lambda x.\ suminf\ (?f\ x))\ (x - 1)$:> $(\sum n.\ (-1)\ \hat{}\ n * (1\ /\ real\ (n + 1)) * real\ (Suc\ n) * (x - 1)\ \hat{}\ n)$
proof (*rule DERIV-power-series'*)
show $x - 1 \in \{-1<..<1\}$ **and** $(0 :: real) < 1$ **using** ‹$0 < x$› ‹$x < 2$› **by** *auto*
{ fix $x :: real$ **assume** $x \in \{-1<..<1\}$ **hence** $norm\ (-x) < 1$ **by** *auto*
show *summable* $(\lambda n.\ -1\ \hat{}\ n * (1\ /\ real\ (n + 1)) * real\ (Suc\ n) * x\ \hat{}\ n)$
unfolding *One-nat-def*
by (*auto simp del*: *power-mult-distrib simp add*: *power-mult-distrib*[*symmetric*] *summable-geometric*[*OF* ‹$norm\ (-x) < 1$›])
}
qed
hence *DERIV* $(\lambda x.\ suminf\ (?f\ x))\ (x - 1)$:> $suminf\ (?f'\ x)$ **unfolding** *One-nat-def* **by** *auto*
hence *DERIV* $(\lambda x.\ suminf\ (?f\ (x - 1)))\ x$:> $suminf\ (?f'\ x)$ **unfolding** *DERIV-iff repos* **.**
ultimately have *DERIV* $(\lambda x.\ ln\ x - suminf\ (?f\ (x - 1)))\ x$:> $(suminf\ (?f'\ x) - suminf\ (?f'\ x))$
by (*rule DERIV-diff*)
thus *DERIV* $(\lambda x.\ ln\ x - suminf\ (?f\ (x - 1)))\ x$:> 0 **by** *auto*
qed (*auto simp add*: *assms*)
thus *?thesis* **by** (*auto simp add*: *suminf-zero*)
qed

63.8 Sine and Cosine

definition
$sin :: real => real$ **where**
$sin\ x = (\sum n.\ (if\ even(n)\ then\ 0\ else$

$(-1 \hat{} ((n - Suc\ 0)\ div\ 2))/(real\ (fact\ n))) * x \hat{} n)$

definition
cos :: *real* => *real* **where**
cos x = $(\sum n.$ (*if even*(*n*) *then* $(-1 \hat{} (n\ div\ 2))/(real\ (fact\ n))$
else 0) * *x* ^ *n*)

lemma *summable-sin*:
summable (%*n*.
(*if even n then 0*
else $-1 \hat{} ((n - Suc\ 0)\ div\ 2)/(real\ (fact\ n)))$ *
x ^ *n*)
apply (*rule-tac g* = (%*n*. *inverse* (*real* (*fact n*)) * |*x*| ^ *n*) **in** *summable-comparison-test*)
apply (*rule-tac* [*2*] *summable-exp*)
apply (*rule-tac x* = *0* **in** *exI*)
apply (*auto simp add*: *divide-inverse abs-mult power-abs* [*symmetric*] *zero-le-mult-iff*)
done

lemma *summable-cos*:
summable (%*n*.
(*if even n then*
$-1 \hat{} (n\ div\ 2)/(real\ (fact\ n))$ *else 0*) * *x* ^ *n*)
apply (*rule-tac g* = (%*n*. *inverse* (*real* (*fact n*)) * |*x*| ^ *n*) **in** *summable-comparison-test*)
apply (*rule-tac* [*2*] *summable-exp*)
apply (*rule-tac x* = *0* **in** *exI*)
apply (*auto simp add*: *divide-inverse abs-mult power-abs* [*symmetric*] *zero-le-mult-iff*)
done

lemma *lemma-STAR-sin*:
(*if even n then 0*
else $-1 \hat{} ((n - Suc\ 0)\ div\ 2)/(real\ (fact\ n))) * 0 \hat{} n = 0$
by (*induct n*, *auto*)

lemma *lemma-STAR-cos*:
$0 < n$ ==>
$-1 \hat{} (n\ div\ 2)/(real\ (fact\ n)) * 0 \hat{} n = 0$
by (*induct n*, *auto*)

lemma *lemma-STAR-cos1*:
$0 < n$ ==>
$(-1) \hat{} (n\ div\ 2)/(real\ (fact\ n)) * 0 \hat{} n = 0$
by (*induct n*, *auto*)

lemma *lemma-STAR-cos2*:
$(\sum n=1..<n.$ *if even n then* $-1 \hat{} (n\ div\ 2)/(real\ (fact\ n)) *\ 0 \hat{} n$
else 0) = *0*
apply (*induct n*)
apply (*case-tac* [*2*] *n*, *auto*)
done

lemma *sin-converges*:
(%*n*. (*if even n then 0*
else −*1* ^ ((*n* − *Suc 0*) *div 2*)/(*real* (*fact n*))) ∗
x ^ *n*) *sums sin*(*x*)
unfolding *sin-def* **by** (*rule summable-sin* [*THEN summable-sums*])

lemma *cos-converges*:
(%*n*. (*if even n then*
−*1* ^ (*n div 2*)/(*real* (*fact n*))
else 0) ∗ *x* ^ *n*) *sums cos*(*x*)
unfolding *cos-def* **by** (*rule summable-cos* [*THEN summable-sums*])

lemma *sin-fdiffs*:
diffs(%*n*. *if even n then 0*
else −*1* ^ ((*n* − *Suc 0*) *div 2*)/(*real* (*fact n*)))
= (%*n*. *if even n then*
−*1* ^ (*n div 2*)/(*real* (*fact n*))
else 0)
by (*auto intro*!: *ext*
simp add: *diffs-def divide-inverse real-of-nat-def of-nat-mult*
simp del: *mult-Suc of-nat-Suc*)

lemma *sin-fdiffs2*:
diffs(%*n*. *if even n then 0*
else −*1* ^ ((*n* − *Suc 0*) *div 2*)/(*real* (*fact n*))) *n*
= (*if even n then*
−*1* ^ (*n div 2*)/(*real* (*fact n*))
else 0)
by (*simp only*: *sin-fdiffs*)

lemma *cos-fdiffs*:
diffs(%*n*. *if even n then*
−*1* ^ (*n div 2*)/(*real* (*fact n*)) *else 0*)
= (%*n*. − (*if even n then 0*
else −*1* ^ ((*n* − *Suc 0*)*div 2*)/(*real* (*fact n*))))
by (*auto intro*!: *ext*
simp add: *diffs-def divide-inverse odd-Suc-mult-two-ex real-of-nat-def of-nat-mult*
simp del: *mult-Suc of-nat-Suc*)

lemma *cos-fdiffs2*:
diffs(%*n*. *if even n then*
−*1* ^ (*n div 2*)/(*real* (*fact n*)) *else 0*) *n*
= − (*if even n then 0*
else −*1* ^ ((*n* − *Suc 0*)*div 2*)/(*real* (*fact n*)))
by (*simp only*: *cos-fdiffs*)

Now at last we can get the derivatives of exp, sin and cos

lemma *lemma-sin-minus*:
$- \sin x = (\sum n. - ((\text{if even } n \text{ then } 0$
$\text{else } -1 \hat{} ((n - \text{Suc } 0) \text{ div } 2)/(\text{real } (\text{fact } n))) * x \hat{} n))$
by (*auto intro*!: *sums-unique sums-minus sin-converges*)

lemma *lemma-sin-ext*:
$\sin = (\%x. \sum n.$
$(\text{if even } n \text{ then } 0$
$\text{else } -1 \hat{} ((n - \text{Suc } 0) \text{ div } 2)/(\text{real } (\text{fact } n))) *$
$x \hat{} n)$
by (*auto intro*!: *ext simp add*: *sin-def*)

lemma *lemma-cos-ext*:
$\cos = (\%x. \sum n.$
$(\text{if even } n \text{ then } -1 \hat{} (n \text{ div } 2)/(\text{real } (\text{fact } n)) \text{ else } 0) *$
$x \hat{} n)$
by (*auto intro*!: *ext simp add*: *cos-def*)

lemma *DERIV-sin* [*simp*]: *DERIV sin x :> cos*(x)
apply (*simp add*: *cos-def*)
apply (*subst lemma-sin-ext*)
apply (*auto simp add*: *sin-fdiffs2* [*symmetric*])
apply (*rule-tac K = 1 + |x|* **in** *termdiffs*)
apply (*auto intro*: *sin-converges cos-converges sums-summable intro*!: *sums-minus* [*THEN sums-summable*] *simp add*: *cos-fdiffs sin-fdiffs*)
done

lemma *DERIV-cos* [*simp*]: *DERIV cos x :> −sin*(x)
apply (*subst lemma-cos-ext*)
apply (*auto simp add*: *lemma-sin-minus cos-fdiffs2* [*symmetric*] *minus-mult-left*)
apply (*rule-tac K = 1 + |x|* **in** *termdiffs*)
apply (*auto intro*: *sin-converges cos-converges sums-summable intro*!: *sums-minus* [*THEN sums-summable*] *simp add*: *cos-fdiffs sin-fdiffs diffs-minus*)
done

lemma *isCont-sin* [*simp*]: *isCont sin x*
by (*rule DERIV-sin* [*THEN DERIV-isCont*])

lemma *isCont-cos* [*simp*]: *isCont cos x*
by (*rule DERIV-cos* [*THEN DERIV-isCont*])

63.9 Properties of Sine and Cosine

lemma *sin-zero* [*simp*]: *sin 0 = 0*
unfolding *sin-def* **by** (*simp add*: *powser-zero*)

lemma *cos-zero* [*simp*]: *cos 0 = 1*
unfolding *cos-def* **by** (*simp add*: *powser-zero*)

lemma *DERIV-sin-sin-mult* [*simp*]:
DERIV (%*x*. *sin*(*x*)∗*sin*(*x*)) *x* :> *cos*(*x*) ∗ *sin*(*x*) + *cos*(*x*) ∗ *sin*(*x*)
by (*rule DERIV-mult*, *auto*)

lemma *DERIV-sin-sin-mult2* [*simp*]:
DERIV (%*x*. *sin*(*x*)∗*sin*(*x*)) *x* :> *2* ∗ *cos*(*x*) ∗ *sin*(*x*)
apply (*cut-tac x* = *x* **in** *DERIV-sin-sin-mult*)
apply (*auto simp add*: *mult-assoc*)
done

lemma *DERIV-sin-realpow2* [*simp*]:
DERIV (%*x*. (*sin x*)^2) *x* :> *cos*(*x*) ∗ *sin*(*x*) + *cos*(*x*) ∗ *sin*(*x*)
by (*auto simp add*: *numeral-2-eq-2 real-mult-assoc* [*symmetric*])

lemma *DERIV-sin-realpow2a* [*simp*]:
DERIV (%*x*. (*sin x*)^2) *x* :> *2* ∗ *cos*(*x*) ∗ *sin*(*x*)
by (*auto simp add*: *numeral-2-eq-2*)

lemma *DERIV-cos-cos-mult* [*simp*]:
DERIV (%*x*. *cos*(*x*)∗*cos*(*x*)) *x* :> −*sin*(*x*) ∗ *cos*(*x*) + −*sin*(*x*) ∗ *cos*(*x*)
by (*rule DERIV-mult*, *auto*)

lemma *DERIV-cos-cos-mult2* [*simp*]:
DERIV (%*x*. *cos*(*x*)∗*cos*(*x*)) *x* :> −*2* ∗ *cos*(*x*) ∗ *sin*(*x*)
apply (*cut-tac x* = *x* **in** *DERIV-cos-cos-mult*)
apply (*auto simp add*: *mult-ac*)
done

lemma *DERIV-cos-realpow2* [*simp*]:
DERIV (%*x*. (*cos x*)^2) *x* :> −*sin*(*x*) ∗ *cos*(*x*) + −*sin*(*x*) ∗ *cos*(*x*)
by (*auto simp add*: *numeral-2-eq-2 real-mult-assoc* [*symmetric*])

lemma *DERIV-cos-realpow2a* [*simp*]:
DERIV (%*x*. (*cos x*)^2) *x* :> −*2* ∗ *cos*(*x*) ∗ *sin*(*x*)
by (*auto simp add*: *numeral-2-eq-2*)

lemma *lemma-DERIV-subst*: [| *DERIV f x* :> *D*; *D* = *E* |] ==> *DERIV f x* :> *E*
by *auto*

lemma *DERIV-cos-realpow2b*: *DERIV* (%*x*. (*cos x*)^2) *x* :> −(*2* ∗ *cos*(*x*) ∗ *sin*(*x*))
apply (*rule lemma-DERIV-subst*)
apply (*rule DERIV-cos-realpow2a*, *auto*)
done

lemma *DERIV-cos-cos-mult3* [*simp*]:
DERIV (%*x*. *cos*(*x*)∗*cos*(*x*)) *x* :> −(*2* ∗ *cos*(*x*) ∗ *sin*(*x*))
apply (*rule lemma-DERIV-subst*)

apply (*rule DERIV-cos-cos-mult2*, *auto*)
done

lemma *DERIV-sin-circle-all*:
  $\forall x.\ DERIV\ (\%x.\ (sin\ x)^2 + (cos\ x)^2)\ x :>$
    $(2*cos(x)*sin(x) - 2*cos(x)*sin(x))$
apply (*simp only*: *diff-minus*, *safe*)
apply (*rule DERIV-add*)
apply (*auto simp add*: *numeral-2-eq-2*)
done

lemma *DERIV-sin-circle-all-zero* [*simp*]:
  $\forall x.\ DERIV\ (\%x.\ (sin\ x)^2 + (cos\ x)^2)\ x :> 0$
by (*cut-tac DERIV-sin-circle-all*, *auto*)

lemma *sin-cos-squared-add* [*simp*]: $((sin\ x)^2) + ((cos\ x)^2) = 1$
apply (*cut-tac* $x = x$ **and** $y = 0$ **in** *DERIV-sin-circle-all-zero* [*THEN DERIV-isconst-all*])
apply (*auto simp add*: *numeral-2-eq-2*)
done

lemma *sin-cos-squared-add2* [*simp*]: $((cos\ x)^2) + ((sin\ x)^2) = 1$
apply (*subst add-commute*)
apply (*rule sin-cos-squared-add*)
done

lemma *sin-cos-squared-add3* [*simp*]: $cos\ x * cos\ x + sin\ x * sin\ x = 1$
apply (*cut-tac* $x = x$ **in** *sin-cos-squared-add2*)
apply (*simp add*: *power2-eq-square*)
done

lemma *sin-squared-eq*: $(sin\ x)^2 = 1 - (cos\ x)^2$
apply (*rule-tac a1* $= (cos\ x)^2$ **in** *add-right-cancel* [*THEN iffD1*])
apply *simp*
done

lemma *cos-squared-eq*: $(cos\ x)^2 = 1 - (sin\ x)^2$
apply (*rule-tac a1* $= (sin\ x)^2$ **in** *add-right-cancel* [*THEN iffD1*])
apply *simp*
done

lemma *abs-sin-le-one* [*simp*]: $|sin\ x| \leq 1$
by (*rule power2-le-imp-le*, *simp-all add*: *sin-squared-eq*)

lemma *sin-ge-minus-one* [*simp*]: $-1 \leq sin\ x$
apply (*insert abs-sin-le-one* [*of x*])
apply (*simp add*: *abs-le-iff del*: *abs-sin-le-one*)
done

lemma *sin-le-one* [*simp*]: $sin\ x \leq 1$

apply (*insert abs-sin-le-one* [*of x*])
apply (*simp add*: *abs-le-iff del*: *abs-sin-le-one*)
done

lemma *abs-cos-le-one* [*simp*]: $|cos\ x| \leq 1$
by (*rule power2-le-imp-le*, *simp-all add*: *cos-squared-eq*)

lemma *cos-ge-minus-one* [*simp*]: $-1 \leq cos\ x$
apply (*insert abs-cos-le-one* [*of x*])
apply (*simp add*: *abs-le-iff del*: *abs-cos-le-one*)
done

lemma *cos-le-one* [*simp*]: $cos\ x \leq 1$
apply (*insert abs-cos-le-one* [*of x*])
apply (*simp add*: *abs-le-iff del*: *abs-cos-le-one*)
done

lemma *DERIV-fun-pow*: $DERIV\ g\ x :> m ==>$
$DERIV\ (\%x.\ (g\ x)\ \hat{}\ n)\ x :> real\ n * (g\ x)\ \hat{}\ (n - 1) * m$
unfolding *One-nat-def*
apply (*rule lemma-DERIV-subst*)
apply (*rule-tac f* = $(\%x.\ x\ \hat{}\ n)$ **in** *DERIV-chain2*)
apply (*rule DERIV-pow*, *auto*)
done

lemma *DERIV-fun-exp*:
$DERIV\ g\ x :> m ==> DERIV\ (\%x.\ exp(g\ x))\ x :> exp(g\ x) * m$
apply (*rule lemma-DERIV-subst*)
apply (*rule-tac f* = *exp* **in** *DERIV-chain2*)
apply (*rule DERIV-exp*, *auto*)
done

lemma *DERIV-fun-sin*:
$DERIV\ g\ x :> m ==> DERIV\ (\%x.\ sin(g\ x))\ x :> cos(g\ x) * m$
apply (*rule lemma-DERIV-subst*)
apply (*rule-tac f* = *sin* **in** *DERIV-chain2*)
apply (*rule DERIV-sin*, *auto*)
done

lemma *DERIV-fun-cos*:
$DERIV\ g\ x :> m ==> DERIV\ (\%x.\ cos(g\ x))\ x :> -sin(g\ x) * m$
apply (*rule lemma-DERIV-subst*)
apply (*rule-tac f* = *cos* **in** *DERIV-chain2*)
apply (*rule DERIV-cos*, *auto*)
done

lemmas *DERIV-intros* = *DERIV-ident DERIV-const DERIV-cos DERIV-cmult*
DERIV-sin DERIV-exp DERIV-inverse DERIV-pow
DERIV-add DERIV-diff DERIV-mult DERIV-minus

DERIV-inverse-fun DERIV-quotient DERIV-fun-pow
DERIV-fun-exp DERIV-fun-sin DERIV-fun-cos

lemma *lemma-DERIV-sin-cos-add*:
  $\forall x.$
    DERIV (%*x*. (*sin* (*x* + *y*) − (*sin x* ∗ *cos y* + *cos x* ∗ *sin y*)) ^ *2* +
      (*cos* (*x* + *y*) − (*cos x* ∗ *cos y* − *sin x* ∗ *sin y*)) ^ *2*) *x* :> *0*
apply (*safe*, *rule lemma-DERIV-subst*)
apply (*best intro*!: *DERIV-intros intro*: *DERIV-chain2*)
 — replaces the old *DERIV-tac*
apply (*auto simp add*: *algebra-simps*)
done

lemma *sin-cos-add* [*simp*]:
  (*sin* (*x* + *y*) − (*sin x* ∗ *cos y* + *cos x* ∗ *sin y*)) ^ *2* +
  (*cos* (*x* + *y*) − (*cos x* ∗ *cos y* − *sin x* ∗ *sin y*)) ^ *2* = *0*
apply (*cut-tac y* = *0* **and** *x* = *x* **and** *y7* = *y*
    in *lemma-DERIV-sin-cos-add* [*THEN DERIV-isconst-all*])
apply (*auto simp add*: *numeral-2-eq-2*)
done

lemma *sin-add*: *sin* (*x* + *y*) = *sin x* ∗ *cos y* + *cos x* ∗ *sin y*
apply (*cut-tac x* = *x* **and** *y* = *y* **in** *sin-cos-add*)
apply (*simp del*: *sin-cos-add*)
done

lemma *cos-add*: *cos* (*x* + *y*) = *cos x* ∗ *cos y* − *sin x* ∗ *sin y*
apply (*cut-tac x* = *x* **and** *y* = *y* **in** *sin-cos-add*)
apply (*simp del*: *sin-cos-add*)
done

lemma *lemma-DERIV-sin-cos-minus*:
  $\forall x.$ *DERIV* (%*x*. (*sin*(−*x*) + (*sin x*)) ^ *2* + (*cos*(−*x*) − (*cos x*)) ^ *2*) *x* :> *0*
apply (*safe*, *rule lemma-DERIV-subst*)
apply (*best intro*!: *DERIV-intros intro*: *DERIV-chain2*)
apply (*simp add*: *algebra-simps*)
done

lemma *sin-cos-minus*:
  (*sin*(−*x*) + (*sin x*)) ^ *2* + (*cos*(−*x*) − (*cos x*)) ^ *2* = *0*
apply (*cut-tac y* = *0* **and** *x* = *x*
    in *lemma-DERIV-sin-cos-minus* [*THEN DERIV-isconst-all*])
apply *simp*
done

lemma *sin-minus* [*simp*]: *sin* (−*x*) = −*sin*(*x*)
 using *sin-cos-minus* [**where** *x*=*x*] **by** *simp*

lemma *cos-minus* [*simp*]: *cos* $(-x) = cos(x)$
using *sin-cos-minus* [**where** $x{=}x$] **by** *simp*

lemma *sin-diff*: *sin* $(x - y) =$ *sin* $x *$ *cos* $y -$ *cos* $x *$ *sin* y
by (*simp add*: *diff-minus sin-add*)

lemma *sin-diff2*: *sin* $(x - y) =$ *cos* $y *$ *sin* $x -$ *sin* $y *$ *cos* x
by (*simp add*: *sin-diff mult-commute*)

lemma *cos-diff*: *cos* $(x - y) =$ *cos* $x *$ *cos* $y +$ *sin* $x *$ *sin* y
by (*simp add*: *diff-minus cos-add*)

lemma *cos-diff2*: *cos* $(x - y) =$ *cos* $y *$ *cos* $x +$ *sin* $y *$ *sin* x
by (*simp add*: *cos-diff mult-commute*)

lemma *sin-double* [*simp*]: $sin(2 * x) = 2{*}$ *sin* $x *$ *cos* x
using *sin-add* [**where** $x{=}x$ **and** $y{=}x$] **by** *simp*

lemma *cos-double*: $cos(2{*}\ x) = ((cos\ x)^2) - ((sin\ x)^2)$
using *cos-add* [**where** $x{=}x$ **and** $y{=}x$]
by (*simp add*: *power2-eq-square*)

63.10 The Constant Pi

definition
pi :: *real* **where**
$pi = 2 * (THE\ x.\ 0 \leq (x{::}real)\ \&\ x \leq 2\ \&\ cos\ x = 0)$

Show that there's a least positive x with $cos\ x = 0$; hence define pi.

lemma *sin-paired*:
$(\%n.\ -1\ \hat{}\ n\ /(real\ (fact\ (2 * n + 1))) * x\ \hat{}\ (2 * n + 1))$
sums *sin* x
proof −
have $(\lambda n.\ \sum k = n * 2..{<}n * 2 + 2.$
$(if\ even\ k\ then\ 0$
$else\ -1\ \hat{}\ ((k - Suc\ 0)\ div\ 2)\ /\ real\ (fact\ k)) *$
$x\ \hat{}\ k)$
sums sin x
unfolding *sin-def*
by (*rule sin-converges* [*THEN sums-summable*, *THEN sums-group*], *simp*)
thus *?thesis* **unfolding** *One-nat-def* **by** (*simp add*: *mult-ac*)
qed

FIXME: This is a long, ugly proof!

lemma *sin-gt-zero*: $[|0 < x;\ x < 2\ |] ==> 0 < sin\ x$
apply (*subgoal-tac*
$(\lambda n.\ \sum k = n * 2..{<}n * 2 + 2.$
$-1\ \hat{}\ k\ /\ real\ (fact\ (2 * k + 1)) * x\ \hat{}\ (2 * k + 1))$
sums $(\sum n.\ -1\ \hat{}\ n\ /\ real\ (fact\ (2 * n + 1)) * x\ \hat{}\ (2 * n + 1)))$

prefer *2*
apply (*rule sin-paired* [*THEN sums-summable*, *THEN sums-group*], *simp*)
apply (*rotate-tac 2*)
apply (*drule sin-paired* [*THEN sums-unique*, *THEN ssubst*])
unfolding *One-nat-def*
apply (*auto simp del*: *fact-Suc*)
apply (*frule sums-unique*)
apply (*auto simp del*: *fact-Suc*)
apply (*rule-tac n1 = 0* **in** *series-pos-less* [*THEN* [*2*] *order-le-less-trans*])
apply (*auto simp del*: *fact-Suc*)
apply (*erule sums-summable*)
apply (*case-tac m=0*)
apply (*simp* (*no-asm-simp*))
apply (*subgoal-tac 6 * (x * (x * x) / real* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*)))))))
*< 6 * x*)
apply (*simp only*: *mult-less-cancel-left*, *simp*)
apply (*simp* (*no-asm-simp*) *add*: *numeral-2-eq-2* [*symmetric*] *mult-assoc* [*symmetric*])
apply (*subgoal-tac x*x < 2*3*, *simp*)
apply (*rule mult-strict-mono*)
apply (*auto simp add*: *real-0-less-add-iff real-of-nat-Suc simp del*: *fact-Suc*)
apply (*subst fact-Suc*)
apply (*subst fact-Suc*)
apply (*subst fact-Suc*)
apply (*subst fact-Suc*)
apply (*subst real-of-nat-mult*)
apply (*subst real-of-nat-mult*)
apply (*subst real-of-nat-mult*)
apply (*subst real-of-nat-mult*)
apply (*simp* (*no-asm*) *add*: *divide-inverse del*: *fact-Suc*)
apply (*auto simp add*: *mult-assoc* [*symmetric*] *simp del*: *fact-Suc*)
apply (*rule-tac c=real* (*Suc* (*Suc* (*4*m*))) **in** *mult-less-imp-less-right*)
apply (*auto simp add*: *mult-assoc simp del*: *fact-Suc*)
apply (*rule-tac c=real* (*Suc* (*Suc* (*Suc* (*4*m*)))) **in** *mult-less-imp-less-right*)
apply (*auto simp add*: *mult-assoc mult-less-cancel-left simp del*: *fact-Suc*)
apply (*subgoal-tac x * (x * x ^ (4*m)) = (x ^ (4*m)) * (x * x)*)
apply (*erule ssubst*)+
apply (*auto simp del*: *fact-Suc*)
apply (*subgoal-tac 0 < x ^ (4 * m)*)
prefer *2* **apply** (*simp only*: *zero-less-power*)
apply (*simp* (*no-asm-simp*) *add*: *mult-less-cancel-left*)
apply (*rule mult-strict-mono*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *sin-gt-zero1*: [|*0 < x*; *x < 2* |] ==> *0 < sin x*
by (*auto intro*: *sin-gt-zero*)

lemma *cos-double-less-one*: [| *0 < x*; *x < 2* |] ==> *cos* (*2 * x*) *< 1*
apply (*cut-tac x = x* **in** *sin-gt-zero1*)

apply (*auto simp add*: *cos-squared-eq cos-double*)
done

lemma *cos-paired*:
(%*n*. −*1* ^ *n* /(*real* (*fact* (*2* ∗ *n*))) ∗ *x* ^ (*2* ∗ *n*)) *sums cos x*
proof −
have (λ*n*. ∑ *k* = *n* ∗ *2*..<*n* ∗ *2* + *2*.
(*if even k then* −*1* ^ (*k div 2*) / *real* (*fact k*) *else 0*) ∗
x ^ *k*)
sums cos x
unfolding *cos-def*
by (*rule cos-converges* [*THEN sums-summable*, *THEN sums-group*], *simp*)
thus *?thesis* **by** (*simp add*: *mult-ac*)
qed

lemma *fact-lemma*: *real* (*n*::*nat*) ∗ *4* = *real* (*4* ∗ *n*)
by *simp*

lemma *cos-two-less-zero* [*simp*]: *cos* (*2*) < *0*
apply (*cut-tac x* = *2* **in** *cos-paired*)
apply (*drule sums-minus*)
apply (*rule neg-less-iff-less* [*THEN iffD1*])
apply (*frule sums-unique*, *auto*)
apply (*rule-tac y* =
∑ *n*=*0*..< *Suc*(*Suc*(*Suc 0*)). − (−*1* ^ *n* / (*real*(*fact* (*2*∗*n*))) ∗ *2* ^ (*2*∗*n*))
in *order-less-trans*)
apply (*simp* (*no-asm*) *add*: *fact-num-eq-if realpow-num-eq-if del*: *fact-Suc*)
apply (*simp* (*no-asm*) *add*: *mult-assoc del*: *setsum-op-ivl-Suc*)
apply (*rule sumr-pos-lt-pair*)
apply (*erule sums-summable*, *safe*)
unfolding *One-nat-def*
apply (*simp* (*no-asm*) *add*: *divide-inverse real-0-less-add-iff mult-assoc* [*symmetric*]

del: *fact-Suc*)
apply (*rule real-mult-inverse-cancel2*)
apply (*rule real-of-nat-fact-gt-zero*)+
apply (*simp* (*no-asm*) *add*: *mult-assoc* [*symmetric*] *del*: *fact-Suc*)
apply (*subst fact-lemma*)
apply (*subst fact-Suc* [*of Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*4* ∗ *d*)))))))])
apply (*simp only*: *real-of-nat-mult*)
apply (*rule mult-strict-mono*, *force*)
apply (*rule-tac* [*3*] *real-of-nat-ge-zero*)
prefer *2* **apply** *force*
apply (*rule real-of-nat-less-iff* [*THEN iffD2*])
apply (*rule fact-less-mono*, *auto*)
done

lemmas *cos-two-neq-zero* [*simp*] = *cos-two-less-zero* [*THEN less-imp-neq*]
lemmas *cos-two-le-zero* [*simp*] = *cos-two-less-zero* [*THEN order-less-imp-le*]

lemma *cos-is-zero*: $EX!\ x.\ 0 \leq x\ \&\ x \leq 2\ \&\ cos\ x = 0$
apply (*subgoal-tac* $\exists x.\ 0 \leq x\ \&\ x \leq 2\ \&\ cos\ x = 0$)
apply (*rule-tac* [2] *IVT2*)
apply (*auto intro*: *DERIV-isCont DERIV-cos*)
apply (*cut-tac* $x = xa$ **and** $y = y$ **in** *linorder-less-linear*)
apply (*rule ccontr*)
apply (*subgoal-tac* $(\forall x.\ cos\ differentiable\ x)\ \&\ (\forall x.\ isCont\ cos\ x)$)
apply (*auto intro*: *DERIV-cos DERIV-isCont simp add*: *differentiable-def*)
apply (*drule-tac* $f = cos$ **in** *Rolle*)
apply (*drule-tac* [5] $f = cos$ **in** *Rolle*)
apply (*auto dest*!: *DERIV-cos* [*THEN DERIV-unique*] *simp add*: *differentiable-def*)
apply (*drule-tac* $y1 = xa$ **in** *order-le-less-trans* [*THEN sin-gt-zero*])
apply (*assumption*, *rule-tac* $y{=}y$ **in** *order-less-le-trans*, *simp-all*)
apply (*drule-tac* $y1 = y$ **in** *order-le-less-trans* [*THEN sin-gt-zero*], *assumption*, *simp-all*)
done

lemma *pi-half*: $pi/2 = (THE\ x.\ 0 \leq x\ \&\ x \leq 2\ \&\ cos\ x = 0)$
by (*simp add*: *pi-def*)

lemma *cos-pi-half* [*simp*]: $cos\ (pi\ /\ 2) = 0$
by (*simp add*: *pi-half cos-is-zero* [*THEN theI'*])

lemma *pi-half-gt-zero* [*simp*]: $0 < pi\ /\ 2$
apply (*rule order-le-neq-trans*)
apply (*simp add*: *pi-half cos-is-zero* [*THEN theI'*])
apply (*rule notI*, *drule arg-cong* [**where** $f{=}cos$], *simp*)
done

lemmas *pi-half-neq-zero* [*simp*] = *pi-half-gt-zero* [*THEN less-imp-neq*, *symmetric*]
lemmas *pi-half-ge-zero* [*simp*] = *pi-half-gt-zero* [*THEN order-less-imp-le*]

lemma *pi-half-less-two* [*simp*]: $pi\ /\ 2 < 2$
apply (*rule order-le-neq-trans*)
apply (*simp add*: *pi-half cos-is-zero* [*THEN theI'*])
apply (*rule notI*, *drule arg-cong* [**where** $f{=}cos$], *simp*)
done

lemmas *pi-half-neq-two* [*simp*] = *pi-half-less-two* [*THEN less-imp-neq*]
lemmas *pi-half-le-two* [*simp*] = *pi-half-less-two* [*THEN order-less-imp-le*]

lemma *pi-gt-zero* [*simp*]: $0 < pi$
by (*insert pi-half-gt-zero*, *simp*)

lemma *pi-ge-zero* [*simp*]: $0 \leq pi$
by (*rule pi-gt-zero* [*THEN order-less-imp-le*])

lemma *pi-neq-zero* [*simp*]: $pi \neq 0$

by (*rule pi-gt-zero* [*THEN less-imp-neq*, *THEN not-sym*])

lemma *pi-not-less-zero* [*simp*]: ¬ *pi* < *0*
by (*simp add*: *linorder-not-less*)

lemma *minus-pi-half-less-zero*: −(*pi*/*2*) < *0*
by *simp*

lemma *m2pi-less-pi*: − (*2* ∗ *pi*) < *pi*
proof −
  have − (*2* ∗ *pi*) < *0* **and** *0* < *pi* **by** *auto*
  from *order-less-trans*[*OF this*] **show** *?thesis* **.**
qed

lemma *sin-pi-half* [*simp*]: *sin*(*pi*/*2*) = *1*
apply (*cut-tac x* = *pi*/*2* **in** *sin-cos-squared-add2*)
apply (*cut-tac sin-gt-zero* [*OF pi-half-gt-zero pi-half-less-two*])
apply (*simp add*: *power2-eq-square*)
done

lemma *cos-pi* [*simp*]: *cos pi* = −*1*
by (*cut-tac x* = *pi*/*2* **and** *y* = *pi*/*2* **in** *cos-add*, *simp*)

lemma *sin-pi* [*simp*]: *sin pi* = *0*
by (*cut-tac x* = *pi*/*2* **and** *y* = *pi*/*2* **in** *sin-add*, *simp*)

lemma *sin-cos-eq*: *sin x* = *cos* (*pi*/*2* − *x*)
by (*simp add*: *diff-minus cos-add*)
declare *sin-cos-eq* [*symmetric*, *simp*]

lemma *minus-sin-cos-eq*: −*sin x* = *cos* (*x* + *pi*/*2*)
by (*simp add*: *cos-add*)
declare *minus-sin-cos-eq* [*symmetric*, *simp*]

lemma *cos-sin-eq*: *cos x* = *sin* (*pi*/*2* − *x*)
by (*simp add*: *diff-minus sin-add*)
declare *cos-sin-eq* [*symmetric*, *simp*]

lemma *sin-periodic-pi* [*simp*]: *sin* (*x* + *pi*) = − *sin x*
by (*simp add*: *sin-add*)

lemma *sin-periodic-pi2* [*simp*]: *sin* (*pi* + *x*) = − *sin x*
by (*simp add*: *sin-add*)

lemma *cos-periodic-pi* [*simp*]: *cos* (*x* + *pi*) = − *cos x*
by (*simp add*: *cos-add*)

lemma *sin-periodic* [*simp*]: *sin* (*x* + *2*∗*pi*) = *sin x*
by (*simp add*: *sin-add cos-double*)

lemma *cos-periodic* [*simp*]: *cos* (*x* + *2*∗*pi*) = *cos x*
by (*simp add*: *cos-add cos-double*)

lemma *cos-npi* [*simp*]: *cos* (*real n* ∗ *pi*) = −*1* ^ *n*
apply (*induct n*)
apply (*auto simp add*: *real-of-nat-Suc left-distrib*)
done

lemma *cos-npi2* [*simp*]: *cos* (*pi* ∗ *real n*) = −*1* ^ *n*
proof −
  have *cos* (*pi* ∗ *real n*) = *cos* (*real n* ∗ *pi*) **by** (*simp only*: *mult-commute*)
  also have ... = −*1* ^ *n* **by** (*rule cos-npi*)
  finally show *?thesis* **.**
qed

lemma *sin-npi* [*simp*]: *sin* (*real* (*n*::*nat*) ∗ *pi*) = *0*
apply (*induct n*)
apply (*auto simp add*: *real-of-nat-Suc left-distrib*)
done

lemma *sin-npi2* [*simp*]: *sin* (*pi* ∗ *real* (*n*::*nat*)) = *0*
by (*simp add*: *mult-commute* [*of pi*])

lemma *cos-two-pi* [*simp*]: *cos* (*2* ∗ *pi*) = *1*
by (*simp add*: *cos-double*)

lemma *sin-two-pi* [*simp*]: *sin* (*2* ∗ *pi*) = *0*
by *simp*

lemma *sin-gt-zero2*: [| *0* < *x*; *x* < *pi*/*2* |] ==> *0* < *sin x*
apply (*rule sin-gt-zero*, *assumption*)
apply (*rule order-less-trans*, *assumption*)
apply (*rule pi-half-less-two*)
done

lemma *sin-less-zero*:
  assumes *lb*: − *pi*/*2* < *x* **and** *x* < *0* **shows** *sin x* < *0*
proof −
  have *0* < *sin* (− *x*) **using** *prems* **by** (*simp only*: *sin-gt-zero2*)
  thus *?thesis* **by** *simp*
qed

lemma *pi-less-4*: *pi* < *4*
by (*cut-tac pi-half-less-two*, *auto*)

lemma *cos-gt-zero*: [| *0* < *x*; *x* < *pi*/*2* |] ==> *0* < *cos x*
apply (*cut-tac pi-less-4*)
apply (*cut-tac f* = *cos* **and** *a* = *0* **and** *b* = *x* **and** *y* = *0* **in** *IVT2-objl*, *safe*,

simp-all)
apply (*cut-tac cos-is-zero*, *safe*)
apply (*rename-tac y z*)
apply (*drule-tac x = y* **in** *spec*)
apply (*drule-tac x = pi/2* **in** *spec*, *simp*)
done

lemma *cos-gt-zero-pi*: [| *−(pi/2) < x*; *x < pi/2* |] ==> *0 < cos x*
apply (*rule-tac x = x* **and** *y = 0* **in** *linorder-cases*)
apply (*rule cos-minus* [*THEN subst*])
apply (*rule cos-gt-zero*)
apply (*auto intro*: *cos-gt-zero*)
done

lemma *cos-ge-zero*: [| *−(pi/2) ≤ x*; *x ≤ pi/2* |] ==> *0 ≤ cos x*
apply (*auto simp add*: *order-le-less cos-gt-zero-pi*)
apply (*subgoal-tac x = pi/2*, *auto*)
done

lemma *sin-gt-zero-pi*: [| *0 < x*; *x < pi* |] ==> *0 < sin x*
apply (*subst sin-cos-eq*)
apply (*rotate-tac 1*)
apply (*drule real-sum-of-halves* [*THEN ssubst*])
apply (*auto intro*!: *cos-gt-zero-pi simp del*: *sin-cos-eq* [*symmetric*])
done

lemma *pi-ge-two*: *2 ≤ pi*
proof (*rule ccontr*)
assume *¬ 2 ≤ pi* **hence** *pi < 2* **by** *auto*
have *∃ y > pi. y < 2 ∧ y < 2 * pi*
proof (*cases 2 < 2 * pi*)
case *True* **with** *dense*[*OF ‹pi < 2›*] **show** *?thesis* **by** *auto*
next
case *False* **have** *pi < 2 * pi* **by** *auto*
from *dense*[*OF this*] **and** *False* **show** *?thesis* **by** *auto*
qed
then obtain *y* **where** *pi < y* **and** *y < 2* **and** *y < 2 * pi* **by** *blast*
hence *0 < sin y* **using** *sin-gt-zero* **by** *auto*
moreover
have *sin y < 0* **using** *sin-gt-zero-pi*[*of y − pi*] *‹pi < y›* **and** *‹y < 2 * pi›*
sin-periodic-pi[*of y − pi*] **by** *auto*
ultimately show *False* **by** *auto*
qed

lemma *sin-ge-zero*: [| *0 ≤ x*; *x ≤ pi* |] ==> *0 ≤ sin x*
by (*auto simp add*: *order-le-less sin-gt-zero-pi*)

lemma *cos-total*: [| *−1 ≤ y*; *y ≤ 1* |] ==> *EX*! *x. 0 ≤ x & x ≤ pi & (cos x =*

y)
apply (*subgoal-tac* $\exists x.\ 0 \leq x\ \&\ x \leq pi\ \&\ cos\ x = y$)
apply (*rule-tac* [*2*] *IVT2*)
apply (*auto intro*: *order-less-imp-le DERIV-isCont DERIV-cos*)
apply (*cut-tac* $x = xa$ **and** $y = y$ **in** *linorder-less-linear*)
apply (*rule ccontr*, *auto*)
apply (*drule-tac* $f = cos$ **in** *Rolle*)
apply (*drule-tac* [*5*] $f = cos$ **in** *Rolle*)
apply (*auto intro*: *order-less-imp-le DERIV-isCont DERIV-cos*
dest!: *DERIV-cos* [*THEN DERIV-unique*]
simp add: *differentiable-def*)
apply (*auto dest*: *sin-gt-zero-pi* [*OF order-le-less-trans order-less-le-trans*])
done

lemma *sin-total*:
$[|\ -1 \leq y;\ y \leq 1\ |] ==> EX!\ x.\ -(pi/2) \leq x\ \&\ x \leq pi/2\ \&\ (sin\ x = y)$
apply (*rule ccontr*)
apply (*subgoal-tac* $\forall x.\ (-\ (pi/2) \leq x\ \&\ x \leq pi/2\ \&\ (sin\ x = y)) = (0 \leq (x +$
$pi/2)\ \&\ (x + pi/2) \leq pi\ \&\ (cos\ (x + pi/2) = -y))$)
apply (*erule contrapos-np*)
apply (*simp del*: *minus-sin-cos-eq* [*symmetric*])
apply (*cut-tac* $y{=}{-}y$ **in** *cos-total*, *simp*) **apply** *simp*
apply (*erule ex1E*)
apply (*rule-tac* $a = x - (pi/2)$ **in** *ex1I*)
apply (*simp* (*no-asm*) *add*: *add-assoc*)
apply (*rotate-tac 3*)
apply (*drule-tac* $x = xa + pi/2$ **in** *spec*, *safe*, *simp-all*)
done

lemma *reals-Archimedean4*:
$[|\ 0 < y;\ 0 \leq x\ |] ==> \exists n.\ real\ n * y \leq x\ \&\ x < real\ (Suc\ n) * y$
apply (*auto dest*!: *reals-Archimedean3*)
apply (*drule-tac* $x = x$ **in** *spec*, *clarify*)
apply (*subgoal-tac* $x < real(LEAST\ m{::}nat.\ x < real\ m * y) * y$)
prefer *2* **apply** (*erule LeastI*)
apply (*case-tac LEAST* $m{::}nat.\ x < real\ m * y$, *simp*)
apply (*subgoal-tac* $\sim x < real\ nat * y$)
prefer *2* **apply** (*rule not-less-Least*, *simp*, *force*)
done

lemma *cos-zero-lemma*:
$[|\ 0 \leq x;\ cos\ x = 0\ |] ==>$
$\exists n{::}nat.\ \sim even\ n\ \&\ x = real\ n * (pi/2)$
apply (*drule pi-gt-zero* [*THEN reals-Archimedean4*], *safe*)
apply (*subgoal-tac* $0 \leq x - real\ n * pi\ \&$
$(x - real\ n * pi) \leq pi\ \&\ (cos\ (x - real\ n * pi) = 0)$)
apply (*auto simp add*: *algebra-simps real-of-nat-Suc*)
prefer *2* **apply** (*simp add*: *cos-diff*)

apply (*simp add*: *cos-diff*)
apply (*subgoal-tac EX*! *x*. $0 \leq x$ & $x \leq pi$ & $cos\ x = 0$)
apply (*rule-tac* [2] *cos-total*, *safe*)
apply (*drule-tac* $x = x - real\ n * pi$ **in** *spec*)
apply (*drule-tac* $x = pi/2$ **in** *spec*)
apply (*simp add*: *cos-diff*)
apply (*rule-tac* $x = Suc\ (2 * n)$ **in** *exI*)
apply (*simp add*: *real-of-nat-Suc algebra-simps*, *auto*)
done

lemma *sin-zero-lemma*:
$[|\ 0 \leq x;\ sin\ x = 0\ |] ==>$
$\exists n::nat.\ even\ n\ \&\ x = real\ n * (pi/2)$
apply (*subgoal-tac* $\exists n::nat.\ \sim\ even\ n\ \&\ x + pi/2 = real\ n * (pi/2)$)
apply (*clarify*, *rule-tac* $x = n - 1$ **in** *exI*)
apply (*force simp add*: *odd-Suc-mult-two-ex real-of-nat-Suc left-distrib*)
apply (*rule cos-zero-lemma*)
apply (*simp-all add*: *add-increasing*)
done

lemma *cos-zero-iff*:
$(cos\ x = 0) =$
$((\exists n::nat.\ \sim even\ n\ \&\ (x = real\ n * (pi/2)))\ |$
$(\exists n::nat.\ \sim even\ n\ \&\ (x = -(real\ n * (pi/2)))))$
apply (*rule iffI*)
apply (*cut-tac linorder-linear* [*of* 0 *x*], *safe*)
apply (*drule cos-zero-lemma*, *assumption*+)
apply (*cut-tac* $x{=}{-}x$ **in** *cos-zero-lemma*, *simp*, *simp*)
apply (*force simp add*: *minus-equation-iff* [*of x*])
apply (*auto simp only*: *odd-Suc-mult-two-ex real-of-nat-Suc left-distrib*)
apply (*auto simp add*: *cos-add*)
done

lemma *sin-zero-iff*:
$(sin\ x = 0) =$
$((\exists n::nat.\ even\ n\ \&\ (x = real\ n * (pi/2)))\ |$
$(\exists n::nat.\ even\ n\ \&\ (x = -(real\ n * (pi/2)))))$
apply (*rule iffI*)
apply (*cut-tac linorder-linear* [*of* 0 *x*], *safe*)
apply (*drule sin-zero-lemma*, *assumption*+)
apply (*cut-tac* $x{=}{-}x$ **in** *sin-zero-lemma*, *simp*, *simp*, *safe*)
apply (*force simp add*: *minus-equation-iff* [*of x*])
apply (*auto simp add*: *even-mult-two-ex*)
done

lemma *cos-monotone-0-pi*: **assumes** $0 \leq y$ **and** $y < x$ **and** $x \leq pi$
shows $cos\ x < cos\ y$

proof −
have − (*x* − *y*) < *0* **by** (*auto*!)

from *MVT2*[*OF* ‹*y* < *x*› *DERIV-cos*[*THEN impI*, *THEN allI*]]
obtain *z* **where** *y* < *z* **and** *z* < *x* **and** *cos-diff*: *cos x* − *cos y* = (*x* − *y*) ∗ − *sin z* **by** *auto*
hence *0* < *z* **and** *z* < *pi* **by** (*auto*!)
hence *0* < *sin z* **using** *sin-gt-zero-pi* **by** *auto*
hence *cos x* − *cos y* < *0* **unfolding** *cos-diff minus-mult-commute*[*symmetric*] **using** ‹− (*x* − *y*) < *0*› **by** (*rule mult-pos-neg2*)
thus *?thesis* **by** *auto*
qed

lemma *cos-monotone-0-pi'*: **assumes** *0* ≤ *y* **and** *y* ≤ *x* **and** *x* ≤ *pi* **shows** *cos x* ≤ *cos y*
proof (*cases y* < *x*)
case *True* **show** *?thesis* **using** *cos-monotone-0-pi*[*OF* ‹*0* ≤ *y*› *True* ‹*x* ≤ *pi*›] **by** *auto*
next
case *False* **hence** *y* = *x* **using** ‹*y* ≤ *x*› **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *cos-monotone-minus-pi-0*: **assumes** −*pi* ≤ *y* **and** *y* < *x* **and** *x* ≤ *0*
shows *cos y* < *cos x*
proof −
have *0* ≤ −*x* **and** −*x* < −*y* **and** −*y* ≤ *pi* **by** (*auto*!)
from *cos-monotone-0-pi*[*OF this*]
show *?thesis* **unfolding** *cos-minus* **.**
qed

lemma *cos-monotone-minus-pi-0'*: **assumes** −*pi* ≤ *y* **and** *y* ≤ *x* **and** *x* ≤ *0*
shows *cos y* ≤ *cos x*
proof (*cases y* < *x*)
case *True* **show** *?thesis* **using** *cos-monotone-minus-pi-0*[*OF* ‹−*pi* ≤ *y*› *True* ‹*x* ≤ *0*›] **by** *auto*
next
case *False* **hence** *y* = *x* **using** ‹*y* ≤ *x*› **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *sin-monotone-2pi'*: **assumes** − (*pi* / *2*) ≤ *y* **and** *y* ≤ *x* **and** *x* ≤ *pi* / *2*
shows *sin y* ≤ *sin x*
proof −
have *0* ≤ *y* + *pi* / *2* **and** *y* + *pi* / *2* ≤ *x* + *pi* / *2* **and** *x* + *pi* /*2* ≤ *pi* **using** *pi-ge-two* **by** (*auto*!)
from *cos-monotone-0-pi'*[*OF this*] **show** *?thesis* **unfolding** *minus-sin-cos-eq*[*symmetric*] **by** *auto*
qed

63.11 Tangent

definition
tan :: *real* => *real* **where**
tan x = (*sin x*)/(*cos x*)

lemma *tan-zero* [*simp*]: *tan 0* = *0*
by (*simp add*: *tan-def*)

lemma *tan-pi* [*simp*]: *tan pi* = *0*
by (*simp add*: *tan-def*)

lemma *tan-npi* [*simp*]: *tan* (*real* (*n*::*nat*) ∗ *pi*) = *0*
by (*simp add*: *tan-def*)

lemma *tan-minus* [*simp*]: *tan* (−*x*) = − *tan x*
by (*simp add*: *tan-def minus-mult-left*)

lemma *tan-periodic* [*simp*]: *tan* (*x* + *2*∗*pi*) = *tan x*
by (*simp add*: *tan-def*)

lemma *lemma-tan-add1*:
[| *cos x* ≠ *0*; *cos y* ≠ *0* |]
==> *1* − *tan*(*x*)∗*tan*(*y*) = *cos* (*x* + *y*)/(*cos x* ∗ *cos y*)
apply (*simp add*: *tan-def divide-inverse*)
apply (*auto simp del*: *inverse-mult-distrib*
simp add: *inverse-mult-distrib* [*symmetric*] *mult-ac*)
apply (*rule-tac c1* = *cos x* ∗ *cos y* **in** *real-mult-right-cancel* [*THEN subst*])
apply (*auto simp del*: *inverse-mult-distrib*
simp add: *mult-assoc left-diff-distrib cos-add*)
done

lemma *add-tan-eq*:
[| *cos x* ≠ *0*; *cos y* ≠ *0* |]
==> *tan x* + *tan y* = *sin*(*x* + *y*)/(*cos x* ∗ *cos y*)
apply (*simp add*: *tan-def*)
apply (*rule-tac c1* = *cos x* ∗ *cos y* **in** *real-mult-right-cancel* [*THEN subst*])
apply (*auto simp add*: *mult-assoc left-distrib*)
apply (*simp add*: *sin-add*)
done

lemma *tan-add*:
[| *cos x* ≠ *0*; *cos y* ≠ *0*; *cos* (*x* + *y*) ≠ *0* |]
==> *tan*(*x* + *y*) = (*tan*(*x*) + *tan*(*y*))/(*1* − *tan*(*x*) ∗ *tan*(*y*))
apply (*simp* (*no-asm-simp*) *add*: *add-tan-eq lemma-tan-add1*)
apply (*simp add*: *tan-def*)
done

lemma *tan-double*:
[| *cos x* ≠ *0*; *cos* (*2* ∗ *x*) ≠ *0* |]

```
    ==> tan (2 * x) = (2 * tan x)/(1 - (tan(x) ^ 2))
apply (insert tan-add [of x x])
apply (simp add: mult-2 [symmetric])
apply (auto simp add: numeral-2-eq-2)
done

lemma tan-gt-zero: [| 0 < x; x < pi/2 |] ==> 0 < tan x
by (simp add: tan-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi)

lemma tan-less-zero:
  assumes lb: - pi/2 < x and x < 0 shows tan x < 0
proof -
  have 0 < tan (- x) using prems by (simp only: tan-gt-zero)
  thus ?thesis by simp
qed

lemma tan-half: fixes x :: real assumes - (pi / 2) < x and x < pi / 2
  shows tan x = sin (2 * x) / (cos (2 * x) + 1)
proof -
  from cos-gt-zero-pi[OF ‹- (pi / 2) < x› ‹x < pi / 2›]
  have cos x ≠ 0 by auto

  have minus-cos-2x: ⋀X. X - cos (2*x) = X - (cos x) ^ 2 + (sin x) ^ 2
unfolding cos-double by algebra

  have tan x = (tan x + tan x) / 2 by auto
  also have ... = sin (x + x) / (cos x * cos x) / 2 unfolding add-tan-eq[OF ‹cos
x ≠ 0› ‹cos x ≠ 0›] ..
  also have ... = sin (2 * x) / ((cos x) ^ 2 + (cos x) ^ 2 + cos (2*x) - cos
(2*x)) unfolding divide-divide-eq-left numeral-2-eq-2 by auto
  also have ... = sin (2 * x) / ((cos x) ^ 2 + cos (2*x) + (sin x) ^2) unfolding
minus-cos-2x by auto
  also have ... = sin (2 * x) / (cos (2*x) + 1) by auto
  finally show ?thesis .
qed

lemma lemma-DERIV-tan:
    cos x ≠ 0 ==> DERIV (%x. sin(x)/cos(x)) x :> inverse((cos x)^2)
apply (rule lemma-DERIV-subst)
apply (best intro!: DERIV-intros intro: DERIV-chain2)
apply (auto simp add: divide-inverse numeral-2-eq-2)
done

lemma DERIV-tan [simp]: cos x ≠ 0 ==> DERIV tan x :> inverse((cos x)^2)
by (auto dest: lemma-DERIV-tan simp add: tan-def [symmetric])

lemma isCont-tan [simp]: cos x ≠ 0 ==> isCont tan x
by (rule DERIV-tan [THEN DERIV-isCont])
```

lemma *LIM-cos-div-sin* [*simp*]: (%*x*. *cos*(*x*)/*sin*(*x*)) −− *pi*/2 −−> 0
apply (*subgoal-tac* (λ*x*. *cos x* ∗ *inverse* (*sin x*)) −− *pi* ∗ *inverse* 2 −−> 0∗1)
apply (*simp add*: *divide-inverse* [*symmetric*])
apply (*rule LIM-mult*)
apply (*rule-tac* [2] *inverse-1* [*THEN subst*])
apply (*rule-tac* [2] *LIM-inverse*)
apply (*simp-all add*: *divide-inverse* [*symmetric*])
apply (*simp-all only*: *isCont-def* [*symmetric*] *cos-pi-half* [*symmetric*] *sin-pi-half* [*symmetric*])
apply (*blast intro*!: *DERIV-isCont DERIV-sin DERIV-cos*)+
done

lemma *lemma-tan-total*: 0 < *y* ==> ∃ *x*. 0 < *x* & *x* < *pi*/2 & *y* < *tan x*
apply (*cut-tac LIM-cos-div-sin*)
apply (*simp only*: *LIM-def*)
apply (*drule-tac x* = *inverse y* **in** *spec*, *safe*, *force*)
apply (*drule-tac* ?*d1*.0 = *s* **in** *pi-half-gt-zero* [*THEN* [2] *real-lbound-gt-zero*], *safe*)
apply (*rule-tac x* = (*pi*/2) − *e* **in** *exI*)
apply (*simp* (*no-asm-simp*))
apply (*drule-tac x* = (*pi*/2) − *e* **in** *spec*)
apply (*auto simp add*: *tan-def*)
apply (*rule inverse-less-iff-less* [*THEN iffD1*])
apply (*auto simp add*: *divide-inverse*)
apply (*rule real-mult-order*)
apply (*subgoal-tac* [3] 0 < *sin e* & 0 < *cos e*)
apply (*auto intro*: *cos-gt-zero sin-gt-zero2 simp add*: *mult-commute*)
done

lemma *tan-total-pos*: 0 ≤ *y* ==> ∃ *x*. 0 ≤ *x* & *x* < *pi*/2 & *tan x* = *y*
apply (*frule order-le-imp-less-or-eq*, *safe*)
prefer 2 **apply** *force*
apply (*drule lemma-tan-total*, *safe*)
apply (*cut-tac f* = *tan* **and** *a* = 0 **and** *b* = *x* **and** *y* = *y* **in** *IVT-objl*)
apply (*auto intro*!: *DERIV-tan* [*THEN DERIV-isCont*])
apply (*drule-tac y* = *xa* **in** *order-le-imp-less-or-eq*)
apply (*auto dest*: *cos-gt-zero*)
done

lemma *lemma-tan-total1*: ∃ *x*. −(*pi*/2) < *x* & *x* < (*pi*/2) & *tan x* = *y*
apply (*cut-tac linorder-linear* [*of* 0 *y*], *safe*)
apply (*drule tan-total-pos*)
apply (*cut-tac* [2] *y*=−*y* **in** *tan-total-pos*, *safe*)
apply (*rule-tac* [3] *x* = −*x* **in** *exI*)
apply (*auto intro*!: *exI*)
done

lemma *tan-total*: *EX*! *x*. −(*pi*/2) < *x* & *x* < (*pi*/2) & *tan x* = *y*
apply (*cut-tac y* = *y* **in** *lemma-tan-total1*, *auto*)
apply (*cut-tac x* = *xa* **and** *y* = *y* **in** *linorder-less-linear*, *auto*)

apply (*subgoal-tac* [*2*] $\exists z.\ y < z\ \&\ z < xa\ \&$ *DERIV tan z :> 0*)
apply (*subgoal-tac* $\exists z.\ xa < z\ \&\ z < y\ \&$ *DERIV tan z :> 0*)
apply (*rule-tac* [*4*] *Rolle*)
apply (*rule-tac* [*2*] *Rolle*)
apply (*auto intro*!: *DERIV-tan DERIV-isCont exI*
simp add: *differentiable-def*)

Now, simulate TRYALL

apply (*rule-tac* [!] *DERIV-tan asm-rl*)
apply (*auto dest*!: *DERIV-unique* [*OF - DERIV-tan*]
simp add: *cos-gt-zero-pi* [*THEN less-imp-neq*, *THEN not-sym*])
done

lemma *tan-monotone*: **assumes** $-(pi\ /\ 2) < y$ **and** $y < x$ **and** $x < pi\ /\ 2$
shows *tan y < tan x*
proof −
have $\forall\ x'.\ y \le x' \wedge x' \le x \longrightarrow$ *DERIV tan x' :> inverse* (*cos x'^2*)
proof (*rule allI*, *rule impI*)
fix x' :: *real* **assume** $y \le x' \wedge x' \le x$
hence $-(pi/2) < x'$ **and** $x' < pi/2$ **by** (*auto*!)
from *cos-gt-zero-pi*[*OF this*]
have $cos\ x' \neq 0$ **by** *auto*
thus *DERIV tan x' :> inverse* (*cos x'^2*) **by** (*rule DERIV-tan*)
qed
from *MVT2*[*OF* ‹$y < x$› *this*]
obtain *z* **where** $y < z$ **and** $z < x$ **and** *tan-diff*: $tan\ x - tan\ y = (x - y) *$ *inverse* $((cos\ z)^2)$ **by** *auto*
hence $-(pi\ /\ 2) < z$ **and** $z < pi\ /\ 2$ **by** (*auto*!)
hence $0 < cos\ z$ **using** *cos-gt-zero-pi* **by** *auto*
hence *inv-pos*: $0 <$ *inverse* $((cos\ z)^2)$ **by** *auto*
have $0 < x - y$ **using** ‹$y < x$› **by** *auto*
from *real-mult-order*[*OF this inv-pos*]
have $0 < tan\ x - tan\ y$ **unfolding** *tan-diff* **by** *auto*
thus *?thesis* **by** *auto*
qed

lemma *tan-monotone'*: **assumes** $-(pi\ /\ 2) < y$ **and** $y < pi\ /\ 2$ **and** $-(pi\ /\ 2) < x$ **and** $x < pi\ /\ 2$
shows $(y < x) = (tan\ y < tan\ x)$
proof
assume $y < x$ **thus** $tan\ y < tan\ x$ **using** *tan-monotone* **and** ‹$-(pi\ /\ 2) < y$› **and** ‹$x < pi\ /\ 2$› **by** *auto*
next
assume $tan\ y < tan\ x$
show $y < x$
proof (*rule ccontr*)
assume ¬ $y < x$ **hence** $x \le y$ **by** *auto*
hence $tan\ x \le tan\ y$
proof (*cases* $x = y$)

case *True* **thus** *?thesis* **by** *auto*
next
case *False* **hence** $x < y$ **using** ‹$x \leq y$› **by** *auto*
from *tan-monotone*[*OF* ‹$-(pi/2) < x$› *this* ‹$y < pi\ /\ 2$›] **show** *?thesis* **by** *auto*
qed
thus *False* **using** ‹*tan y* < *tan x*› **by** *auto*
qed
qed

lemma *tan-inverse*: $1\ /\ (tan\ y) = tan\ (pi\ /\ 2 - y)$ **unfolding** *tan-def sin-cos-eq*[*of y*] *cos-sin-eq*[*of y*] **by** *auto*

lemma *tan-periodic-pi*[*simp*]: $tan\ (x + pi) = tan\ x$
by (*simp add*: *tan-def*)

lemma *tan-periodic-nat*[*simp*]: **fixes** $n :: nat$ **shows** $tan\ (x + real\ n * pi) = tan\ x$
proof (*induct n arbitrary*: *x*)
case (*Suc n*)
have *split-pi-off*: $x + real\ (Suc\ n) * pi = (x + real\ n * pi) + pi$ **unfolding** *Suc-plus1 real-of-nat-add real-of-one real-add-mult-distrib* **by** *auto*
show *?case* **unfolding** *split-pi-off* **using** *Suc* **by** *auto*
qed *auto*

lemma *tan-periodic-int*[*simp*]: **fixes** $i :: int$ **shows** $tan\ (x + real\ i * pi) = tan\ x$
proof (*cases* $0 \leq i$)
case *True* **hence** *i-nat*: $real\ i = real\ (nat\ i)$ **by** *auto*
show *?thesis* **unfolding** *i-nat* **by** *auto*
next
case *False* **hence** *i-nat*: $real\ i = -\ real\ (nat\ (-i))$ **by** *auto*
have $tan\ x = tan\ (x + real\ i * pi - real\ i * pi)$ **by** *auto*
also have $\ldots = tan\ (x + real\ i * pi)$ **unfolding** *i-nat mult-minus-left diff-minus-eq-add* **by** (*rule tan-periodic-nat*)
finally show *?thesis* **by** *auto*
qed

lemma *tan-periodic-n*[*simp*]: $tan\ (x + number\text{-}of\ n * pi) = tan\ x$
using *tan-periodic-int*[*of - number-of n*] **unfolding** *real-number-of* **.**

63.12 Inverse Trigonometric Functions

definition
$arcsin :: real \Rightarrow real$ **where**
$arcsin\ y = (THE\ x.\ -(pi/2) \leq x\ \&\ x \leq pi/2\ \&\ sin\ x = y)$

definition
$arccos :: real \Rightarrow real$ **where**
$arccos\ y = (THE\ x.\ 0 \leq x\ \&\ x \leq pi\ \&\ cos\ x = y)$

definition
arctan :: *real* => *real* **where**
arctan y = (*THE x*. −(*pi*/2) < *x* & *x* < *pi*/2 & *tan x* = *y*)

lemma *arcsin*:
[| −1 ≤ *y*; *y* ≤ 1 |]
==> −(*pi*/2) ≤ *arcsin y* &
arcsin y ≤ *pi*/2 & *sin*(*arcsin y*) = *y*
unfolding *arcsin-def* **by** (*rule theI'* [*OF sin-total*])

lemma *arcsin-pi*:
[| −1 ≤ *y*; *y* ≤ 1 |]
==> −(*pi*/2) ≤ *arcsin y* & *arcsin y* ≤ *pi* & *sin*(*arcsin y*) = *y*
apply (*drule* (*1*) *arcsin*)
apply (*force intro*: *order-trans*)
done

lemma *sin-arcsin* [*simp*]: [| −1 ≤ *y*; *y* ≤ 1 |] ==> *sin*(*arcsin y*) = *y*
by (*blast dest*: *arcsin*)

lemma *arcsin-bounded*:
[| −1 ≤ *y*; *y* ≤ 1 |] ==> −(*pi*/2) ≤ *arcsin y* & *arcsin y* ≤ *pi*/2
by (*blast dest*: *arcsin*)

lemma *arcsin-lbound*: [| −1 ≤ *y*; *y* ≤ 1 |] ==> −(*pi*/2) ≤ *arcsin y*
by (*blast dest*: *arcsin*)

lemma *arcsin-ubound*: [| −1 ≤ *y*; *y* ≤ 1 |] ==> *arcsin y* ≤ *pi*/2
by (*blast dest*: *arcsin*)

lemma *arcsin-lt-bounded*:
[| −1 < *y*; *y* < 1 |] ==> −(*pi*/2) < *arcsin y* & *arcsin y* < *pi*/2
apply (*frule order-less-imp-le*)
apply (*frule-tac y* = *y* **in** *order-less-imp-le*)
apply (*frule arcsin-bounded*)
apply (*safe*, *simp*)
apply (*drule-tac y* = *arcsin y* **in** *order-le-imp-less-or-eq*)
apply (*drule-tac* [*2*] *y* = *pi*/2 **in** *order-le-imp-less-or-eq*, *safe*)
apply (*drule-tac* [!] *f* = *sin* **in** *arg-cong*, *auto*)
done

lemma *arcsin-sin*: [|−(*pi*/2) ≤ *x*; *x* ≤ *pi*/2 |] ==> *arcsin*(*sin x*) = *x*
apply (*unfold arcsin-def*)
apply (*rule the1-equality*)
apply (*rule sin-total*, *auto*)
done

lemma *arccos*:

[| $-1 \leq y$; $y \leq 1$ |]
==> $0 \leq$ *arccos* y & *arccos* $y \leq$ *pi* & *cos*(*arccos* y) = y
unfolding *arccos-def* **by** (*rule theI'* [*OF cos-total*])

lemma *cos-arccos* [*simp*]: [| $-1 \leq y$; $y \leq 1$ |] ==> *cos*(*arccos* y) = y
by (*blast dest*: *arccos*)

lemma *arccos-bounded*: [| $-1 \leq y$; $y \leq 1$ |] ==> $0 \leq$ *arccos* y & *arccos* $y \leq$ *pi*
by (*blast dest*: *arccos*)

lemma *arccos-lbound*: [| $-1 \leq y$; $y \leq 1$ |] ==> $0 \leq$ *arccos* y
by (*blast dest*: *arccos*)

lemma *arccos-ubound*: [| $-1 \leq y$; $y \leq 1$ |] ==> *arccos* $y \leq$ *pi*
by (*blast dest*: *arccos*)

lemma *arccos-lt-bounded*:
[| $-1 < y$; $y < 1$ |]
==> $0 <$ *arccos* y & *arccos* $y <$ *pi*
apply (*frule order-less-imp-le*)
apply (*frule-tac* $y = y$ **in** *order-less-imp-le*)
apply (*frule arccos-bounded*, *auto*)
apply (*drule-tac* $y =$ *arccos* y **in** *order-le-imp-less-or-eq*)
apply (*drule-tac* [2] $y =$ *pi* **in** *order-le-imp-less-or-eq*, *auto*)
apply (*drule-tac* [!] $f =$ *cos* **in** *arg-cong*, *auto*)
done

lemma *arccos-cos*: [|$0 \leq x$; $x \leq$ *pi* |] ==> *arccos*(*cos* x) = x
apply (*simp add*: *arccos-def*)
apply (*auto intro*!: *the1-equality cos-total*)
done

lemma *arccos-cos2*: [|$x \leq 0$; $-$*pi* $\leq x$ |] ==> *arccos*(*cos* x) = $-x$
apply (*simp add*: *arccos-def*)
apply (*auto intro*!: *the1-equality cos-total*)
done

lemma *cos-arcsin*: $[\![-1 \leq x; x \leq 1]\!] \Longrightarrow$ *cos* (*arcsin* x) = *sqrt* (1 $- x^2$)
apply (*subgoal-tac* $x^2 \leq 1$)
apply (*rule power2-eq-imp-eq*)
apply (*simp add*: *cos-squared-eq*)
apply (*rule cos-ge-zero*)
apply (*erule* (1) *arcsin-lbound*)
apply (*erule* (1) *arcsin-ubound*)
apply *simp*
apply (*subgoal-tac* $|x|^2 \leq 1^2$, *simp*)
apply (*rule power-mono*, *simp*, *simp*)
done

lemma *sin-arccos*: $[\![-1 \leq x; x \leq 1]\!] \Longrightarrow sin\ (arccos\ x) = sqrt\ (1 - x^2)$
apply (*subgoal-tac* $x^2 \leq 1$)
apply (*rule power2-eq-imp-eq*)
apply (*simp add*: *sin-squared-eq*)
apply (*rule sin-ge-zero*)
apply (*erule* (*1*) *arccos-lbound*)
apply (*erule* (*1*) *arccos-ubound*)
apply *simp*
apply (*subgoal-tac* $|x|^2 \leq 1^2$, *simp*)
apply (*rule power-mono*, *simp*, *simp*)
done

lemma *arctan* [*simp*]:
  $-(pi/2) < arctan\ y\ \&\ arctan\ y < pi/2\ \&\ tan\ (arctan\ y) = y$
unfolding *arctan-def* **by** (*rule theI'* [*OF tan-total*])

lemma *tan-arctan*: $tan(arctan\ y) = y$
by *auto*

lemma *arctan-bounded*: $-(pi/2) < arctan\ y\ \&\ arctan\ y < pi/2$
by (*auto simp only*: *arctan*)

lemma *arctan-lbound*: $-(pi/2) < arctan\ y$
by *auto*

lemma *arctan-ubound*: $arctan\ y < pi/2$
by (*auto simp only*: *arctan*)

lemma *arctan-tan*:
  $[|-(pi/2) < x; x < pi/2\ |] ==> arctan(tan\ x) = x$
apply (*unfold arctan-def*)
apply (*rule the1-equality*)
apply (*rule tan-total*, *auto*)
done

lemma *arctan-zero-zero* [*simp*]: $arctan\ 0 = 0$
by (*insert arctan-tan* [*of 0*], *simp*)

lemma *cos-arctan-not-zero* [*simp*]: $cos(arctan\ x) \neq 0$
apply (*auto simp add*: *cos-zero-iff*)
apply (*case-tac n*)
apply (*case-tac* [*3*] *n*)
apply (*cut-tac* [*2*] $y = x$ **in** *arctan-ubound*)
apply (*cut-tac* [*4*] $y = x$ **in** *arctan-lbound*)
apply (*auto simp add*: *real-of-nat-Suc left-distrib mult-less-0-iff*)
done

lemma *tan-sec*: $cos\ x \neq 0 ==> 1 + tan(x)\ \hat{}\ 2 = inverse(cos\ x)\ \hat{}\ 2$
apply (*rule power-inverse* [*THEN subst*])

apply (*rule-tac c1* = $(cos\ x)^2$ **in** *real-mult-right-cancel* [*THEN iffD1*])
apply (*auto dest*: *field-power-not-zero*
simp add: *power-mult-distrib left-distrib power-divide tan-def*
mult-assoc power-inverse [*symmetric*])
done

lemma *isCont-inverse-function2*:
fixes *f g* :: *real* ⇒ *real* **shows**
⟦$a < x$; $x < b$;
$\forall z.\ a \leq z \wedge z \leq b \longrightarrow g\ (f\ z) = z$;
$\forall z.\ a \leq z \wedge z \leq b \longrightarrow isCont\ f\ z$⟧
⟹ *isCont g* (*f x*)
apply (*rule isCont-inverse-function*
[**where** *f*=*f* **and** *d*=*min* $(x - a)$ $(b - x)$])
apply (*simp-all add*: *abs-le-iff*)
done

lemma *isCont-arcsin*: ⟦$-1 < x$; $x < 1$⟧ ⟹ *isCont arcsin x*
apply (*subgoal-tac isCont arcsin* (*sin* (*arcsin x*)), *simp*)
apply (*rule isCont-inverse-function2* [**where** *f*=*sin*])
apply (*erule* (*1*) *arcsin-lt-bounded* [*THEN conjunct1*])
apply (*erule* (*1*) *arcsin-lt-bounded* [*THEN conjunct2*])
apply (*fast intro*: *arcsin-sin*, *simp*)
done

lemma *isCont-arccos*: ⟦$-1 < x$; $x < 1$⟧ ⟹ *isCont arccos x*
apply (*subgoal-tac isCont arccos* (*cos* (*arccos x*)), *simp*)
apply (*rule isCont-inverse-function2* [**where** *f*=*cos*])
apply (*erule* (*1*) *arccos-lt-bounded* [*THEN conjunct1*])
apply (*erule* (*1*) *arccos-lt-bounded* [*THEN conjunct2*])
apply (*fast intro*: *arccos-cos*, *simp*)
done

lemma *isCont-arctan*: *isCont arctan x*
apply (*rule arctan-lbound* [*of x*, *THEN dense*, *THEN exE*], *clarify*)
apply (*rule arctan-ubound* [*of x*, *THEN dense*, *THEN exE*], *clarify*)
apply (*subgoal-tac isCont arctan* (*tan* (*arctan x*)), *simp*)
apply (*erule* (*1*) *isCont-inverse-function2* [**where** *f*=*tan*])
apply (*clarify*, *rule arctan-tan*)
apply (*erule* (*1*) *order-less-le-trans*)
apply (*erule* (*1*) *order-le-less-trans*)
apply (*clarify*, *rule isCont-tan*)
apply (*rule less-imp-neq* [*symmetric*])
apply (*rule cos-gt-zero-pi*)
apply (*erule* (*1*) *order-less-le-trans*)
apply (*erule* (*1*) *order-le-less-trans*)
done

lemma *DERIV-arcsin*:

⟦−1 < x; x < 1⟧ ⟹ *DERIV arcsin x :> inverse* (*sqrt* (1 − x^2))
apply (*rule DERIV-inverse-function* [**where** *f=sin* **and** *a=−1* **and** *b=1*])
apply (*rule lemma-DERIV-subst* [*OF DERIV-sin*])
apply (*simp add*: *cos-arcsin*)
apply (*subgoal-tac* $|x|^2 < 1^2$, *simp*)
apply (*rule power-strict-mono*, *simp*, *simp*, *simp*)
apply *assumption*
apply *assumption*
apply *simp*
apply (*erule* (*1*) *isCont-arcsin*)
done

lemma *DERIV-arccos*:
⟦−1 < x; x < 1⟧ ⟹ *DERIV arccos x :> inverse* (− *sqrt* (1 − x^2))
apply (*rule DERIV-inverse-function* [**where** *f=cos* **and** *a=−1* **and** *b=1*])
apply (*rule lemma-DERIV-subst* [*OF DERIV-cos*])
apply (*simp add*: *sin-arccos*)
apply (*subgoal-tac* $|x|^2 < 1^2$, *simp*)
apply (*rule power-strict-mono*, *simp*, *simp*, *simp*)
apply *assumption*
apply *assumption*
apply *simp*
apply (*erule* (*1*) *isCont-arccos*)
done

lemma *DERIV-arctan*: *DERIV arctan x :> inverse* (1 + x^2)
apply (*rule DERIV-inverse-function* [**where** *f=tan* **and** *a=x − 1* **and** *b=x + 1*])
apply (*rule lemma-DERIV-subst* [*OF DERIV-tan*])
apply (*rule cos-arctan-not-zero*)
apply (*simp add*: *power-inverse tan-sec* [*symmetric*])
apply (*subgoal-tac 0 < 1 +* x^2, *simp*)
apply (*simp add*: *add-pos-nonneg*)
apply (*simp*, *simp*, *simp*, *rule isCont-arctan*)
done

63.13 More Theorems about Sin and Cos

lemma *cos-45*: *cos* (*pi / 4*) = *sqrt 2 / 2*
proof −
let *?c = cos* (*pi / 4*) **and** *?s = sin* (*pi / 4*)
have *nonneg*: *0 ≤ ?c*
by (*rule cos-ge-zero*, *rule order-trans* [**where** *y=0*], *simp-all*)
have *0 = cos* (*pi / 4 + pi / 4*)
by *simp*
also have *cos* (*pi / 4 + pi / 4*) = $?c^2 - ?s^2$
by (*simp only*: *cos-add power2-eq-square*)
also have ... = *2 * ?c*^{2} *− 1*
by (*simp add*: *sin-squared-eq*)

```
  finally have ?c^2 = (sqrt 2 / 2)^2
    by (simp add: power-divide)
  thus ?thesis
    using nonneg by (rule power2-eq-imp-eq) simp
qed

lemma cos-30: cos (pi / 6) = sqrt 3 / 2
proof −
  let ?c = cos (pi / 6) and ?s = sin (pi / 6)
  have pos-c: 0 < ?c
    by (rule cos-gt-zero, simp, simp)
  have 0 = cos (pi / 6 + pi / 6 + pi / 6)
    by simp
  also have ... = (?c * ?c − ?s * ?s) * ?c − (?s * ?c + ?c * ?s) * ?s
    by (simp only: cos-add sin-add)
  also have ... = ?c * (?c^2 − 3 * ?s^2)
    by (simp add: algebra-simps power2-eq-square)
  finally have ?c^2 = (sqrt 3 / 2)^2
    using pos-c by (simp add: sin-squared-eq power-divide)
  thus ?thesis
    using pos-c [THEN order-less-imp-le]
    by (rule power2-eq-imp-eq) simp
qed

lemma sin-45: sin (pi / 4) = sqrt 2 / 2
proof −
  have sin (pi / 4) = cos (pi / 2 − pi / 4) by (rule sin-cos-eq)
  also have pi / 2 − pi / 4 = pi / 4 by simp
  also have cos (pi / 4) = sqrt 2 / 2 by (rule cos-45)
  finally show ?thesis .
qed

lemma sin-60: sin (pi / 3) = sqrt 3 / 2
proof −
  have sin (pi / 3) = cos (pi / 2 − pi / 3) by (rule sin-cos-eq)
  also have pi / 2 − pi / 3 = pi / 6 by simp
  also have cos (pi / 6) = sqrt 3 / 2 by (rule cos-30)
  finally show ?thesis .
qed

lemma cos-60: cos (pi / 3) = 1 / 2
apply (rule power2-eq-imp-eq)
apply (simp add: cos-squared-eq sin-60 power-divide)
apply (rule cos-ge-zero, rule order-trans [where y=0], simp-all)
done

lemma sin-30: sin (pi / 6) = 1 / 2
proof −
  have sin (pi / 6) = cos (pi / 2 − pi / 6) by (rule sin-cos-eq)
```

```
  also have pi / 2 − pi / 6 = pi / 3 by simp
  also have cos (pi / 3) = 1 / 2 by (rule cos-60)
  finally show ?thesis .
qed

lemma tan-30: tan (pi / 6) = 1 / sqrt 3
unfolding tan-def by (simp add: sin-30 cos-30)

lemma tan-45: tan (pi / 4) = 1
unfolding tan-def by (simp add: sin-45 cos-45)

lemma tan-60: tan (pi / 3) = sqrt 3
unfolding tan-def by (simp add: sin-60 cos-60)
```

NEEDED??

```
lemma [simp]:
     sin (x + 1 / 2 ∗ real (Suc m) ∗ pi) =
      cos (x + 1 / 2 ∗ real  (m) ∗ pi)
by (simp only: cos-add sin-add real-of-nat-Suc left-distrib right-distrib, auto)
```

NEEDED??

```
lemma [simp]:
     sin (x + real (Suc m) ∗ pi / 2) =
      cos (x + real (m) ∗ pi / 2)
by (simp only: cos-add sin-add real-of-nat-Suc add-divide-distrib left-distrib, auto)

lemma DERIV-sin-add [simp]: DERIV (%x. sin (x + k)) xa :> cos (xa + k)
apply (rule lemma-DERIV-subst)
apply (rule-tac f = sin and g = %x. x + k in DERIV-chain2)
apply (best intro!: DERIV-intros intro: DERIV-chain2)+
apply (simp (no-asm))
done

lemma sin-cos-npi [simp]: sin (real (Suc (2 ∗ n)) ∗ pi / 2) = (−1) ^ n
proof −
  have sin ((real n + 1/2) ∗ pi) = cos (real n ∗ pi)
    by (auto simp add: algebra-simps sin-add)
  thus ?thesis
    by (simp add: real-of-nat-Suc left-distrib add-divide-distrib
                  mult-commute [of pi])
qed

lemma cos-2npi [simp]: cos (2 ∗ real (n::nat) ∗ pi) = 1
by (simp add: cos-double mult-assoc power-add [symmetric] numeral-2-eq-2)

lemma cos-3over2-pi [simp]: cos (3 / 2 ∗ pi) = 0
apply (subgoal-tac cos (pi + pi/2) = 0, simp)
apply (subst cos-add, simp)
done
```

lemma *sin-2npi* [*simp*]: *sin* (*2* ∗ *real* (*n*::*nat*) ∗ *pi*) = *0*
by (*auto simp add*: *mult-assoc*)

lemma *sin-3over2-pi* [*simp*]: *sin* (*3* / *2* ∗ *pi*) = − *1*
apply (*subgoal-tac sin* (*pi* + *pi*/*2*) = − *1*, *simp*)
apply (*subst sin-add*, *simp*)
done

lemma [*simp*]:
cos(*x* + *1* / *2* ∗ *real*(*Suc m*) ∗ *pi*) = −*sin* (*x* + *1* / *2* ∗ *real m* ∗ *pi*)
apply (*simp only*: *cos-add sin-add real-of-nat-Suc right-distrib left-distrib minus-mult-right*, *auto*)
done

lemma [*simp*]: *cos* (*x* + *real*(*Suc m*) ∗ *pi* / *2*) = −*sin* (*x* + *real m* ∗ *pi* / *2*)
by (*simp only*: *cos-add sin-add real-of-nat-Suc left-distrib add-divide-distrib*, *auto*)

lemma *cos-pi-eq-zero* [*simp*]: *cos* (*pi* ∗ *real* (*Suc* (*2* ∗ *m*)) / *2*) = *0*
by (*simp only*: *cos-add sin-add real-of-nat-Suc left-distrib right-distrib add-divide-distrib*, *auto*)

lemma *DERIV-cos-add* [*simp*]: *DERIV* (%*x*. *cos* (*x* + *k*)) *xa* :> − *sin* (*xa* + *k*)
apply (*rule lemma-DERIV-subst*)
apply (*rule-tac f* = *cos* **and** *g* = %*x*. *x* + *k* **in** *DERIV-chain2*)
apply (*best intro*!: *DERIV-intros intro*: *DERIV-chain2*)+
apply (*simp* (*no-asm*))
done

lemma *sin-zero-abs-cos-one*: *sin x* = *0* ==> |*cos x*| = *1*
by (*auto simp add*: *sin-zero-iff even-mult-two-ex*)

lemma *cos-one-sin-zero*: *cos x* = *1* ==> *sin x* = *0*
by (*cut-tac x* = *x* **in** *sin-cos-squared-add3*, *auto*)

63.14 Machins formula

lemma *tan-total-pi4*: **assumes** |*x*| < *1*
shows ∃ *z*. − (*pi* / *4*) < *z* ∧ *z* < *pi* / *4* ∧ *tan z* = *x*
proof −
obtain *z* **where** − (*pi* / *2*) < *z* **and** *z* < *pi* / *2* **and** *tan z* = *x* **using** *tan-total*
by *blast*
have *tan* (*pi* / *4*) = *1* **and** *tan* (− (*pi* / *4*)) = − *1* **using** *tan-45 tan-minus* **by** *auto*
have *z* ≠ *pi* / *4*
proof (*rule ccontr*)
assume ¬ (*z* ≠ *pi* / *4*) **hence** *z* = *pi* / *4* **by** *auto*

```
    have tan z = 1 unfolding ‹z = pi / 4› ‹tan (pi / 4) = 1› ..
    thus False unfolding ‹tan z = x› using ‹|x| < 1› by auto
  qed
  have z ≠ - (pi / 4)
  proof (rule ccontr)
    assume ¬ (z ≠ - (pi / 4)) hence z = - (pi / 4) by auto
    have tan z = - 1 unfolding ‹z = - (pi / 4)› ‹tan (- (pi / 4)) = - 1› ..
    thus False unfolding ‹tan z = x› using ‹|x| < 1› by auto
  qed

  have z < pi / 4
  proof (rule ccontr)
    assume ¬ (z < pi / 4) hence pi / 4 < z using ‹z ≠ pi / 4› by auto
    have - (pi / 2) < pi / 4 using m2pi-less-pi by auto
    from tan-monotone[OF this ‹pi / 4 < z› ‹z < pi / 2›]
    have 1 < x unfolding ‹tan z = x› ‹tan (pi / 4) = 1› .
    thus False using ‹|x| < 1› by auto
  qed
  moreover
  have -(pi / 4) < z
  proof (rule ccontr)
    assume ¬ (-(pi / 4) < z) hence z < - (pi / 4) using ‹z ≠ - (pi / 4)› by
auto
    have -(pi / 4) < pi / 2 using m2pi-less-pi by auto
    from tan-monotone[OF ‹-(pi / 2) < z› ‹z < -(pi / 4)› this]
    have x < - 1 unfolding ‹tan z = x› ‹tan (-(pi / 4)) = - 1› .
    thus False using ‹|x| < 1› by auto
  qed
  ultimately show ?thesis using ‹tan z = x› by auto
qed

lemma arctan-add: assumes |x| ≤ 1 and |y| < 1
  shows arctan x + arctan y = arctan ((x + y) / (1 - x * y))
proof -
 obtain y' where -(pi/4) < y' and y' < pi/4 and tan y' = y using tan-total-pi4[OF
‹|y| < 1›] by blast

  have pi / 4 < pi / 2 by auto

  have ∃ x'. -(pi/4) ≤ x' ∧ x' ≤ pi/4 ∧ tan x' = x
  proof (cases |x| < 1)
    case True from tan-total-pi4[OF this] obtain x' where -(pi/4) < x' and x'
< pi/4 and tan x' = x by blast
    hence -(pi/4) ≤ x' and x' ≤ pi/4 and tan x' = x by auto
    thus ?thesis by auto
  next
    case False
    show ?thesis
    proof (cases x = 1)
```

case *True* **hence** *tan* $(pi/4) = x$ **using** *tan-45* **by** *auto*
moreover
have $-pi \leq pi$ **unfolding** *minus-le-self-iff* **by** *auto*
hence $-(pi/4) \leq pi/4$ **and** $pi/4 \leq pi/4$ **by** *auto*
ultimately show *?thesis* **by** *blast*
next
case *False* **hence** $x = -1$ **using** ‹$\neg\ |x| < 1$› **and** ‹$|x| \leq 1$› **by** *auto*
hence *tan* $(-(pi/4)) = x$ **using** *tan-45 tan-minus* **by** *auto*
moreover
have $-pi \leq pi$ **unfolding** *minus-le-self-iff* **by** *auto*
hence $-(pi/4) \leq pi/4$ **and** $-(pi/4) \leq -(pi/4)$ **by** *auto*
ultimately show *?thesis* **by** *blast*
qed
qed
then obtain x' **where** $-(pi/4) \leq x'$ **and** $x' \leq pi/4$ **and** *tan* $x' = x$ **by** *blast*
hence $-(pi/2) < x'$ **and** $x' < pi/2$ **using** *order-le-less-trans*[*OF* ‹$x' \leq pi/4$› ‹*pi / 4 < pi / 2*›] **by** *auto*

have *cos* $x' \neq 0$ **using** *cos-gt-zero-pi*[*THEN less-imp-neq*] **and** ‹$-(pi/2) < x'$› **and** ‹$x' < pi/2$› **by** *auto*
moreover have *cos* $y' \neq 0$ **using** *cos-gt-zero-pi*[*THEN less-imp-neq*] **and** ‹$-(pi/4) < y'$› **and** ‹$y' < pi/4$› **by** *auto*
ultimately have *cos* $x' *$ *cos* $y' \neq 0$ **by** *auto*

have *divide-nonzero-divide*: $\bigwedge A\ B\ C$ :: *real*. $C \neq 0 \Longrightarrow (A\ /\ C)\ /\ (B\ /\ C) = A\ /\ B$ **by** *auto*
have *divide-mult-commute*: $\bigwedge A\ B\ C\ D$ :: *real*. $A * B\ /\ (C * D) = (A\ /\ C) * (B\ /\ D)$ **by** *auto*

have *tan* $(x' + y') =$ *sin* $(x' + y')\ /$ (*cos* $x' *$ *cos* $y' -$ *sin* $x' *$ *sin* y') **unfolding** *tan-def cos-add* **..**
also have $\ldots =$ (*tan* $x' +$ *tan* y') / ((*cos* $x' *$ *cos* $y' -$ *sin* $x' *$ *sin* y') / (*cos* $x' *$ *cos* y')) **unfolding** *add-tan-eq*[*OF* ‹*cos* $x' \neq 0$› ‹*cos* $y' \neq 0$›] *divide-nonzero-divide*[*OF* ‹*cos* $x' *$ *cos* $y' \neq 0$›] **..**
also have $\ldots =$ (*tan* $x' +$ *tan* y') / (*1* $-$ *tan* $x' *$ *tan* y') **unfolding** *tan-def diff-divide-distrib divide-self*[*OF* ‹*cos* $x' *$ *cos* $y' \neq 0$›] **unfolding** *divide-mult-commute* **..**
finally have *tan-eq*: *tan* $(x' + y') = (x + y)\ /\ (1 - x * y)$ **unfolding** ‹*tan* $y' = y$› ‹*tan* $x' = x$› **.**

have *arctan* (*tan* $(x' + y')) = x' + y'$ **using** ‹$-(pi/4) < y'$› ‹$-(pi/4) \leq x'$› ‹$y' < pi/4$› **and** ‹$x' \leq pi/4$› **by** (*auto intro*!: *arctan-tan*)
moreover have *arctan* (*tan* $(x')) = x'$ **using** ‹$-(pi/2) < x'$› **and** ‹$x' < pi/2$› **by** (*auto intro*!: *arctan-tan*)
moreover have *arctan* (*tan* $(y')) = y'$ **using** ‹$-(pi/4) < y'$› **and** ‹$y' < pi/4$› **by** (*auto intro*!: *arctan-tan*)
ultimately have *arctan* $x +$ *arctan* $y =$ *arctan* (*tan* $(x' + y'))$ **unfolding** ‹*tan* $y' = y$› [*symmetric*] ‹*tan* $x' = x$›[*symmetric*] **by** *auto*
thus *arctan* $x +$ *arctan* $y =$ *arctan* $((x + y)\ /\ (1 - x * y))$ **unfolding** *tan-eq* **.**

qed

lemma *arctan1-eq-pi4*: *arctan 1 = pi / 4* **unfolding** *tan-45*[*symmetric*] **by** (*rule arctan-tan*, *auto simp add*: *m2pi-less-pi*)

theorem *machin*: *pi / 4 = 4 * arctan (1/5) − arctan (1 / 239)*
proof −
have *|1 / 5| < (1 :: real)* **by** *auto*
from *arctan-add*[*OF less-imp-le*[*OF this*] *this*]
have *2 * arctan (1 / 5) = arctan (5 / 12)* **by** *auto*
moreover
have *|5 / 12| < (1 :: real)* **by** *auto*
from *arctan-add*[*OF less-imp-le*[*OF this*] *this*]
have *2 * arctan (5 / 12) = arctan (120 / 119)* **by** *auto*
moreover
have *|1| ≤ (1::real)* **and** *|1 / 239| < (1::real)* **by** *auto*
from *arctan-add*[*OF this*]
have *arctan 1 + arctan (1 / 239) = arctan (120 / 119)* **by** *auto*
ultimately have *arctan 1 + arctan (1 / 239) = 4 * arctan (1 / 5)* **by** *auto*
thus *?thesis* **unfolding** *arctan1-eq-pi4* **by** *algebra*
qed

63.15 Introducing the arcus tangens power series

lemma *monoseq-arctan-series*: **fixes** *x :: real*
assumes *|x| ≤ 1* **shows** *monoseq (λ n. 1 / real (n*2+1) * x^(n*2+1))* (**is** *monoseq ?a*)
proof (*cases x = 0*) **case** *True* **thus** *?thesis* **unfolding** *monoseq-def One-nat-def* **by** *auto*
next
case *False*
have *norm x ≤ 1* **and** *x ≤ 1* **and** *−1 ≤ x* **using** *assms* **by** *auto*
show *monoseq ?a*
proof −
{ fix *n* **fix** *x :: real* **assume** *0 ≤ x* **and** *x ≤ 1*
have *1 / real (Suc (Suc n * 2)) * x ^ Suc (Suc n * 2) ≤ 1 / real (Suc (n * 2)) * x ^ Suc (n * 2)*
proof (*rule mult-mono*)
show *1 / real (Suc (Suc n * 2)) ≤ 1 / real (Suc (n * 2))* **by** (*rule frac-le*) *simp-all*
show *0 ≤ 1 / real (Suc (n * 2))* **by** *auto*
show *x ^ Suc (Suc n * 2) ≤ x ^ Suc (n * 2)* **by** (*rule power-decreasing*) (*simp-all add*: *‹0 ≤ x› ‹x ≤ 1›*)
show *0 ≤ x ^ Suc (Suc n * 2)* **by** (*rule zero-le-power*) (*simp add*: *‹0 ≤ x›*)
qed
} note *mono = this*

show *?thesis*
proof (*cases 0 ≤ x*)

```
    case True from mono[OF this ‹x ≤ 1›, THEN allI]
    show ?thesis unfolding Suc-plus1[symmetric] by (rule mono-SucI2)
   next
    case False hence 0 ≤ −x and −x ≤ 1 using ‹−1 ≤ x› by auto
    from mono[OF this]
    have ⋀n. 1 / real (Suc (Suc n * 2)) * x ^ Suc (Suc n * 2) ≥ 1 / real (Suc
(n * 2)) * x ^ Suc (n * 2) using ‹0 ≤ −x› by auto
   thus ?thesis unfolding Suc-plus1[symmetric] by (rule mono-SucI1[OF allI])
   qed
  qed
qed

lemma zeroseq-arctan-series: fixes x :: real
  assumes |x| ≤ 1 shows (λ n. 1 / real (n*2+1) * x^(n*2+1)) ----> 0 (is
?a ----> 0)
proof (cases x = 0) case True thus ?thesis unfolding One-nat-def by (auto
simp add: LIMSEQ-const)
next
  case False
  have norm x ≤ 1 and x ≤ 1 and −1 ≤ x using assms by auto
  show ?a ----> 0
  proof (cases |x| < 1)
    case True hence norm x < 1 by auto
    from LIMSEQ-mult[OF LIMSEQ-inverse-real-of-nat LIMSEQ-power-zero[OF
‹norm x < 1›, THEN LIMSEQ-Suc]]
    have (λn. 1 / real (n + 1) * x ^ (n + 1)) ----> 0
      unfolding inverse-eq-divide Suc-plus1 by simp
    then show ?thesis using pos2 by (rule LIMSEQ-linear)
  next
    case False hence x = −1 ∨ x = 1 using ‹|x| ≤ 1› by auto
    hence n-eq: ⋀ n. x ^ (n * 2 + 1) = x unfolding One-nat-def by auto
    from LIMSEQ-mult[OF LIMSEQ-inverse-real-of-nat[THEN LIMSEQ-linear,
OF pos2, unfolded inverse-eq-divide] LIMSEQ-const[of x]]
    show ?thesis unfolding n-eq Suc-plus1 by auto
  qed
qed

lemma summable-arctan-series: fixes x :: real and n :: nat
  assumes |x| ≤ 1 shows summable (λ k. (−1)^k * (1 / real (k*2+1) * x ^
(k*2+1))) (is summable (?c x))
 by (rule summable-Leibniz(1), rule zeroseq-arctan-series[OF assms], rule monoseq-arctan-series[OF
assms])

lemma less-one-imp-sqr-less-one: fixes x :: real assumes |x| < 1 shows x^2 <
1
proof −
  from mult-mono1[OF less-imp-le[OF ‹|x| < 1›] abs-ge-zero[of x]]
  have | x^2 | < 1 using ‹| x | < 1› unfolding numeral-2-eq-2 power-Suc2 by
auto
```

thus *?thesis* **using** *zero-le-power2* **by** *auto*
qed

lemma *DERIV-arctan-series*: **assumes** | *x* | < *1*
shows *DERIV* (λ *x'*. ∑ *k*. (−*1*) ^*k* ∗ (*1* / *real* (*k*∗*2*+*1*) ∗ *x'* ^ (*k*∗*2*+*1*))) *x* :>
(∑ *k*. (−*1*) ^*k* ∗ *x*^(*k*∗*2*)) (**is** *DERIV ?arctan* - :> *?Int*)
proof −
let *?f n* = *if even n then* (−*1*) ^(*n div 2*) ∗ *1* / *real* (*Suc n*) *else 0*

{ **fix** *n* :: *nat* **assume** *even n* **hence** *2* ∗ (*n div 2*) = *n* **by** *presburger* } **note**
n-even=*this*
have *if-eq*: ⋀ *n x'*. *?f n* ∗ *real* (*Suc n*) ∗ *x'*^*n* = (*if even n then* (−*1*) ^(*n div 2*)
∗ *x'*^(*2* ∗ (*n div 2*)) *else 0*) **using** *n-even* **by** *auto*

{ **fix** *x* :: *real* **assume** |*x*| < *1* **hence** *x*^*2* < *1* **by** (*rule less-one-imp-sqr-less-one*)
have *summable* (λ *n*. −*1* ^ *n* ∗ (*x*^*2*) ^*n*)
by (*rule summable-Leibniz*(*1*), *auto intro*!: *LIMSEQ-realpow-zero monoseq-realpow*
‹*x*^*2* < *1*› *order-less-imp-le*[*OF* ‹*x*^*2* < *1*›])
hence *summable* (λ *n*. −*1* ^ *n* ∗ *x*^(*2*∗*n*)) **unfolding** *power-mult* **.**
} **note** *summable-Integral* = *this*

{ **fix** *f* :: *nat* ⇒ *real*
have ⋀ *x*. *f sums x* = (λ *n*. *if even n then f* (*n div 2*) *else 0*) *sums x*
proof
fix *x* :: *real* **assume** *f sums x*
from *sums-if*[*OF sums-zero this*]
show (λ *n*. *if even n then f* (*n div 2*) *else 0*) *sums x* **by** *auto*
next
fix *x* :: *real* **assume** (λ *n*. *if even n then f* (*n div 2*) *else 0*) *sums x*
from *LIMSEQ-linear*[*OF this*[*unfolded sums-def*] *pos2*, *unfolded sum-split-even-odd*[*unfolded*
mult-commute]]
show *f sums x* **unfolding** *sums-def* **by** *auto*
qed
hence *op sums f* = *op sums* (λ *n*. *if even n then f* (*n div 2*) *else 0*) **..**
} **note** *sums-even* = *this*

have *Int-eq*: (∑ *n*. *?f n* ∗ *real* (*Suc n*) ∗ *x*^*n*) = *?Int* **unfolding** *if-eq mult-commute*[*of*
- *2*] *suminf-def sums-even*[*of* λ *n*. −*1* ^ *n* ∗ *x* ^ (*2* ∗ *n*), *symmetric*]
by *auto*

{ **fix** *x* :: *real*
have *if-eq'*: ⋀ *n*. (*if even n then* −*1* ^ (*n div 2*) ∗ *1* / *real* (*Suc n*) *else 0*) ∗ *x*
^ *Suc n* =
(*if even n then* −*1* ^ (*n div 2*) ∗ (*1* / *real* (*Suc* (*2* ∗ (*n div 2*))) ∗ *x* ^ *Suc* (*2*
∗ (*n div 2*))) *else 0*)
using *n-even* **by** *auto*
have *idx-eq*: ⋀ *n*. *n* ∗ *2* + *1* = *Suc* (*2* ∗ *n*) **by** *auto*
have (∑ *n*. *?f n* ∗ *x*^(*Suc n*)) = *?arctan x* **unfolding** *if-eq' idx-eq suminf-def*
sums-even[*of* λ *n*. −*1* ^ *n* ∗ (*1* / *real* (*Suc* (*2* ∗ *n*)) ∗ *x* ^ *Suc* (*2* ∗ *n*)), *symmetric*]

```
    by auto
  } note arctan-eq = this

  have DERIV (λ x. ∑ n. ?f n * x^(Suc n)) x :> (∑ n. ?f n * real (Suc n) *
x^n)
  proof (rule DERIV-power-series')
    show x ∈ {- 1 <..< 1} using ‹| x | < 1› by auto
    { fix x' :: real assume x'-bounds: x' ∈ {- 1 <..< 1}
      hence |x'| < 1 by auto

      let ?S = ∑ n. (-1)^n * x'^(2 * n)
      show summable (λ n. ?f n * real (Suc n) * x'^n) unfolding if-eq
        by (rule sums-summable[where l=0 + ?S], rule sums-if, rule sums-zero,
rule summable-sums, rule summable-Integral[OF ‹|x'| < 1›])
    }
  qed auto
  thus ?thesis unfolding Int-eq arctan-eq .
qed

lemma arctan-series: assumes | x | ≤ 1
  shows arctan x = (∑ k. (-1)^k * (1 / real (k*2+1) * x ^ (k*2+1))) (is - =
suminf (λ n. ?c x n))
proof -
  let ?c' x n = (-1)^n * x^(n*2)

  { fix r x :: real assume 0 < r and r < 1 and | x | < r
    have |x| < 1 using ‹r < 1› and ‹|x| < r› by auto
    from DERIV-arctan-series[OF this]
    have DERIV (λ x. suminf (?c x)) x :> (suminf (?c' x)) .
  } note DERIV-arctan-suminf = this

 { fix x :: real assume |x| ≤ 1 note summable-Leibniz[OF zeroseq-arctan-series[OF
this] monoseq-arctan-series[OF this]] }
  note arctan-series-borders = this

  { fix x :: real assume |x| < 1 have arctan x = (∑ k. ?c x k)
  proof -
    obtain r where |x| < r and r < 1 using dense[OF ‹|x| < 1›] by blast
    hence 0 < r and -r < x and x < r by auto

    have suminf-eq-arctan-bounded: ⋀ x a b. ⟦ -r < a ; b < r ; a < b ; a ≤ x ; x
≤ b ⟧ ⟹ suminf (?c x) - arctan x = suminf (?c a) - arctan a
    proof -
      fix x a b assume -r < a and b < r and a < b and a ≤ x and x ≤ b
      hence |x| < r by auto
      show suminf (?c x) - arctan x = suminf (?c a) - arctan a
      proof (rule DERIV-isconst2[of a b])
        show a < b and a ≤ x and x ≤ b using ‹a < b› ‹a ≤ x› ‹x ≤ b› by auto
        have ∀ x. -r < x ∧ x < r ⟶ DERIV (λ x. suminf (?c x) - arctan x) x
```

```
:> 0
      proof (rule allI, rule impI)
        fix x assume −r < x ∧ x < r hence |x| < r by auto
        hence |x| < 1 using ‹r < 1› by auto
         have | − (x^2) | < 1 using less-one-imp-sqr-less-one[OF ‹|x| < 1›] by
auto
          hence (λ n. (− (x^2)) ^ n) sums (1 / (1 − (− (x^2)))) unfolding
real-norm-def[symmetric] by (rule geometric-sums)
       hence (?c' x) sums (1 / (1 − (− (x^2)))) unfolding power-mult-distrib[symmetric]
power-mult nat-mult-commute[of - 2] by auto
          hence suminf-c'-eq-geom: inverse (1 + x^2) = suminf (?c' x) using
sums-unique unfolding inverse-eq-divide by auto
         have DERIV (λ x. suminf (?c x)) x :> (inverse (1 + x^2)) unfolding
suminf-c'-eq-geom
          by (rule DERIV-arctan-suminf[OF ‹0 < r› ‹r < 1› ‹|x| < r›])
        from DERIV-add-minus[OF this DERIV-arctan]
       show DERIV (λ x. suminf (?c x) − arctan x) x :> 0 unfolding diff-minus
by auto
       qed
       hence DERIV-in-rball: ∀ y. a ≤ y ∧ y ≤ b ⟶ DERIV (λ x. suminf (?c
x) − arctan x) y :> 0 using ‹−r < a› ‹b < r› by auto
       thus ∀ y. a < y ∧ y < b ⟶ DERIV (λ x. suminf (?c x) − arctan x) y
:> 0 using ‹|x| < r› by auto
       show ∀ y. a ≤ y ∧ y ≤ b ⟶ isCont (λ x. suminf (?c x) − arctan x) y
using DERIV-in-rball DERIV-isCont by auto
     qed
   qed

   have suminf-arctan-zero: suminf (?c 0) − arctan 0 = 0
     unfolding Suc-plus1[symmetric] power-Suc2 mult-zero-right arctan-zero-zero
suminf-zero by auto

   have suminf (?c x) − arctan x = 0
   proof (cases x = 0)
     case True thus ?thesis using suminf-arctan-zero by auto
   next
     case False hence 0 < |x| and − |x| < |x| by auto
     have suminf (?c (−|x|)) − arctan (−|x|) = suminf (?c 0) − arctan 0
       by (rule suminf-eq-arctan-bounded[where x=0 and a=−|x| and b=|x|,
symmetric], auto simp add: ‹|x| < r› ‹−|x| < |x|›)
     moreover
     have suminf (?c x) − arctan x = suminf (?c (−|x|)) − arctan (−|x|)
       by (rule suminf-eq-arctan-bounded[where x=x and a=−|x| and b=|x|],
auto simp add: ‹|x| < r› ‹−|x| < |x|›)
     ultimately
     show ?thesis using suminf-arctan-zero by auto
   qed
   thus ?thesis by auto
 qed } note when-less-one = this
```

```
  show arctan x = suminf (λ n. ?c x n)
  proof (cases |x| < 1)
    case True thus ?thesis by (rule when-less-one)
  next case False hence |x| = 1 using ‹|x| ≤ 1› by auto
    let ?a x n = |1 / real (n*2+1) * x^(n*2+1)|
    let ?diff x n = | arctan x - (∑ i = 0..<n. ?c x i)|
    { fix n :: nat
      have 0 < (1 :: real) by auto
      moreover
      { fix x :: real assume 0 < x and x < 1 hence |x| ≤ 1 and |x| < 1 by auto
        from ‹0 < x› have 0 < 1 / real (0 * 2 + (1::nat)) * x ^ (0 * 2 + 1) by
auto
        note bounds = mp[OF arctan-series-borders(2)[OF ‹|x| ≤ 1›] this, unfolded
when-less-one[OF ‹|x| < 1›, symmetric], THEN spec]
        have 0 < 1 / real (n*2+1) * x^(n*2+1) by (rule mult-pos-pos, auto simp
only: zero-less-power[OF ‹0 < x›], auto)
        hence a-pos: ?a x n = 1 / real (n*2+1) * x^(n*2+1) by (rule abs-of-pos)
        have ?diff x n ≤ ?a x n
        proof (cases even n)
          case True hence sgn-pos: (-1) ^n = (1::real) by auto
          from ‹even n› obtain m where 2 * m = n unfolding even-mult-two-ex
by auto
          from bounds[of m, unfolded this atLeastAtMost-iff]
          have |arctan x - (∑ i = 0..<n. (?c x i))| ≤ (∑ i = 0..<n + 1. (?c x i))
- (∑ i = 0..<n. (?c x i)) by auto
          also have ... = ?c x n unfolding One-nat-def by auto
          also have ... = ?a x n unfolding sgn-pos a-pos by auto
          finally show ?thesis .
        next
          case False hence sgn-neg: (-1) ^n = (-1::real) by auto
            from ‹odd n› obtain m where m-def: 2 * m + 1 = n unfolding
odd-Suc-mult-two-ex by auto
          hence m-plus: 2 * (m + 1) = n + 1 by auto
          from bounds[of m + 1, unfolded this atLeastAtMost-iff, THEN conjunct1]
bounds[of m, unfolded m-def atLeastAtMost-iff, THEN conjunct2]
          have |arctan x - (∑ i = 0..<n. (?c x i))| ≤ (∑ i = 0..<n. (?c x i)) -
(∑ i = 0..<n+1. (?c x i)) by auto
          also have ... = - ?c x n unfolding One-nat-def by auto
          also have ... = ?a x n unfolding sgn-neg a-pos by auto
          finally show ?thesis .
        qed
        hence 0 ≤ ?a x n - ?diff x n by auto
      }
      hence ∀ x ∈ { 0 <..< 1 }. 0 ≤ ?a x n - ?diff x n by auto
      moreover have ⋀x. isCont (λ x. ?a x n - ?diff x n) x
        unfolding real-diff-def divide-inverse
      by (auto intro!: isCont-add isCont-rabs isCont-ident isCont-minus isCont-arctan
isCont-inverse isCont-mult isCont-power isCont-const isCont-setsum)
```

```
    ultimately have 0 ≤ ?a 1 n − ?diff 1 n by (rule LIM-less-bound)
    hence ?diff 1 n ≤ ?a 1 n by auto
   }
   have ?a 1 ----> 0
    unfolding LIMSEQ-rabs-zero power-one divide-inverse One-nat-def
    by (auto intro!: LIMSEQ-mult LIMSEQ-linear LIMSEQ-inverse-real-of-nat)
   have ?diff 1 ----> 0
   proof (rule LIMSEQ-I)
    fix r :: real assume 0 < r
   obtain N :: nat where N-I: ⋀ n. N ≤ n ⟹ ?a 1 n < r using LIMSEQ-D[OF
‹?a 1 ----> 0› ‹0 < r›] by auto
    { fix n assume N ≤ n from ‹?diff 1 n ≤ ?a 1 n› N-I[OF this]
     have norm (?diff 1 n − 0) < r by auto }
    thus ∃ N. ∀ n ≥ N. norm (?diff 1 n − 0) < r by blast
   qed
   from this[unfolded LIMSEQ-rabs-zero real-diff-def add-commute[of arctan 1],
THEN LIMSEQ-add-const, of − arctan 1, THEN LIMSEQ-minus]
   have (?c 1) sums (arctan 1) unfolding sums-def by auto
   hence arctan 1 = (∑ i. ?c 1 i) by (rule sums-unique)

   show ?thesis
   proof (cases x = 1, simp add: ‹arctan 1 = (∑ i. ?c 1 i)›)
    assume x ≠ 1 hence x = −1 using ‹|x| = 1› by auto

    have − (pi / 2) < 0 using pi-gt-zero by auto
    have − (2 * pi) < 0 using pi-gt-zero by auto

   have c-minus-minus: ⋀ i. ?c (− 1) i = − ?c 1 i unfolding One-nat-def by
auto

    have arctan (− 1) = arctan (tan (−(pi / 4))) unfolding tan-45 tan-minus
..
   also have ... = − (pi / 4) by (rule arctan-tan, auto simp add: order-less-trans[OF
‹− (pi / 2) < 0› pi-gt-zero])
    also have ... = − (arctan (tan (pi / 4))) unfolding neg-equal-iff-equal by
(rule arctan-tan[symmetric], auto simp add: order-less-trans[OF ‹− (2 * pi) < 0›
pi-gt-zero])
    also have ... = − (arctan 1) unfolding tan-45 ..
    also have ... = − (∑ i. ?c 1 i) using ‹arctan 1 = (∑ i. ?c 1 i)› by auto
   also have ... = (∑ i. ?c (− 1) i) using suminf-minus[OF sums-summable[OF
‹(?c 1) sums (arctan 1)›]] unfolding c-minus-minus by auto
    finally show ?thesis using ‹x = −1› by auto
   qed
  qed
qed

lemma arctan-half: fixes x :: real
  shows arctan x = 2 * arctan (x / (1 + sqrt(1 + x^2)))
proof −
```

obtain *y* **where** *low*: $-(pi\ /\ 2) < y$ **and** *high*: $y < pi\ /\ 2$ **and** *y-eq*: $tan\ y = x$ **using** *tan-total* **by** *blast*
hence *low2*: $-(pi\ /\ 2) < y\ /\ 2$ **and** *high2*: $y\ /\ 2 < pi\ /\ 2$ **by** *auto*

have *divide-nonzero-divide*: $\bigwedge A\ B\ C :: real.\ C \neq 0 \Longrightarrow A\ /\ B = (A\ /\ C)\ /\ (B\ /\ C)$ **by** *auto*

have $0 < cos\ y$ **using** *cos-gt-zero-pi*[*OF low high*] **.**
hence $cos\ y \neq 0$ **and** *cos-sqrt*: $sqrt\ ((cos\ y)\ \hat{}\ 2) = cos\ y$ **by** *auto*

have $1 + (tan\ y)\hat{}2 = 1 + sin\ y\hat{}2\ /\ cos\ y\hat{}2$ **unfolding** *tan-def power-divide* **..**
also have $\ldots = cos\ y\hat{}2\ /\ cos\ y\hat{}2 + sin\ y\hat{}2\ /\ cos\ y\hat{}2$ **using** ‹$cos\ y \neq 0$› **by** *auto*
also have $\ldots = 1\ /\ cos\ y\hat{}2$ **unfolding** *add-divide-distrib*[*symmetric*] *sin-cos-squared-add2* **..**
finally have $1 + (tan\ y)\hat{}2 = 1\ /\ cos\ y\hat{}2$ **.**

have $sin\ y\ /\ (cos\ y + 1) = tan\ y\ /\ ((cos\ y + 1)\ /\ cos\ y)$ **unfolding** *tan-def divide-nonzero-divide*[*OF* ‹$cos\ y \neq 0$›, *symmetric*] **..**
also have $\ldots = tan\ y\ /\ (1 + 1\ /\ cos\ y)$ **using** ‹$cos\ y \neq 0$› **unfolding** *add-divide-distrib* **by** *auto*
also have $\ldots = tan\ y\ /\ (1 + 1\ /\ sqrt(cos\ y\hat{}2))$ **unfolding** *cos-sqrt* **..**
also have $\ldots = tan\ y\ /\ (1 + sqrt(1\ /\ cos\ y\hat{}2))$ **unfolding** *real-sqrt-divide* **by** *auto*
finally have *eq*: $sin\ y\ /\ (cos\ y + 1) = tan\ y\ /\ (1 + sqrt(1 + (tan\ y)\hat{}2))$ **unfolding** ‹$1 + (tan\ y)\hat{}2 = 1\ /\ cos\ y\hat{}2$› **.**

have $arctan\ x = y$ **using** *arctan-tan low high y-eq* **by** *auto*
also have $\ldots = 2 * (arctan\ (tan\ (y/2)))$ **using** *arctan-tan*[*OF low2 high2*] **by** *auto*
also have $\ldots = 2 * (arctan\ (sin\ y\ /\ (cos\ y + 1)))$ **unfolding** *tan-half*[*OF low2 high2*] **by** *auto*
finally show *?thesis* **unfolding** *eq* ‹$tan\ y = x$› **.**
qed

lemma *arctan-monotone*: **assumes** $x < y$
shows $arctan\ x < arctan\ y$
proof −
obtain *z* **where** $-(pi\ /\ 2) < z$ **and** $z < pi\ /\ 2$ **and** $tan\ z = x$ **using** *tan-total* **by** *blast*
obtain *w* **where** $-(pi\ /\ 2) < w$ **and** $w < pi\ /\ 2$ **and** $tan\ w = y$ **using** *tan-total* **by** *blast*
have $z < w$ **unfolding** *tan-monotone'*[*OF* ‹$-(pi\ /\ 2) < z$› ‹$z < pi\ /\ 2$› ‹$-(pi\ /\ 2) < w$› ‹$w < pi\ /\ 2$›] ‹$tan\ z = x$› ‹$tan\ w = y$› **using** ‹$x < y$› **.**
thus *?thesis*
unfolding ‹$tan\ z = x$›[*symmetric*] *arctan-tan*[*OF* ‹$-(pi\ /\ 2) < z$› ‹$z < pi\ /\ 2$›]
unfolding ‹$tan\ w = y$›[*symmetric*] *arctan-tan*[*OF* ‹$-(pi\ /\ 2) < w$› ‹$w < pi\ /\ 2$›] **.**
qed

lemma *arctan-monotone'*: **assumes** $x \leq y$ **shows** *arctan* $x \leq$ *arctan* y
proof (*cases* $x = y$)
case *False* **hence** $x < y$ **using** ‹$x \leq y$› **by** *auto* **from** *arctan-monotone*[*OF this*]
show *?thesis* **by** *auto*
qed *auto*

lemma *arctan-minus*: *arctan* $(-x) = -$ *arctan* x
proof −
obtain y **where** $-(pi\ /\ 2) < y$ **and** $y < pi\ /\ 2$ **and** *tan* $y = x$ **using** *tan-total*
by *blast*
thus *?thesis* **unfolding** ‹*tan* $y = x$›[*symmetric*] *tan-minus*[*symmetric*] **using**
arctan-tan[*of* y] *arctan-tan*[*of* $-y$] **by** *auto*
qed

lemma *arctan-inverse*: **assumes** $x \neq 0$ **shows** *arctan* $(1\ /\ x) =$ *sgn* $x *$ *pi* $/\ 2$
$-$ *arctan* x
proof −
obtain y **where** $-(pi\ /\ 2) < y$ **and** $y < pi\ /\ 2$ **and** *tan* $y = x$ **using** *tan-total*
by *blast*
hence $y =$ *arctan* x **unfolding** ‹*tan* $y = x$›[*symmetric*] **using** *arctan-tan* **by**
auto

{ fix $y\ x$ :: *real* **assume** $0 < y$ **and** $y <$ *pi* $/2$ **and** $y =$ *arctan* x **and** *tan* $y =$
x **hence** $-(pi\ /\ 2) < y$ **by** *auto*
have *tan* $y > 0$ **using** *tan-monotone'*[*OF* - - ‹$-(pi\ /\ 2) < y$› ‹$y <$ *pi* $/\ 2$›, *of*
0] *tan-zero* ‹$0 < y$› **by** *auto*
hence $x > 0$ **using** ‹*tan* $y = x$› **by** *auto*

have $-(pi\ /\ 2) <$ *pi* $/\ 2 - y$ **using** ‹$y > 0$› ‹$y <$ *pi* $/\ 2$› **by** *auto*
moreover have *pi* $/\ 2 - y <$ *pi* $/\ 2$ **using** ‹$y > 0$› ‹$y <$ *pi* $/\ 2$› **by** *auto*
ultimately have *arctan* $(1\ /\ x) =$ *pi* $/\ 2 - y$ **unfolding** ‹*tan* $y = x$›[*symmetric*]
tan-inverse **using** *arctan-tan* **by** *auto*
hence *arctan* $(1\ /\ x) =$ *sgn* $x *$ *pi* $/\ 2 -$ *arctan* x **unfolding** ‹$y =$ *arctan* x›
real-sgn-pos[*OF* ‹$x > 0$›] **by** *auto*
} **note** *pos-y* = *this*

show *?thesis*
proof (*cases* $y > 0$)
case *True* **from** *pos-y*[*OF this* ‹$y <$ *pi* $/\ 2$› ‹$y =$ *arctan* x› ‹*tan* $y = x$›] **show**
?thesis **.**
next
case *False* **hence** $y \leq 0$ **by** *auto*
moreover have $y \neq 0$
proof (*rule ccontr*)
assume ¬ $y \neq 0$ **hence** $y = 0$ **by** *auto*
have $x = 0$ **unfolding** ‹*tan* $y = x$›[*symmetric*] ‹$y = 0$› *tan-zero* **..**
thus *False* **using** ‹$x \neq 0$› **by** *auto*
qed

ultimately have $y < 0$ **by** *auto*
hence $0 < -y$ **and** $-y < pi / 2$ **using** ‹$-(pi / 2) < y$› **by** *auto*
moreover have $-y = arctan\ (-x)$ **unfolding** *arctan-minus* ‹$y = arctan\ x$› **..**
moreover have $tan\ (-y) = -x$ **unfolding** *tan-minus* ‹$tan\ y = x$› **..**
ultimately have $arctan\ (1 / -x) = sgn\ (-x) * pi / 2 - arctan\ (-x)$ **using** *pos-y* **by** *blast*
hence $arctan\ (-(1 / x)) = -(sgn\ x * pi / 2 - arctan\ x)$ **unfolding** *arctan-minus*[*of x*] *divide-minus-right sgn-minus* **by** *auto*
thus *?thesis* **unfolding** *arctan-minus neg-equal-iff-equal* **.**
qed
qed

theorem *pi-series*: $pi / 4 = (\sum\ k.\ (-1)\hat{}k * 1 / real\ (k*2+1))$ (**is** - = *?SUM*)
proof −
have $pi / 4 = arctan\ 1$ **using** *arctan1-eq-pi4* **by** *auto*
also have $\ldots = ?SUM$ **using** *arctan-series*[*of 1*] **by** *auto*
finally show *?thesis* **by** *auto*
qed

63.16 Existence of Polar Coordinates

lemma *cos-x-y-le-one*: $|x / sqrt\ (x^2 + y^2)| \leq 1$
apply (*rule power2-le-imp-le* [*OF - zero-le-one*])
apply (*simp add*: *abs-divide power-divide divide-le-eq not-sum-power2-lt-zero*)
done

lemma *cos-arccos-abs*: $|y| \leq 1 \Longrightarrow cos\ (arccos\ y) = y$
by (*simp add*: *abs-le-iff*)

lemma *sin-arccos-abs*: $|y| \leq 1 \Longrightarrow sin\ (arccos\ y) = sqrt\ (1 - y^2)$
by (*simp add*: *sin-arccos abs-le-iff*)

lemmas *cos-arccos-lemma1* = *cos-arccos-abs* [*OF cos-x-y-le-one*]

lemmas *sin-arccos-lemma1* = *sin-arccos-abs* [*OF cos-x-y-le-one*]

lemma *polar-ex1*:
$0 < y ==> \exists r\ a.\ x = r * cos\ a\ \&\ y = r * sin\ a$
apply (*rule-tac* $x = sqrt\ (x^2 + y^2)$ **in** *exI*)
apply (*rule-tac* $x = arccos\ (x / sqrt\ (x^2 + y^2))$ **in** *exI*)
apply (*simp add*: *cos-arccos-lemma1*)
apply (*simp add*: *sin-arccos-lemma1*)
apply (*simp add*: *power-divide*)
apply (*simp add*: *real-sqrt-mult* [*symmetric*])
apply (*simp add*: *right-diff-distrib*)
done

lemma *polar-ex2*:
$y < 0 ==> \exists r\ a.\ x = r * cos\ a\ \&\ y = r * sin\ a$

apply (*insert polar-ex1* [**where** *x*=*x* **and** *y*=−*y*], *simp*, *clarify*)
apply (*rule-tac x* = *r* **in** *exI*)
apply (*rule-tac x* = −*a* **in** *exI*, *simp*)
done

lemma *polar-Ex*: ∃ *r a*. *x* = *r* ∗ *cos a* & *y* = *r* ∗ *sin a*
apply (*rule-tac x=0* **and** *y=y* **in** *linorder-cases*)
apply (*erule polar-ex1*)
apply (*rule-tac x=x* **in** *exI*, *rule-tac x=0* **in** *exI*, *simp*)
apply (*erule polar-ex2*)
done

end

64 Complex: Complex Numbers: Rectangular and Polar Representations

theory *Complex*
imports *Transcendental*
begin

datatype *complex* = *Complex real real*

primrec
Re :: *complex* ⇒ *real*
where
Re: *Re* (*Complex x y*) = *x*

primrec
Im :: *complex* ⇒ *real*
where
Im: *Im* (*Complex x y*) = *y*

lemma *complex-surj* [*simp*]: *Complex* (*Re z*) (*Im z*) = *z*
by (*induct z*) *simp*

lemma *complex-equality* [*intro?*]: ⟦*Re x* = *Re y*; *Im x* = *Im y*⟧ ⟹ *x* = *y*
by (*induct x*, *induct y*) *simp*

lemma *expand-complex-eq*: *x* = *y* ⟷ *Re x* = *Re y* ∧ *Im x* = *Im y*
by (*induct x*, *induct y*) *simp*

lemmas *complex-Re-Im-cancel-iff* = *expand-complex-eq*

64.1 Addition and Subtraction

instantiation *complex* :: *ab-group-add*

begin

definition
complex-zero-def: $0 = Complex\ 0\ 0$

definition
complex-add-def: $x + y = Complex\ (Re\ x + Re\ y)\ (Im\ x + Im\ y)$

definition
complex-minus-def: $- x = Complex\ (- Re\ x)\ (- Im\ x)$

definition
complex-diff-def: $x - (y::complex) = x + - y$

lemma *Complex-eq-0* [*simp*]: $Complex\ a\ b = 0 \longleftrightarrow a = 0 \wedge b = 0$
by (*simp add*: *complex-zero-def*)

lemma *complex-Re-zero* [*simp*]: $Re\ 0 = 0$
by (*simp add*: *complex-zero-def*)

lemma *complex-Im-zero* [*simp*]: $Im\ 0 = 0$
by (*simp add*: *complex-zero-def*)

lemma *complex-add* [*simp*]:
$Complex\ a\ b + Complex\ c\ d = Complex\ (a + c)\ (b + d)$
by (*simp add*: *complex-add-def*)

lemma *complex-Re-add* [*simp*]: $Re\ (x + y) = Re\ x + Re\ y$
by (*simp add*: *complex-add-def*)

lemma *complex-Im-add* [*simp*]: $Im\ (x + y) = Im\ x + Im\ y$
by (*simp add*: *complex-add-def*)

lemma *complex-minus* [*simp*]:
$- (Complex\ a\ b) = Complex\ (- a)\ (- b)$
by (*simp add*: *complex-minus-def*)

lemma *complex-Re-minus* [*simp*]: $Re\ (- x) = - Re\ x$
by (*simp add*: *complex-minus-def*)

lemma *complex-Im-minus* [*simp*]: $Im\ (- x) = - Im\ x$
by (*simp add*: *complex-minus-def*)

lemma *complex-diff* [*simp*]:
$Complex\ a\ b - Complex\ c\ d = Complex\ (a - c)\ (b - d)$
by (*simp add*: *complex-diff-def*)

lemma *complex-Re-diff* [*simp*]: $Re\ (x - y) = Re\ x - Re\ y$
by (*simp add*: *complex-diff-def*)

lemma *complex-Im-diff* [*simp*]: *Im* (*x* − *y*) = *Im x* − *Im y*
by (*simp add*: *complex-diff-def*)

instance
by *intro-classes* (*simp-all add*: *complex-add-def complex-diff-def*)

end

64.2 Multiplication and Division

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

definition
complex-one-def: *1* = *Complex 1 0*

definition
complex-mult-def: *x* ∗ *y* =
Complex (*Re x* ∗ *Re y* − *Im x* ∗ *Im y*) (*Re x* ∗ *Im y* + *Im x* ∗ *Re y*)

definition
complex-inverse-def: *inverse x* =
Complex (*Re x* / ((*Re x*)^2 + (*Im x*)^2)) (− *Im x* / ((*Re x*)^2 + (*Im x*)^2))

definition
complex-divide-def: *x* / (*y*::*complex*) = *x* ∗ *inverse y*

lemma *Complex-eq-1* [*simp*]: (*Complex a b* = *1*) = (*a* = *1* ∧ *b* = *0*)
by (*simp add*: *complex-one-def*)

lemma *complex-Re-one* [*simp*]: *Re 1* = *1*
by (*simp add*: *complex-one-def*)

lemma *complex-Im-one* [*simp*]: *Im 1* = *0*
by (*simp add*: *complex-one-def*)

lemma *complex-mult* [*simp*]:
Complex a b ∗ *Complex c d* = *Complex* (*a* ∗ *c* − *b* ∗ *d*) (*a* ∗ *d* + *b* ∗ *c*)
by (*simp add*: *complex-mult-def*)

lemma *complex-Re-mult* [*simp*]: *Re* (*x* ∗ *y*) = *Re x* ∗ *Re y* − *Im x* ∗ *Im y*
by (*simp add*: *complex-mult-def*)

lemma *complex-Im-mult* [*simp*]: *Im* (*x* ∗ *y*) = *Re x* ∗ *Im y* + *Im x* ∗ *Re y*
by (*simp add*: *complex-mult-def*)

lemma *complex-inverse* [*simp*]:
inverse (*Complex a b*) = *Complex* (*a* / (*a*^{2} + *b*^{2})) (− *b* / (*a*^{2} + *b*^{2}))

by (*simp add*: *complex-inverse-def*)

lemma *complex-Re-inverse*:
Re (*inverse x*) = *Re x* / ((*Re x*)^2 + (*Im x*)^2)
by (*simp add*: *complex-inverse-def*)

lemma *complex-Im-inverse*:
Im (*inverse x*) = − *Im x* / ((*Re x*)^2 + (*Im x*)^2)
by (*simp add*: *complex-inverse-def*)

instance
by *intro-classes* (*simp-all add*: *complex-mult-def*
right-distrib left-distrib right-diff-distrib left-diff-distrib
complex-inverse-def complex-divide-def
power2-eq-square add-divide-distrib [*symmetric*]
expand-complex-eq)

end

64.3 Exponentiation

instantiation *complex* :: *recpower*
begin

primrec *power-complex* **where**
z ^ *0* = (*1*::*complex*)
| *z* ^ *Suc n* = (*z*::*complex*) ∗ *z* ^ *n*

instance proof
qed *simp-all*

declare *power-complex.simps* [*simp del*]

end

64.4 Numerals and Arithmetic

instantiation *complex* :: *number-ring*
begin

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

instance
by *intro-classes* (*simp only*: *complex-number-of-def*)

end

lemma *complex-Re-of-nat* [*simp*]: *Re* (*of-nat n*) = *of-nat n*
by (*induct n*) *simp-all*

lemma *complex-Im-of-nat* [*simp*]: *Im* (*of-nat n*) = *0*
by (*induct n*) *simp-all*

lemma *complex-Re-of-int* [*simp*]: *Re* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*) *simp*

lemma *complex-Im-of-int* [*simp*]: *Im* (*of-int z*) = *0*
by (*cases z rule*: *int-diff-cases*) *simp*

lemma *complex-Re-number-of* [*simp*]: *Re* (*number-of v*) = *number-of v*
unfolding *number-of-eq* **by** (*rule complex-Re-of-int*)

lemma *complex-Im-number-of* [*simp*]: *Im* (*number-of v*) = *0*
unfolding *number-of-eq* **by** (*rule complex-Im-of-int*)

lemma *Complex-eq-number-of* [*simp*]:
(*Complex a b* = *number-of w*) = (*a* = *number-of w* ∧ *b* = *0*)
by (*simp add*: *expand-complex-eq*)

64.5 Scalar Multiplication

instantiation *complex* :: *real-field*
begin

definition
complex-scaleR-def: *scaleR r x* = *Complex* (*r* ∗ *Re x*) (*r* ∗ *Im x*)

lemma *complex-scaleR* [*simp*]:
scaleR r (*Complex a b*) = *Complex* (*r* ∗ *a*) (*r* ∗ *b*)
unfolding *complex-scaleR-def* **by** *simp*

lemma *complex-Re-scaleR* [*simp*]: *Re* (*scaleR r x*) = *r* ∗ *Re x*
unfolding *complex-scaleR-def* **by** *simp*

lemma *complex-Im-scaleR* [*simp*]: *Im* (*scaleR r x*) = *r* ∗ *Im x*
unfolding *complex-scaleR-def* **by** *simp*

instance
proof
fix *a b* :: *real* **and** *x y* :: *complex*
show *scaleR a* (*x* + *y*) = *scaleR a x* + *scaleR a y*
by (*simp add*: *expand-complex-eq right-distrib*)
show *scaleR* (*a* + *b*) *x* = *scaleR a x* + *scaleR b x*
by (*simp add*: *expand-complex-eq left-distrib*)
show *scaleR a* (*scaleR b x*) = *scaleR* (*a* ∗ *b*) *x*
by (*simp add*: *expand-complex-eq mult-assoc*)
show *scaleR 1 x* = *x*
by (*simp add*: *expand-complex-eq*)

show *scaleR a x* $*$ *y* = *scaleR a* (*x* $*$ *y*)
by (*simp add*: *expand-complex-eq algebra-simps*)
show *x* $*$ *scaleR a y* = *scaleR a* (*x* $*$ *y*)
by (*simp add*: *expand-complex-eq algebra-simps*)
qed

end

64.6 Properties of Embedding from Reals

abbreviation
complex-of-real :: *real* ⇒ *complex* **where**
complex-of-real ≡ *of-real*

lemma *complex-of-real-def*: *complex-of-real r* = *Complex r 0*
by (*simp add*: *of-real-def complex-scaleR-def*)

lemma *Re-complex-of-real* [*simp*]: *Re* (*complex-of-real z*) = *z*
by (*simp add*: *complex-of-real-def*)

lemma *Im-complex-of-real* [*simp*]: *Im* (*complex-of-real z*) = *0*
by (*simp add*: *complex-of-real-def*)

lemma *Complex-add-complex-of-real* [*simp*]:
Complex x y + *complex-of-real r* = *Complex* (*x*+*r*) *y*
by (*simp add*: *complex-of-real-def*)

lemma *complex-of-real-add-Complex* [*simp*]:
complex-of-real r + *Complex x y* = *Complex* (*r*+*x*) *y*
by (*simp add*: *complex-of-real-def*)

lemma *Complex-mult-complex-of-real*:
Complex x y $*$ *complex-of-real r* = *Complex* (*x*$*$*r*) (*y*$*$*r*)
by (*simp add*: *complex-of-real-def*)

lemma *complex-of-real-mult-Complex*:
complex-of-real r $*$ *Complex x y* = *Complex* (*r*$*$*x*) (*r*$*$*y*)
by (*simp add*: *complex-of-real-def*)

64.7 Vector Norm

instantiation *complex* :: *real-normed-field*
begin

definition
complex-norm-def: *norm z* = *sqrt* ((*Re z*)^2 + (*Im z*)^2)

abbreviation
cmod :: *complex* ⇒ *real* **where**
cmod ≡ *norm*

definition
complex-sgn-def: *sgn x* = *x* /$_R$ *cmod x*

lemmas *cmod-def* = *complex-norm-def*

lemma *complex-norm* [*simp*]: *cmod* (*Complex x y*) = *sqrt* (x^2 + y^2)
by (*simp add*: *complex-norm-def*)

instance
proof
fix *r* :: *real* **and** *x y* :: *complex*
show *0* ≤ *norm x*
by (*induct x*) *simp*
show (*norm x* = *0*) = (*x* = *0*)
by (*induct x*) *simp*
show *norm* (*x* + *y*) ≤ *norm x* + *norm y*
by (*induct x*, *induct y*)
(*simp add*: *real-sqrt-sum-squares-triangle-ineq*)
show *norm* (*scaleR r x*) = |*r*| * *norm x*
by (*induct x*)
(*simp add*: *power-mult-distrib right-distrib* [*symmetric*] *real-sqrt-mult*)
show *norm* (*x* * *y*) = *norm x* * *norm y*
by (*induct x*, *induct y*)
(*simp add*: *real-sqrt-mult* [*symmetric*] *power2-eq-square algebra-simps*)
show *sgn x* = *x* /$_R$ *cmod x* **by**(*simp add*: *complex-sgn-def*)
qed

end

lemma *cmod-unit-one* [*simp*]: *cmod* (*Complex* (*cos a*) (*sin a*)) = *1*
by *simp*

lemma *cmod-complex-polar* [*simp*]:
cmod (*complex-of-real r* * *Complex* (*cos a*) (*sin a*)) = *abs r*
by (*simp add*: *norm-mult*)

lemma *complex-Re-le-cmod*: *Re x* ≤ *cmod x*
unfolding *complex-norm-def*
by (*rule real-sqrt-sum-squares-ge1*)

lemma *complex-mod-minus-le-complex-mod* [*simp*]: − *cmod x* ≤ *cmod x*
by (*rule order-trans* [*OF - norm-ge-zero*], *simp*)

lemma *complex-mod-triangle-ineq2* [*simp*]: *cmod*(*b* + *a*) − *cmod b* ≤ *cmod a*
by (*rule ord-le-eq-trans* [*OF norm-triangle-ineq2*], *simp*)

lemmas *real-sum-squared-expand* = *power2-sum* [**where** *'a*=*real*]

lemma *abs-Re-le-cmod*: $|Re\ x| \leq cmod\ x$
by (*cases x*) *simp*

lemma *abs-Im-le-cmod*: $|Im\ x| \leq cmod\ x$
by (*cases x*) *simp*

64.8 Completeness of the Complexes

interpretation *Re*: *bounded-linear Re*
apply (*unfold-locales*, *simp*, *simp*)
apply (*rule-tac x=1* **in** *exI*)
apply (*simp add*: *complex-norm-def*)
done

interpretation *Im*: *bounded-linear Im*
apply (*unfold-locales*, *simp*, *simp*)
apply (*rule-tac x=1* **in** *exI*)
apply (*simp add*: *complex-norm-def*)
done

lemma *LIMSEQ-Complex*:
$[\![X \longrightarrow a;\ Y \longrightarrow b]\!] \Longrightarrow (\lambda n.\ Complex\ (X\ n)\ (Y\ n)) \longrightarrow Complex\ a\ b$
apply (*rule LIMSEQ-I*)
apply (*subgoal-tac 0 < r / sqrt 2*)
apply (*drule-tac r=r / sqrt 2* **in** *LIMSEQ-D*, *safe*)
apply (*drule-tac r=r / sqrt 2* **in** *LIMSEQ-D*, *safe*)
apply (*rename-tac M N*, *rule-tac x=max M N* **in** *exI*, *safe*)
apply (*simp add*: *real-sqrt-sum-squares-less*)
apply (*simp add*: *divide-pos-pos*)
done

instance *complex* :: *banach*
proof
  fix *X* :: *nat* ⇒ *complex*
  assume *X*: *Cauchy X*
  from *Re.Cauchy* [*OF X*] **have** *1*: $(\lambda n.\ Re\ (X\ n)) \longrightarrow lim\ (\lambda n.\ Re\ (X\ n))$
    by (*simp add*: *Cauchy-convergent-iff convergent-LIMSEQ-iff*)
  from *Im.Cauchy* [*OF X*] **have** *2*: $(\lambda n.\ Im\ (X\ n)) \longrightarrow lim\ (\lambda n.\ Im\ (X\ n))$
    by (*simp add*: *Cauchy-convergent-iff convergent-LIMSEQ-iff*)
  have $X \longrightarrow Complex\ (lim\ (\lambda n.\ Re\ (X\ n)))\ (lim\ (\lambda n.\ Im\ (X\ n)))$
    using *LIMSEQ-Complex* [*OF 1 2*] **by** *simp*
  thus *convergent X*
    by (*rule convergentI*)
qed

64.9 The Complex Number i

definition
  ii :: *complex* (i) **where**

i-def: $ii \equiv Complex\ 0\ 1$

lemma *complex-Re-i* [*simp*]: $Re\ ii = 0$
by (*simp add*: *i-def*)

lemma *complex-Im-i* [*simp*]: $Im\ ii = 1$
by (*simp add*: *i-def*)

lemma *Complex-eq-i* [*simp*]: $(Complex\ x\ y = ii) = (x = 0 \wedge y = 1)$
by (*simp add*: *i-def*)

lemma *complex-i-not-zero* [*simp*]: $ii \neq 0$
by (*simp add*: *expand-complex-eq*)

lemma *complex-i-not-one* [*simp*]: $ii \neq 1$
by (*simp add*: *expand-complex-eq*)

lemma *complex-i-not-number-of* [*simp*]: $ii \neq number\text{-}of\ w$
by (*simp add*: *expand-complex-eq*)

lemma *i-mult-Complex* [*simp*]: $ii * Complex\ a\ b = Complex\ (-\ b)\ a$
by (*simp add*: *expand-complex-eq*)

lemma *Complex-mult-i* [*simp*]: $Complex\ a\ b * ii = Complex\ (-\ b)\ a$
by (*simp add*: *expand-complex-eq*)

lemma *i-complex-of-real* [*simp*]: $ii * complex\text{-}of\text{-}real\ r = Complex\ 0\ r$
by (*simp add*: *i-def complex-of-real-def*)

lemma *complex-of-real-i* [*simp*]: $complex\text{-}of\text{-}real\ r * ii = Complex\ 0\ r$
by (*simp add*: *i-def complex-of-real-def*)

lemma *i-squared* [*simp*]: $ii * ii = -1$
by (*simp add*: *i-def*)

lemma *power2-i* [*simp*]: $ii^2 = -1$
by (*simp add*: *power2-eq-square*)

lemma *inverse-i* [*simp*]: $inverse\ ii = -\ ii$
by (*rule inverse-unique*, *simp*)

64.10 Complex Conjugation

definition
$cnj :: complex \Rightarrow complex$ **where**
$cnj\ z = Complex\ (Re\ z)\ (-\ Im\ z)$

lemma *complex-cnj* [*simp*]: $cnj\ (Complex\ a\ b) = Complex\ a\ (-\ b)$
by (*simp add*: *cnj-def*)

lemma *complex-Re-cnj* [*simp*]: *Re* (*cnj x*) = *Re x*
by (*simp add*: *cnj-def*)

lemma *complex-Im-cnj* [*simp*]: *Im* (*cnj x*) = − *Im x*
by (*simp add*: *cnj-def*)

lemma *complex-cnj-cancel-iff* [*simp*]: (*cnj x* = *cnj y*) = (*x* = *y*)
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-cnj* [*simp*]: *cnj* (*cnj z*) = *z*
by (*simp add*: *cnj-def*)

lemma *complex-cnj-zero* [*simp*]: *cnj 0* = *0*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-zero-iff* [*iff*]: (*cnj z* = *0*) = (*z* = *0*)
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-add*: *cnj* (*x* + *y*) = *cnj x* + *cnj y*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-diff*: *cnj* (*x* − *y*) = *cnj x* − *cnj y*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-minus*: *cnj* (− *x*) = − *cnj x*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-one* [*simp*]: *cnj 1* = *1*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-mult*: *cnj* (*x* ∗ *y*) = *cnj x* ∗ *cnj y*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-inverse*: *cnj* (*inverse x*) = *inverse* (*cnj x*)
by (*simp add*: *complex-inverse-def*)

lemma *complex-cnj-divide*: *cnj* (*x* / *y*) = *cnj x* / *cnj y*
by (*simp add*: *complex-divide-def complex-cnj-mult complex-cnj-inverse*)

lemma *complex-cnj-power*: *cnj* (*x* ^ *n*) = *cnj x* ^ *n*
by (*induct n*, *simp-all add*: *complex-cnj-mult*)

lemma *complex-cnj-of-nat* [*simp*]: *cnj* (*of-nat n*) = *of-nat n*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-of-int* [*simp*]: *cnj* (*of-int z*) = *of-int z*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-number-of* [*simp*]: *cnj* (*number-of w*) = *number-of w*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-scaleR*: *cnj* (*scaleR r x*) = *scaleR r* (*cnj x*)
by (*simp add*: *expand-complex-eq*)

lemma *complex-mod-cnj* [*simp*]: *cmod* (*cnj z*) = *cmod z*
by (*simp add*: *complex-norm-def*)

lemma *complex-cnj-complex-of-real* [*simp*]: *cnj* (*of-real x*) = *of-real x*
by (*simp add*: *expand-complex-eq*)

lemma *complex-cnj-i* [*simp*]: *cnj ii* = − *ii*
by (*simp add*: *expand-complex-eq*)

lemma *complex-add-cnj*: *z* + *cnj z* = *complex-of-real* (*2* ∗ *Re z*)
by (*simp add*: *expand-complex-eq*)

lemma *complex-diff-cnj*: *z* − *cnj z* = *complex-of-real* (*2* ∗ *Im z*) ∗ *ii*
by (*simp add*: *expand-complex-eq*)

lemma *complex-mult-cnj*: *z* ∗ *cnj z* = *complex-of-real* ((*Re z*)^2 + (*Im z*)^2)
by (*simp add*: *expand-complex-eq power2-eq-square*)

lemma *complex-mod-mult-cnj*: *cmod* (*z* ∗ *cnj z*) = (*cmod z*)^2
by (*simp add*: *norm-mult power2-eq-square*)

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

64.11 The Functions *sgn* and *arg*

———— Argand ————-

definition
arg :: *complex* => *real* **where**
arg z = (*SOME a*. *Re*(*sgn z*) = *cos a* & *Im*(*sgn z*) = *sin a* & −*pi* < *a* & *a* ≤ *pi*)

lemma *sgn-eq*: *sgn z* = *z* / *complex-of-real* (*cmod z*)
by (*simp add*: *complex-sgn-def divide-inverse scaleR-conv-of-real mult-commute*)

lemma *i-mult-eq*: *ii* ∗ *ii* = *complex-of-real* (−*1*)
by (*simp add*: *i-def complex-of-real-def*)

lemma *i-mult-eq2* [*simp*]: *ii* ∗ *ii* = −(*1*::*complex*)

by (*simp add*: *i-def complex-one-def*)

lemma *complex-eq-cancel-iff2* [*simp*]:
(*Complex x y* = *complex-of-real xa*) = (*x* = *xa* & *y* = *0*)
by (*simp add*: *complex-of-real-def*)

lemma *Re-sgn* [*simp*]: *Re*(*sgn z*) = *Re*(*z*)/*cmod z*
by (*simp add*: *complex-sgn-def divide-inverse*)

lemma *Im-sgn* [*simp*]: *Im*(*sgn z*) = *Im*(*z*)/*cmod z*
by (*simp add*: *complex-sgn-def divide-inverse*)

lemma *complex-inverse-complex-split*:
inverse(*complex-of-real x* + *ii* ∗ *complex-of-real y*) =
complex-of-real(*x*/(*x* ^ *2* + *y* ^ *2*)) −
ii ∗ *complex-of-real*(*y*/(*x* ^ *2* + *y* ^ *2*))
by (*simp add*: *complex-of-real-def i-def diff-minus divide-inverse*)

lemma *cos-arg-i-mult-zero-pos*:
0 < *y* ==> *cos* (*arg*(*Complex 0 y*)) = *0*
apply (*simp add*: *arg-def abs-if*)
apply (*rule-tac a* = *pi*/*2* **in** *someI2*, *auto*)
apply (*rule order-less-trans* [*of - 0*], *auto*)
done

lemma *cos-arg-i-mult-zero-neg*:
y < *0* ==> *cos* (*arg*(*Complex 0 y*)) = *0*
apply (*simp add*: *arg-def abs-if*)
apply (*rule-tac a* = − *pi*/*2* **in** *someI2*, *auto*)
apply (*rule order-trans* [*of - 0*], *auto*)
done

lemma *cos-arg-i-mult-zero* [*simp*]:
y ≠ *0* ==> *cos* (*arg*(*Complex 0 y*)) = *0*
by (*auto simp add*: *linorder-neq-iff cos-arg-i-mult-zero-pos cos-arg-i-mult-zero-neg*)

64.12 Finally! Polar Form for Complex Numbers

definition

cis :: *real* => *complex* **where**
cis a = *Complex* (*cos a*) (*sin a*)

definition

rcis :: [*real*, *real*] => *complex* **where**
rcis r a = *complex-of-real r* ∗ *cis a*

definition

expi :: *complex* => *complex* **where**
expi z = *complex-of-real*(*exp* (*Re z*)) ∗ *cis* (*Im z*)

lemma *complex-split-polar*:
∃ *r a*. *z* = *complex-of-real r* ∗ (*Complex* (*cos a*) (*sin a*))
apply (*induct z*)
apply (*auto simp add*: *polar-Ex complex-of-real-mult-Complex*)
done

lemma *rcis-Ex*: ∃ *r a*. *z* = *rcis r a*
apply (*induct z*)
apply (*simp add*: *rcis-def cis-def polar-Ex complex-of-real-mult-Complex*)
done

lemma *Re-rcis* [*simp*]: *Re*(*rcis r a*) = *r* ∗ *cos a*
by (*simp add*: *rcis-def cis-def*)

lemma *Im-rcis* [*simp*]: *Im*(*rcis r a*) = *r* ∗ *sin a*
by (*simp add*: *rcis-def cis-def*)

lemma *sin-cos-squared-add2-mult*: (*r* ∗ *cos a*)^2 + (*r* ∗ *sin a*)^2 = *r*^{2}
proof −
have (*r* ∗ *cos a*)^2 + (*r* ∗ *sin a*)^2 = *r*^{2} ∗ ((*cos a*)^2 + (*sin a*)^2)
by (*simp only*: *power-mult-distrib right-distrib*)
thus *?thesis* **by** *simp*
qed

lemma *complex-mod-rcis* [*simp*]: *cmod*(*rcis r a*) = *abs r*
by (*simp add*: *rcis-def cis-def sin-cos-squared-add2-mult*)

lemma *complex-mod-sqrt-Re-mult-cnj*: *cmod z* = *sqrt* (*Re* (*z* ∗ *cnj z*))
by (*simp add*: *cmod-def power2-eq-square*)

lemma *complex-In-mult-cnj-zero* [*simp*]: *Im* (*z* ∗ *cnj z*) = *0*
by *simp*

lemma *cis-rcis-eq*: *cis a* = *rcis 1 a*

by (*simp add*: *rcis-def*)

lemma *rcis-mult*: *rcis r1 a* $*$ *rcis r2 b* = *rcis* (*r1*$*$*r2*) (*a* + *b*)
by (*simp add*: *rcis-def cis-def cos-add sin-add right-distrib right-diff-distrib*
complex-of-real-def)

lemma *cis-mult*: *cis a* $*$ *cis b* = *cis* (*a* + *b*)
by (*simp add*: *cis-rcis-eq rcis-mult*)

lemma *cis-zero* [*simp*]: *cis 0* = *1*
by (*simp add*: *cis-def complex-one-def*)

lemma *rcis-zero-mod* [*simp*]: *rcis 0 a* = *0*
by (*simp add*: *rcis-def*)

lemma *rcis-zero-arg* [*simp*]: *rcis r 0* = *complex-of-real r*
by (*simp add*: *rcis-def*)

lemma *complex-of-real-minus-one*:
complex-of-real (−(*1*::*real*)) = −(*1*::*complex*)
by (*simp add*: *complex-of-real-def complex-one-def*)

lemma *complex-i-mult-minus* [*simp*]: *ii* $*$ (*ii* $*$ *x*) = − *x*
by (*simp add*: *mult-assoc* [*symmetric*])

lemma *cis-real-of-nat-Suc-mult*:
cis (*real* (*Suc n*) $*$ *a*) = *cis a* $*$ *cis* (*real n* $*$ *a*)
by (*simp add*: *cis-def real-of-nat-Suc left-distrib cos-add sin-add right-distrib*)

lemma *DeMoivre*: (*cis a*) ^ *n* = *cis* (*real n* $*$ *a*)
apply (*induct-tac n*)
apply (*auto simp add*: *cis-real-of-nat-Suc-mult*)
done

lemma *DeMoivre2*: (*rcis r a*) ^ *n* = *rcis* (*r* ^ *n*) (*real n* $*$ *a*)
by (*simp add*: *rcis-def power-mult-distrib DeMoivre*)

lemma *cis-inverse* [*simp*]: *inverse*(*cis a*) = *cis* (−*a*)
by (*simp add*: *cis-def complex-inverse-complex-split diff-minus*)

lemma *rcis-inverse*: *inverse*(*rcis r a*) = *rcis* (*1*/*r*) (−*a*)
by (*simp add*: *divide-inverse rcis-def*)

lemma *cis-divide*: *cis a* / *cis b* = *cis* (*a* − *b*)
by (*simp add*: *complex-divide-def cis-mult real-diff-def*)

lemma *rcis-divide*: *rcis r1 a* / *rcis r2 b* = *rcis* (*r1*/*r2*) (*a* − *b*)
apply (*simp add*: *complex-divide-def*)

apply (*case-tac r2=0*, *simp*)
apply (*simp add*: *rcis-inverse rcis-mult real-diff-def*)
done

lemma *Re-cis* [*simp*]: *Re*(*cis a*) = *cos a*
by (*simp add*: *cis-def*)

lemma *Im-cis* [*simp*]: *Im*(*cis a*) = *sin a*
by (*simp add*: *cis-def*)

lemma *cos-n-Re-cis-pow-n*: *cos* (*real n* ∗ *a*) = *Re*(*cis a* ^ *n*)
by (*auto simp add*: *DeMoivre*)

lemma *sin-n-Im-cis-pow-n*: *sin* (*real n* ∗ *a*) = *Im*(*cis a* ^ *n*)
by (*auto simp add*: *DeMoivre*)

lemma *expi-add*: *expi*(*a* + *b*) = *expi*(*a*) ∗ *expi*(*b*)
by (*simp add*: *expi-def exp-add cis-mult* [*symmetric*] *mult-ac*)

lemma *expi-zero* [*simp*]: *expi* (*0*::*complex*) = *1*
by (*simp add*: *expi-def*)

lemma *complex-expi-Ex*: ∃ *a r*. *z* = *complex-of-real r* ∗ *expi a*
apply (*insert rcis-Ex* [*of z*])
apply (*auto simp add*: *expi-def rcis-def mult-assoc* [*symmetric*])
apply (*rule-tac x* = *ii* ∗ *complex-of-real a* **in** *exI*, *auto*)
done

lemma *expi-two-pi-i* [*simp*]: *expi*((*2*::*complex*) ∗ *complex-of-real pi* ∗ *ii*) = *1*
by (*simp add*: *expi-def cis-def*)

end

65 Log: Logarithms: Standard Version

theory *Log*
imports *Transcendental*
begin

definition
powr :: [*real*,*real*] => *real* (**infixr** *powr 80*) **where**
— exponentation with real exponent
x powr a = *exp*(*a* ∗ *ln x*)

definition
log :: [*real*,*real*] => *real* **where**
— logarithm of *x* to base *a*
log a x = *ln x* / *ln a*

lemma *powr-one-eq-one* [*simp*]: *1 powr a = 1*
by (*simp add*: *powr-def*)

lemma *powr-zero-eq-one* [*simp*]: *x powr 0 = 1*
by (*simp add*: *powr-def*)

lemma *powr-one-gt-zero-iff* [*simp*]: *(x powr 1 = x) = (0 < x)*
by (*simp add*: *powr-def*)
declare *powr-one-gt-zero-iff* [*THEN iffD2*, *simp*]

lemma *powr-mult*:
  [| *0 < x*; *0 < y* |] ==> *(x * y) powr a = (x powr a) * (y powr a)*
by (*simp add*: *powr-def exp-add* [*symmetric*] *ln-mult right-distrib*)

lemma *powr-gt-zero* [*simp*]: *0 < x powr a*
by (*simp add*: *powr-def*)

lemma *powr-ge-pzero* [*simp*]: *0 <= x powr y*
by (*rule order-less-imp-le*, *rule powr-gt-zero*)

lemma *powr-not-zero* [*simp*]: *x powr a ≠ 0*
by (*simp add*: *powr-def*)

lemma *powr-divide*:
  [| *0 < x*; *0 < y* |] ==> *(x / y) powr a = (x powr a)/(y powr a)*
apply (*simp add*: *divide-inverse positive-imp-inverse-positive powr-mult*)
apply (*simp add*: *powr-def exp-minus* [*symmetric*] *exp-add* [*symmetric*] *ln-inverse*)
done

lemma *powr-divide2*: *x powr a / x powr b = x powr (a - b)*
 apply (*simp add*: *powr-def*)
 apply (*subst exp-diff* [*THEN sym*])
 apply (*simp add*: *left-diff-distrib*)
done

lemma *powr-add*: *x powr (a + b) = (x powr a) * (x powr b)*
by (*simp add*: *powr-def exp-add* [*symmetric*] *left-distrib*)

lemma *powr-powr*: *(x powr a) powr b = x powr (a * b)*
by (*simp add*: *powr-def*)

lemma *powr-powr-swap*: *(x powr a) powr b = (x powr b) powr a*
by (*simp add*: *powr-powr real-mult-commute*)

lemma *powr-minus*: *x powr (-a) = inverse (x powr a)*
by (*simp add*: *powr-def exp-minus* [*symmetric*])

lemma *powr-minus-divide*: $x\ powr\ (-a) = 1/(x\ powr\ a)$
by (*simp add*: *divide-inverse powr-minus*)

lemma *powr-less-mono*: $[|\ a < b;\ 1 < x\ |] ==> x\ powr\ a < x\ powr\ b$
by (*simp add*: *powr-def*)

lemma *powr-less-cancel*: $[|\ x\ powr\ a < x\ powr\ b;\ 1 < x\ |] ==> a < b$
by (*simp add*: *powr-def*)

lemma *powr-less-cancel-iff* [*simp*]: $1 < x ==> (x\ powr\ a < x\ powr\ b) = (a < b)$
by (*blast intro*: *powr-less-cancel powr-less-mono*)

lemma *powr-le-cancel-iff* [*simp*]: $1 < x ==> (x\ powr\ a \leq x\ powr\ b) = (a \leq b)$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *log-ln*: $ln\ x = log\ (exp(1))\ x$
by (*simp add*: *log-def*)

lemma *powr-log-cancel* [*simp*]:
$[|\ 0 < a;\ a \neq 1;\ 0 < x\ |] ==> a\ powr\ (log\ a\ x) = x$
by (*simp add*: *powr-def log-def*)

lemma *log-powr-cancel* [*simp*]: $[|\ 0 < a;\ a \neq 1\ |] ==> log\ a\ (a\ powr\ y) = y$
by (*simp add*: *log-def powr-def*)

lemma *log-mult*:
$[|\ 0 < a;\ a \neq 1;\ 0 < x;\ 0 < y\ |]$
$==> log\ a\ (x * y) = log\ a\ x + log\ a\ y$
by (*simp add*: *log-def ln-mult divide-inverse left-distrib*)

lemma *log-eq-div-ln-mult-log*:
$[|\ 0 < a;\ a \neq 1;\ 0 < b;\ b \neq 1;\ 0 < x\ |]$
$==> log\ a\ x = (ln\ b/ln\ a) * log\ b\ x$
by (*simp add*: *log-def divide-inverse*)

Base 10 logarithms

lemma *log-base-10-eq1*: $0 < x ==> log\ 10\ x = (ln\ (exp\ 1)\ /\ ln\ 10) * ln\ x$
by (*simp add*: *log-def*)

lemma *log-base-10-eq2*: $0 < x ==> log\ 10\ x = (log\ 10\ (exp\ 1)) * ln\ x$
by (*simp add*: *log-def*)

lemma *log-one* [*simp*]: $log\ a\ 1 = 0$
by (*simp add*: *log-def*)

lemma *log-eq-one* [*simp*]: $[|\ 0 < a;\ a \neq 1\ |] ==> log\ a\ a = 1$
by (*simp add*: *log-def*)

lemma *log-inverse*:
[| $0 < a$; $a \neq 1$; $0 < x$ |] ==> $log\ a\ (inverse\ x) = -\ log\ a\ x$
apply (*rule-tac a1* = $log\ a\ x$ **in** *add-left-cancel* [*THEN iffD1*])
apply (*simp add*: *log-mult* [*symmetric*])
done

lemma *log-divide*:
[|$0 < a$; $a \neq 1$; $0 < x$; $0 < y$|] ==> $log\ a\ (x/y) = log\ a\ x - log\ a\ y$
by (*simp add*: *log-mult divide-inverse log-inverse*)

lemma *log-less-cancel-iff* [*simp*]:
[| $1 < a$; $0 < x$; $0 < y$ |] ==> $(log\ a\ x < log\ a\ y) = (x < y)$
apply *safe*
apply (*rule-tac* [*2*] *powr-less-cancel*)
apply (*drule-tac a* = $log\ a\ x$ **in** *powr-less-mono*, *auto*)
done

lemma *log-le-cancel-iff* [*simp*]:
[| $1 < a$; $0 < x$; $0 < y$ |] ==> $(log\ a\ x \leq log\ a\ y) = (x \leq y)$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *powr-realpow*: $0 < x$ ==> $x\ powr\ (real\ n) = x\hat{}n$
apply (*induct n*, *simp*)
apply (*subgoal-tac real*$(Suc\ n) = real\ n + 1$)
apply (*erule ssubst*)
apply (*subst powr-add*, *simp*, *simp*)
done

lemma *powr-realpow2*: $0 <= x$ ==> $0 < n$ ==> $x\hat{}n = (if\ (x = 0)\ then\ 0$
else x powr $(real\ n)$)
apply (*case-tac x* = *0*, *simp*, *simp*)
apply (*rule powr-realpow* [*THEN sym*], *simp*)
done

lemma *ln-pwr*: $0 < x$ ==> $0 < y$ ==> $ln(x\ powr\ y) = y * ln\ x$
by (*unfold powr-def*, *simp*)

lemma *ln-bound*: $1 <= x$ ==> $ln\ x <= x$
apply (*subgoal-tac* $ln(1 + (x - 1)) <= x - 1$)
apply *simp*
apply (*rule ln-add-one-self-le-self*, *simp*)
done

lemma *powr-mono*: $a <= b$ ==> $1 <= x$ ==> $x\ powr\ a <= x\ powr\ b$
apply (*case-tac x* = *1*, *simp*)
apply (*case-tac a* = *b*, *simp*)
apply (*rule order-less-imp-le*)
apply (*rule powr-less-mono*, *auto*)

done

lemma *ge-one-powr-ge-zero*: *1 <= x ==> 0 <= a ==> 1 <= x powr a*
apply (*subst powr-zero-eq-one* [*THEN sym*])
apply (*rule powr-mono*, *assumption+*)
done

lemma *powr-less-mono2*: *0 < a ==> 0 < x ==> x < y ==> x powr a < y powr a*
apply (*unfold powr-def*)
apply (*rule exp-less-mono*)
apply (*rule mult-strict-left-mono*)
apply (*subst ln-less-cancel-iff*, *assumption*)
apply (*rule order-less-trans*)
prefer *2*
apply *assumption+*
done

lemma *powr-less-mono2-neg*: *a < 0 ==> 0 < x ==> x < y ==> y powr a < x powr a*
apply (*unfold powr-def*)
apply (*rule exp-less-mono*)
apply (*rule mult-strict-left-mono-neg*)
apply (*subst ln-less-cancel-iff*)
apply *assumption*
apply (*rule order-less-trans*)
prefer *2*
apply *assumption+*
done

lemma *powr-mono2*: *0 <= a ==> 0 < x ==> x <= y ==> x powr a <= y powr a*
apply (*case-tac a = 0*, *simp*)
apply (*case-tac x = y*, *simp*)
apply (*rule order-less-imp-le*)
apply (*rule powr-less-mono2*, *auto*)
done

lemma *ln-powr-bound*: *1 <= x ==> 0 < a ==> ln x <= (x powr a) / a*
apply (*rule mult-imp-le-div-pos*)
apply (*assumption*)
apply (*subst mult-commute*)
apply (*subst ln-pwr* [*THEN sym*])
apply *auto*
apply (*rule ln-bound*)
apply (*erule ge-one-powr-ge-zero*)
apply (*erule order-less-imp-le*)
done

lemma *ln-powr-bound2*: *1 < x ==> 0 < a ==> (ln x) powr a <= (a powr a) * x*
proof −
assume *1 < x* **and** *0 < a*
then have *ln x <= (x powr (1 / a)) / (1 / a)*
apply (*intro ln-powr-bound*)
apply (*erule order-less-imp-le*)
apply (*rule divide-pos-pos*)
apply *simp-all*
done
also have *... = a * (x powr (1 / a))*
by *simp*
finally have *(ln x) powr a <= (a * (x powr (1 / a))) powr a*
apply (*intro powr-mono2*)
apply (*rule order-less-imp-le*, *rule prems*)
apply (*rule ln-gt-zero*)
apply (*rule prems*)
apply *assumption*
done
also have *... = (a powr a) * ((x powr (1 / a)) powr a)*
apply (*rule powr-mult*)
apply (*rule prems*)
apply (*rule powr-gt-zero*)
done
also have *(x powr (1 / a)) powr a = x powr ((1 / a) * a)*
by (*rule powr-powr*)
also have *... = x*
apply *simp*
apply (*subgoal-tac a ~= 0*)
apply (*insert prems*, *auto*)
done
finally show *?thesis* **.**
qed

lemma *LIMSEQ-neg-powr*: *0 < s ==> (%x. (real x) powr − s) −−−−> 0*
apply (*unfold LIMSEQ-def*)
apply *clarsimp*
apply (*rule-tac x = natfloor(r powr (1 / − s)) + 1* **in** *exI*)
apply *clarify*
proof −
fix *r* **fix** *n*
assume *0 < s* **and** *0 < r* **and** *natfloor (r powr (1 / − s)) + 1 <= n*
have *r powr (1 / − s) < real(natfloor(r powr (1 / − s))) + 1*
by (*rule real-natfloor-add-one-gt*)
also have *... = real(natfloor(r powr (1 / −s)) + 1)*
by *simp*
also have *... <= real n*
apply (*subst real-of-nat-le-iff*)
apply (*rule prems*)

```
    done
  finally have r powr (1 / − s) < real n.
  then have real n powr (− s) < (r powr (1 / − s)) powr − s
    apply (intro powr-less-mono2-neg)
    apply (auto simp add: prems)
    done
  also have ... = r
    by (simp add: powr-powr prems less-imp-neq [THEN not-sym])
  finally show real n powr − s < r .
 qed

end
```

66 Ln: Properties of ln

```
theory Ln
imports Transcendental
begin

lemma exp-first-two-terms: exp x = 1 + x + suminf (%n.
  inverse(real (fact (n+2))) * (x ^ (n+2)))
proof −
  have exp x = suminf (%n. inverse(real (fact n)) * (x ^ n))
    by (simp add: exp-def)
  also from summable-exp have ... = (SUM n : {0..<2}.
      inverse(real (fact n)) * (x ^ n)) + suminf (%n.
      inverse(real (fact (n+2))) * (x ^ (n+2))) (is - = ?a + -)
    by (rule suminf-split-initial-segment)
  also have ?a = 1 + x
    by (simp add: numerals)
  finally show ?thesis .
qed

lemma exp-tail-after-first-two-terms-summable:
  summable (%n. inverse(real (fact (n+2))) * (x ^ (n+2)))
proof −
  note summable-exp
  thus ?thesis
    by (frule summable-ignore-initial-segment)
qed

lemma aux1: assumes a: 0 <= x and b: x <= 1
    shows inverse (real (fact (n + 2))) * x ^ (n + 2) <= (x^2/2) * ((1/2)^n)
proof (induct n)
  show inverse (real (fact (0 + 2))) * x ^ (0 + 2) <=
      x ^ 2 / 2 * (1 / 2) ^ 0
    by (simp add: real-of-nat-Suc power2-eq-square)
next
```

fix *n*
assume *c*: *inverse* (*real* (*fact* (*n* + *2*))) * *x* ^ (*n* + *2*)
    <= *x* ^ *2* / *2* * (*1* / *2*) ^ *n*
show *inverse* (*real* (*fact* (*Suc n* + *2*))) * *x* ^ (*Suc n* + *2*)
      <= *x* ^ *2* / *2* * (*1* / *2*) ^ *Suc n*
proof −
 have *inverse*(*real* (*fact* (*Suc n* + *2*))) <=
    (*1* / *2*) **inverse* (*real* (*fact* (*n*+*2*)))
 proof −
  have *Suc n* + *2* = *Suc* (*n* + *2*) **by** *simp*
  then have *fact* (*Suc n* + *2*) = *Suc* (*n* + *2*) * *fact* (*n* + *2*)
   by *simp*
  then have *real*(*fact* (*Suc n* + *2*)) = *real*(*Suc* (*n* + *2*) * *fact* (*n* + *2*))
   apply (*rule subst*)
   apply (*rule refl*)
   done
  also have ... = *real*(*Suc* (*n* + *2*)) * *real*(*fact* (*n* + *2*))
   by (*rule real-of-nat-mult*)
  finally have *real* (*fact* (*Suc n* + *2*)) =
    real (*Suc* (*n* + *2*)) * *real* (*fact* (*n* + *2*)) **.**
  then have *inverse*(*real* (*fact* (*Suc n* + *2*))) =
    inverse(*real* (*Suc* (*n* + *2*))) * *inverse*(*real* (*fact* (*n* + *2*)))
   apply (*rule ssubst*)
   apply (*rule inverse-mult-distrib*)
   done
  also have ... <= (*1*/*2*) * *inverse*(*real* (*fact* (*n* + *2*)))
   apply (*rule mult-right-mono*)
   apply (*subst inverse-eq-divide*)
   apply *simp*
   apply (*rule inv-real-of-nat-fact-ge-zero*)
   done
  finally show *?thesis* **.**
 qed
 moreover have *x* ^ (*Suc n* + *2*) <= *x* ^ (*n* + *2*)
  apply (*simp add*: *mult-compare-simps*)
  apply (*simp add*: *prems*)
  apply (*subgoal-tac 0* <= *x* * (*x* * *x*^*n*))
  apply *force*
  apply (*rule mult-nonneg-nonneg*, *rule a*)+
  apply (*rule zero-le-power*, *rule a*)
  done
 ultimately have *inverse* (*real* (*fact* (*Suc n* + *2*))) * *x* ^ (*Suc n* + *2*) <=
    (*1* / *2* * *inverse* (*real* (*fact* (*n* + *2*)))) * *x* ^ (*n* + *2*)
  apply (*rule mult-mono*)
  apply (*rule mult-nonneg-nonneg*)
  apply *simp*
  apply (*subst inverse-nonnegative-iff-nonnegative*)
  apply (*rule real-of-nat-ge-zero*)
  apply (*rule zero-le-power*)

```
    apply (rule a)
    done
  also have ... = 1 / 2 * (inverse (real (fact (n + 2))) * x ^ (n + 2))
    by simp
  also have ... <= 1 / 2 * (x ^ 2 / 2 * (1 / 2) ^ n)
    apply (rule mult-left-mono)
    apply (rule prems)
    apply simp
    done
  also have ... = x ^ 2 / 2 * (1 / 2 * (1 / 2) ^ n)
    by auto
  also have (1::real) / 2 * (1 / 2) ^ n = (1 / 2) ^ (Suc n)
    by (rule power-Suc [THEN sym])
  finally show ?thesis .
 qed
qed

lemma aux2: (%n. (x::real) ^ 2 / 2 * (1 / 2) ^ n) sums x^2
proof -
 have (%n. (1 / 2::real)^n) sums (1 / (1 - (1/2)))
   apply (rule geometric-sums)
   by (simp add: abs-less-iff)
 also have (1::real) / (1 - 1/2) = 2
   by simp
 finally have (%n. (1 / 2::real)^n) sums 2 .
 then have (%n. x ^ 2 / 2 * (1 / 2) ^ n) sums (x^2 / 2 * 2)
   by (rule sums-mult)
 also have x^2 / 2 * 2 = x^2
   by simp
 finally show ?thesis .
qed

lemma exp-bound: 0 <= (x::real) ==> x <= 1 ==> exp x <= 1 + x + x^2
proof -
 assume a: 0 <= x
 assume b: x <= 1
 have c: exp x = 1 + x + suminf (%n. inverse(real (fact (n+2))) *
    (x ^ (n+2)))
   by (rule exp-first-two-terms)
 moreover have suminf (%n. inverse(real (fact (n+2))) * (x ^ (n+2))) <= x^2
 proof -
   have suminf (%n. inverse(real (fact (n+2))) * (x ^ (n+2))) <=
      suminf (%n. (x^2/2) * ((1/2)^n))
     apply (rule summable-le)
     apply (auto simp only: aux1 prems)
     apply (rule exp-tail-after-first-two-terms-summable)
     by (rule sums-summable, rule aux2)
   also have ... = x^2
     by (rule sums-unique [THEN sym], rule aux2)
```

```
    finally show ?thesis .
  qed
  ultimately show ?thesis
    by auto
qed

lemma aux4: 0 <= (x::real) ==> x <= 1 ==> exp (x - x^2) <= 1 + x
proof -
  assume a: 0 <= x and b: x <= 1
  have exp (x - x^2) = exp x / exp (x^2)
    by (rule exp-diff)
  also have ... <= (1 + x + x^2) / exp (x ^2)
    apply (rule divide-right-mono)
    apply (rule exp-bound)
    apply (rule a, rule b)
    apply simp
    done
  also have ... <= (1 + x + x^2) / (1 + x^2)
    apply (rule divide-left-mono)
    apply (auto simp add: exp-ge-add-one-self-aux)
    apply (rule add-nonneg-nonneg)
    apply (insert prems, auto)
    apply (rule mult-pos-pos)
    apply auto
    apply (rule add-pos-nonneg)
    apply auto
    done
  also from a have ... <= 1 + x
    by(simp add:field-simps zero-compare-simps)
  finally show ?thesis .
qed

lemma ln-one-plus-pos-lower-bound: 0 <= x ==> x <= 1 ==>
    x - x^2 <= ln (1 + x)
proof -
  assume a: 0 <= x and b: x <= 1
  then have exp (x - x^2) <= 1 + x
    by (rule aux4)
  also have ... = exp (ln (1 + x))
  proof -
    from a have 0 < 1 + x by auto
    thus ?thesis
      by (auto simp only: exp-ln-iff [THEN sym])
  qed
  finally have exp (x - x ^ 2) <= exp (ln (1 + x)) .
  thus ?thesis by (auto simp only: exp-le-cancel-iff)
qed

lemma ln-one-minus-pos-upper-bound: 0 <= x ==> x < 1 ==> ln (1 - x) <=
```

```
- x
proof -
  assume a: 0 <= (x::real) and b: x < 1
  have (1 - x) * (1 + x + x^2) = (1 - x^3)
    by (simp add: algebra-simps power2-eq-square power3-eq-cube)
  also have ... <= 1
    by (auto simp add: a)
  finally have (1 - x) * (1 + x + x ^ 2) <= 1 .
  moreover have 0 < 1 + x + x^2
    apply (rule add-pos-nonneg)
    apply (insert a, auto)
    done
  ultimately have 1 - x <= 1 / (1 + x + x^2)
    by (elim mult-imp-le-div-pos)
  also have ... <= 1 / exp x
    apply (rule divide-left-mono)
    apply (rule exp-bound, rule a)
    apply (insert prems, auto)
    apply (rule mult-pos-pos)
    apply (rule add-pos-nonneg)
    apply auto
    done
  also have ... = exp (-x)
    by (auto simp add: exp-minus real-divide-def)
  finally have 1 - x <= exp (- x) .
  also have 1 - x = exp (ln (1 - x))
  proof -
    have 0 < 1 - x
      by (insert b, auto)
    thus ?thesis
      by (auto simp only: exp-ln-iff [THEN sym])
  qed
  finally have exp (ln (1 - x)) <= exp (- x) .
  thus ?thesis by (auto simp only: exp-le-cancel-iff)
qed

lemma aux5: x < 1 ==> ln(1 - x) = - ln(1 + x / (1 - x))
proof -
  assume a: x < 1
  have ln(1 - x) = - ln(1 / (1 - x))
  proof -
    have ln(1 - x) = - (- ln (1 - x))
      by auto
    also have - ln(1 - x) = ln 1 - ln(1 - x)
      by simp
    also have ... = ln(1 / (1 - x))
      apply (rule ln-div [THEN sym])
      by (insert a, auto)
    finally show ?thesis .
```

```
  qed
  also have  1 / (1 - x) = 1 + x / (1 - x) using a by(simp add:field-simps)
  finally show ?thesis .
qed

lemma ln-one-minus-pos-lower-bound: 0 <= x ==> x <= (1 / 2) ==>
    - x - 2 * x^2 <= ln (1 - x)
proof -
  assume a: 0 <= x and b: x <= (1 / 2)
  from b have c: x < 1
    by auto
  then have ln (1 - x) = - ln (1 + x / (1 - x))
    by (rule aux5)
  also have - (x / (1 - x)) <= ...
  proof -
    have ln (1 + x / (1 - x)) <= x / (1 - x)
      apply (rule ln-add-one-self-le-self)
      apply (rule divide-nonneg-pos)
      by (insert a c, auto)
    thus ?thesis
      by auto
  qed
  also have - (x / (1 - x)) = -x / (1 - x)
    by auto
  finally have d: - x / (1 - x) <= ln (1 - x) .
  have 0 < 1 - x using prems by simp
  hence e: -x - 2 * x^2 <= - x / (1 - x)
    using mult-right-le-one-le[of x*x 2*x] prems
    by(simp add:field-simps power2-eq-square)
  from e d show - x - 2 * x^2 <= ln (1 - x)
    by (rule order-trans)
qed

lemma exp-ge-add-one-self [simp]: 1 + (x::real) <= exp x
  apply (case-tac 0 <= x)
  apply (erule exp-ge-add-one-self-aux)
  apply (case-tac x <= -1)
  apply (subgoal-tac 1 + x <= 0)
  apply (erule order-trans)
  apply simp
  apply simp
  apply (subgoal-tac 1 + x = exp(ln (1 + x)))
  apply (erule ssubst)
  apply (subst exp-le-cancel-iff)
  apply (subgoal-tac ln (1 - (- x)) <= - (- x))
  apply simp
  apply (rule ln-one-minus-pos-upper-bound)
  apply auto
done
```

lemma *ln-add-one-self-le-self2*: $-1 < x \Longrightarrow ln(1 + x) <= x$
apply (*subgoal-tac* $x = ln\ (exp\ x)$)
apply (*erule ssubst*)**back**
apply (*subst ln-le-cancel-iff*)
apply *auto*
done

lemma *abs-ln-one-plus-x-minus-x-bound-nonneg*:
$0 <= x \Longrightarrow x <= 1 \Longrightarrow abs(ln\ (1 + x) - x) <= x\hat{}2$
proof −
assume *x*: $0 <= x$
assume $x <= 1$
from *x* **have** $ln\ (1 + x) <= x$
by (*rule ln-add-one-self-le-self*)
then have $ln\ (1 + x) - x <= 0$
by *simp*
then have $abs(ln(1 + x) - x) = -\ (ln(1 + x) - x)$
by (*rule abs-of-nonpos*)
also have $... = x - ln\ (1 + x)$
by *simp*
also have $... <= x\hat{}2$
proof −
from *prems* **have** $x - x\hat{}2 <= ln\ (1 + x)$
by (*intro ln-one-plus-pos-lower-bound*)
thus *?thesis*
by *simp*
qed
finally show *?thesis* **.**
qed

lemma *abs-ln-one-plus-x-minus-x-bound-nonpos*:
$-(1\ /\ 2) <= x \Longrightarrow x <= 0 \Longrightarrow abs(ln\ (1 + x) - x) <= 2 * x\hat{}2$
proof −
assume $-(1\ /\ 2) <= x$
assume $x <= 0$
have $abs(ln\ (1 + x) - x) = x - ln(1 - (-x))$
apply (*subst abs-of-nonpos*)
apply *simp*
apply (*rule ln-add-one-self-le-self2*)
apply (*insert prems*, *auto*)
done
also have $... <= 2 * x\hat{}2$
apply (*subgoal-tac* $- (-x) - 2 * (-x)\hat{}2 <= ln\ (1 - (-x))$)
apply (*simp add*: *algebra-simps*)
apply (*rule ln-one-minus-pos-lower-bound*)
apply (*insert prems*, *auto*)
done
finally show *?thesis* **.**

qed

lemma *abs-ln-one-plus-x-minus-x-bound*:
  *abs x <= 1 / 2 ==> abs(ln (1 + x) − x) <= 2 * x^2*
 apply (*case-tac 0 <= x*)
 apply (*rule order-trans*)
 apply (*rule abs-ln-one-plus-x-minus-x-bound-nonneg*)
 apply *auto*
 apply (*rule abs-ln-one-plus-x-minus-x-bound-nonpos*)
 apply *auto*
done

lemma *DERIV-ln*: *0 < x ==> DERIV ln x :> 1 / x*
 apply (*unfold deriv-def*, *unfold LIM-def*, *clarsimp*)
 apply (*rule exI*)
 apply (*rule conjI*)
 prefer *2*
 apply *clarsimp*
 apply (*subgoal-tac* (*ln* (*x* + *xa*) − *ln x*) / *xa* − (*1* / *x*) =
   (*ln* (*1* + *xa* / *x*) − *xa* / *x*) / *xa*)
 apply (*erule ssubst*)
 apply (*subst abs-divide*)
 apply (*rule mult-imp-div-pos-less*)
 apply *force*
 apply (*rule order-le-less-trans*)
 apply (*rule abs-ln-one-plus-x-minus-x-bound*)
 apply (*subst abs-divide*)
 apply (*subst abs-of-pos*, *assumption*)
 apply (*erule mult-imp-div-pos-le*)
 apply (*subgoal-tac abs xa < min* (*x* / *2*) (*r* * *x^2* / *2*))
 apply *force*
 apply *assumption*
 apply (*simp add*: *power2-eq-square mult-compare-simps*)
 apply (*rule mult-imp-div-pos-less*)
 apply (*rule mult-pos-pos*, *assumption*, *assumption*)
 apply (*subgoal-tac xa* * *xa* = *abs xa* * *abs xa*)
 apply (*erule ssubst*)
 apply (*subgoal-tac abs xa* * (*abs xa* * *2*) < *abs xa* * (*r* * (*x* * *x*)))
 apply (*simp only*: *mult-ac*)
 apply (*rule mult-strict-left-mono*)
 apply (*erule conjE*, *assumption*)
 apply *force*
 apply *simp*
 apply (*subst ln-div* [*THEN sym*])
 apply *arith*
 apply (*auto simp add*: *algebra-simps add-frac-eq frac-eq-eq*
  add-divide-distrib power2-eq-square)
 apply (*rule mult-pos-pos*, *assumption*)+
 apply *assumption*

done

lemma *ln-x-over-x-mono*: *exp 1 <= x ==> x <= y ==> (ln y / y) <= (ln x / x)*
proof −
  assume *exp 1 <= x* **and** *x <= y*
  have *a*: *0 < x* **and** *b*: *0 < y*
    apply (*insert prems*)
    apply (*subgoal-tac 0 < exp (1::real)*)
    apply *arith*
    apply *auto*
    apply (*subgoal-tac 0 < exp (1::real)*)
    apply *arith*
    apply *auto*
    done
  have *x * ln y − x * ln x = x * (ln y − ln x)*
    by (*simp add*: *algebra-simps*)
  also have *... = x * ln(y / x)*
    apply (*subst ln-div*)
    apply (*rule b*, *rule a*, *rule refl*)
    done
  also have *y / x = (x + (y − x)) / x*
    by *simp*
  also have *... = 1 + (y − x) / x* **using** *a prems* **by**(*simp add*:*field-simps*)
  also have *x * ln(1 + (y − x) / x) <= x * ((y − x) / x)*
    apply (*rule mult-left-mono*)
    apply (*rule ln-add-one-self-le-self*)
    apply (*rule divide-nonneg-pos*)
    apply (*insert prems a*, *simp-all*)
    done
  also have *... = y − x* **using** *a* **by** *simp*
  also have *... = (y − x) * ln (exp 1)* **by** *simp*
  also have *... <= (y − x) * ln x*
    apply (*rule mult-left-mono*)
    apply (*subst ln-le-cancel-iff*)
    apply *force*
    apply (*rule a*)
    apply (*rule prems*)
    apply (*insert prems*, *simp*)
    done
  also have *... = y * ln x − x * ln x*
    by (*rule left-diff-distrib*)
  finally have *x * ln y <= y * ln x*
    by *arith*
  then have *ln y <= (y * ln x) / x* **using** *a* **by**(*simp add*:*field-simps*)
  also have *... = y * (ln x / x)* **by** *simp*
  finally show *?thesis* **using** *b* **by**(*simp add*:*field-simps*)
qed

end

67 MacLaurin: MacLaurin Series

theory *MacLaurin*
imports *Transcendental*
begin

67.1 Maclaurin's Theorem with Lagrange Form of Remainder

This is a very long, messy proof even now that it's been broken down into lemmas.

lemma *Maclaurin-lemma*:
$0 < h \Longrightarrow$
$\exists B.\ f\ h = (\sum m{=}0..{<}n.\ (j\ m\ /\ real\ (fact\ m)) * (h\hat{}m)) +$
$(B * ((h\hat{}n)\ /\ real(fact\ n)))$
apply (*rule-tac x* = $(f\ h - (\sum m{=}0..{<}n.\ (j\ m\ /\ real\ (fact\ m)) * h\hat{}m)) *$
$real(fact\ n)\ /\ (h\hat{}n)$
in *exI*)
apply (*simp*)
done

lemma *eq-diff-eq'*: $(x = y - z) = (y = x + (z{::}real))$
by *arith*

A crude tactic to differentiate by proof.

lemmas *deriv-rulesI* =
DERIV-ident DERIV-const DERIV-cos DERIV-cmult
DERIV-sin DERIV-exp DERIV-inverse DERIV-pow
DERIV-add DERIV-diff DERIV-mult DERIV-minus
DERIV-inverse-fun DERIV-quotient DERIV-fun-pow
DERIV-fun-exp DERIV-fun-sin DERIV-fun-cos
DERIV-ident DERIV-const DERIV-cos

ML
⟪
local
exception DERIV-name;
fun get-fun-name (- $ (*Const* (*Lim.deriv*,-) $ *Abs*(-,-, *Const* (*f*,-) $ -) $ - $ -)) = *f*
| *get-fun-name* (- $ (- $ (*Const* (*Lim.deriv*,-) $ *Abs*(-,-, *Const* (*f*,-) $ -) $ - $ -)))
= *f*
| *get-fun-name* - = *raise DERIV-name*;

in

fun deriv-tac ctxt = *SUBGOAL* (*fn* (*prem*, *i*) =>

```
  resolve-tac @{thms deriv-rulesI} i ORELSE
    ((rtac (read-instantiate ctxt [(f, 0), get-fun-name prem)]
                  @{thm DERIV-chain2}) i) handle DERIV-name => no-tac));

fun DERIV-tac ctxt = ALLGOALS (fn i => REPEAT (deriv-tac ctxt i));

end
⟩⟩

lemma Maclaurin-lemma2:
  assumes diff: ∀ m t. m < n ∧ 0≤t ∧ t≤h ⟶ DERIV (diff m) t :> diff (Suc
m) t
  assumes n: n = Suc k
  assumes difg: difg =
        (λm t. diff m t −
              ((∑ p = 0..<n − m. diff (m + p) 0 / real (fact p) * t ^ p) +
               B * (t ^ (n − m) / real (fact (n − m)))))
  shows
      ∀ m t. m < n & 0 ≤ t & t ≤ h −−> DERIV (difg m) t :> difg (Suc m) t
unfolding difg
 apply clarify
 apply (rule DERIV-diff)
  apply (simp add: diff)
 apply (simp only: n)
 apply (rule DERIV-add)
  apply (rule-tac [2] DERIV-cmult)
  apply (rule-tac [2] lemma-DERIV-subst)
   apply (rule-tac [2] DERIV-quotient)
     apply (rule-tac [3] DERIV-const)
    apply (rule-tac [2] DERIV-pow)
   prefer 3 apply (simp add: fact-diff-Suc)
  prefer 2 apply simp
 apply (frule less-iff-Suc-add [THEN iffD1], clarify)
 apply (simp del: setsum-op-ivl-Suc)
 apply (insert sumr-offset4 [of Suc 0])
 apply (simp del: setsum-op-ivl-Suc fact-Suc power-Suc)
 apply (rule lemma-DERIV-subst)
 apply (rule DERIV-add)
  apply (rule-tac [2] DERIV-const)
 apply (rule DERIV-sumr, clarify)
 prefer 2 apply simp
 apply (simp (no-asm) add: divide-inverse mult-assoc del: fact-Suc power-Suc)
 apply (rule DERIV-cmult)
 apply (rule lemma-DERIV-subst)
  apply (best intro: DERIV-chain2 intro!: DERIV-intros)
 apply (subst fact-Suc)
 apply (subst real-of-nat-mult)
 apply (simp add: mult-ac)
done
```

lemma *Maclaurin*:
assumes *h*: $0 < h$
assumes *n*: $0 < n$
assumes *diff-0*: $diff\ 0 = f$
assumes *diff-Suc*:
$\forall m\ t.\ m < n\ \&\ 0 \leq t\ \&\ t \leq h \longrightarrow DERIV\ (diff\ m)\ t :> diff\ (Suc\ m)\ t$
shows
$\exists t.\ 0 < t\ \&\ t < h\ \&$
$f\ h =$
$setsum\ (\%m.\ (diff\ m\ 0\ /\ real\ (fact\ m)) * h\ \hat{}\ m)\ \{0..<n\} +$
$(diff\ n\ t\ /\ real\ (fact\ n)) * h\ \hat{}\ n$
proof −
from *n* **obtain** *m* **where** *m*: $n = Suc\ m$
by (*cases n*, *simp add*: *n*)

obtain *B* **where** *f-h*: $f\ h =$
$(\sum m = 0..<n.\ diff\ m\ (0::real)\ /\ real\ (fact\ m) * h\ \hat{}\ m) +$
$B * (h\ \hat{}\ n\ /\ real\ (fact\ n))$
using *Maclaurin-lemma* [*OF h*] **..**

obtain *g* **where** *g-def*: $g = (\%t.\ f\ t -$
$(setsum\ (\%m.\ (diff\ m\ 0\ /\ real(fact\ m)) * t\hat{}m)\ \{0..<n\}$
$+ (B * (t\hat{}n\ /\ real(fact\ n)))))$ **by** *blast*

have *g2*: $g\ 0 = 0\ \&\ g\ h = 0$
apply (*simp add*: *m f-h g-def del*: *setsum-op-ivl-Suc*)
apply (*cut-tac n* = *m* **and** *k* = *Suc 0* **in** *sumr-offset2*)
apply (*simp add*: *eq-diff-eq' diff-0 del*: *setsum-op-ivl-Suc*)
done

obtain *difg* **where** *difg-def*: $difg = (\%m\ t.\ diff\ m\ t -$
$(setsum\ (\%p.\ (diff\ (m + p)\ 0\ /\ real\ (fact\ p)) * (t\ \hat{}\ p))\ \{0..<n-m\}$
$+ (B * ((t\ \hat{}\ (n - m))\ /\ real\ (fact\ (n - m))))))$ **by** *blast*

have *difg-0*: $difg\ 0 = g$
unfolding *difg-def g-def* **by** (*simp add*: *diff-0*)

have *difg-Suc*: $\forall (m::nat)\ t::real.$
$m < n \wedge (0::real) \leq t \wedge t \leq h \longrightarrow DERIV\ (difg\ m)\ t :> difg\ (Suc\ m)\ t$
using *diff-Suc m difg-def* **by** (*rule Maclaurin-lemma2*)

have *difg-eq-0*: $\forall m.\ m < n \longrightarrow difg\ m\ 0 = 0$
apply *clarify*
apply (*simp add*: *m difg-def*)
apply (*frule less-iff-Suc-add* [*THEN iffD1*], *clarify*)
apply (*simp del*: *setsum-op-ivl-Suc*)
apply (*insert sumr-offset4* [*of Suc 0*])

```
  apply (simp del: setsum-op-ivl-Suc fact-Suc)
  done

have isCont-difg: ⋀m x. ⟦m < n; 0 ≤ x; x ≤ h⟧ ⟹ isCont (difg m) x
  by (rule DERIV-isCont [OF difg-Suc [rule-format]]) simp

have differentiable-difg:
  ⋀m x. ⟦m < n; 0 ≤ x; x ≤ h⟧ ⟹ difg m differentiable x
  by (rule differentiableI [OF difg-Suc [rule-format]]) simp

have difg-Suc-eq-0: ⋀m t. ⟦m < n; 0 ≤ t; t ≤ h; DERIV (difg m) t :> 0⟧
    ⟹ difg (Suc m) t = 0
  by (rule DERIV-unique [OF difg-Suc [rule-format]]) simp

have m < n using m by simp

have ∃t. 0 < t ∧ t < h ∧ DERIV (difg m) t :> 0
using ‹m < n›
proof (induct m)
case 0
  show ?case
  proof (rule Rolle)
    show 0 < h by fact
    show difg 0 0 = difg 0 h by (simp add: difg-0 g2)
    show ∀x. 0 ≤ x ∧ x ≤ h ⟶ isCont (difg (0::nat)) x
      by (simp add: isCont-difg n)
    show ∀x. 0 < x ∧ x < h ⟶ difg (0::nat) differentiable x
      by (simp add: differentiable-difg n)
  qed
next
case (Suc m')
  hence ∃t. 0 < t ∧ t < h ∧ DERIV (difg m') t :> 0 by simp
  then obtain t where t: 0 < t t < h DERIV (difg m') t :> 0 by fast
  have ∃t'. 0 < t' ∧ t' < t ∧ DERIV (difg (Suc m')) t' :> 0
  proof (rule Rolle)
    show 0 < t by fact
    show difg (Suc m') 0 = difg (Suc m') t
      using t ‹Suc m' < n› by (simp add: difg-Suc-eq-0 difg-eq-0)
    show ∀x. 0 ≤ x ∧ x ≤ t ⟶ isCont (difg (Suc m')) x
      using ‹t < h› ‹Suc m' < n› by (simp add: isCont-difg)
    show ∀x. 0 < x ∧ x < t ⟶ difg (Suc m') differentiable x
      using ‹t < h› ‹Suc m' < n› by (simp add: differentiable-difg)
  qed
  thus ?case
    using ‹t < h› by auto
qed

then obtain t where 0 < t t < h DERIV (difg m) t :> 0 by fast
```

```
  hence difg (Suc m) t = 0
    using ‹m < n› by (simp add: difg-Suc-eq-0)

  show ?thesis
  proof (intro exI conjI)
    show 0 < t by fact
    show t < h by fact
    show f h =
      (∑ m = 0..<n. diff m 0 / real (fact m) * h ^ m) +
      diff n t / real (fact n) * h ^ n
      using ‹difg (Suc m) t = 0›
      by (simp add: m f-h difg-def del: fact-Suc)
  qed

qed

lemma Maclaurin-objl:
  0 < h & n>0 & diff 0 = f &
  (∀ m t. m < n & 0 ≤ t & t ≤ h --> DERIV (diff m) t :> diff (Suc m) t)
   --> (∃ t. 0 < t & t < h &
          f h = (∑ m=0..<n. diff m 0 / real (fact m) * h ^ m) +
                diff n t / real (fact n) * h ^ n)
by (blast intro: Maclaurin)


lemma Maclaurin2:
   [| 0 < h; diff 0 = f;
      ∀ m t.
         m < n & 0 ≤ t & t ≤ h --> DERIV (diff m) t :> diff (Suc m) t |]
   ==> ∃ t. 0 < t &
            t ≤ h &
            f h =
            (∑ m=0..<n. diff m 0 / real (fact m) * h ^ m) +
            diff n t / real (fact n) * h ^ n
apply (case-tac n, auto)
apply (drule Maclaurin, auto)
done

lemma Maclaurin2-objl:
     0 < h & diff 0 = f &
      (∀ m t.
         m < n & 0 ≤ t & t ≤ h --> DERIV (diff m) t :> diff (Suc m) t)
   --> (∃ t. 0 < t &
            t ≤ h &
            f h =
            (∑ m=0..<n. diff m 0 / real (fact m) * h ^ m) +
            diff n t / real (fact n) * h ^ n)
by (blast intro: Maclaurin2)
```

lemma *Maclaurin-minus*:
[| $h < 0$; $n > 0$; *diff* $0 = f$;
$\forall m\ t.\ m < n \ \&\ h \leq t \ \&\ t \leq 0 \ \text{--->}\ \mathit{DERIV}\ (\mathit{diff}\ m)\ t :> \mathit{diff}\ (\mathit{Suc}\ m)\ t$ |]
==> $\exists t.\ h < t \ \&$
$t < 0 \ \&$
$f\ h =$
$(\sum m{=}0..{<}n.\ \mathit{diff}\ m\ 0\ /\ \mathit{real}\ (\mathit{fact}\ m) * h\ \hat{}\ m) +$
$\mathit{diff}\ n\ t\ /\ \mathit{real}\ (\mathit{fact}\ n) * h\ \hat{}\ n$
apply (*cut-tac* $f = \%x.\ f\ (-x)$
and *diff* $= \%n\ x.\ (-1\ \hat{}\ n) * \mathit{diff}\ n\ (-x)$
and $h = -h$ **and** $n = n$ **in** *Maclaurin-objl*)
apply (*simp*)
apply *safe*
apply (*subst minus-mult-right*)
apply (*rule DERIV-cmult*)
apply (*rule lemma-DERIV-subst*)
apply (*rule DERIV-chain2* [**where** *g=uminus*])
apply (*rule-tac* [*2*] *DERIV-minus*, *rule-tac* [*2*] *DERIV-ident*)
prefer *2* **apply** *force*
apply *force*
apply (*rule-tac* $x = -t$ **in** *exI*, *auto*)
apply (*subgoal-tac* $(\sum m = 0..{<}n.\ -1\ \hat{}\ m * \mathit{diff}\ m\ 0 * (-h)\ \hat{}\ m\ /\ \mathit{real}(\mathit{fact}\ m))$
=
$(\sum m = 0..{<}n.\ \mathit{diff}\ m\ 0 * h\ \hat{}\ m\ /\ \mathit{real}(\mathit{fact}\ m)))$
apply (*rule-tac* [*2*] *setsum-cong*[*OF refl*])
apply (*auto simp add*: *divide-inverse power-mult-distrib* [*symmetric*])
done

lemma *Maclaurin-minus-objl*:
$(h < 0 \ \&\ n > 0 \ \&\ \mathit{diff}\ 0 = f \ \&$
$(\forall m\ t.$
$m < n \ \&\ h \leq t \ \&\ t \leq 0 \ \text{--->}\ \mathit{DERIV}\ (\mathit{diff}\ m)\ t :> \mathit{diff}\ (\mathit{Suc}\ m)\ t))$
$\text{--->}\ (\exists t.\ h < t \ \&$
$t < 0 \ \&$
$f\ h =$
$(\sum m{=}0..{<}n.\ \mathit{diff}\ m\ 0\ /\ \mathit{real}\ (\mathit{fact}\ m) * h\ \hat{}\ m) +$
$\mathit{diff}\ n\ t\ /\ \mathit{real}\ (\mathit{fact}\ n) * h\ \hat{}\ n)$
by (*blast intro*: *Maclaurin-minus*)

67.2 More Convenient "Bidirectional" Version.

lemma *Maclaurin-bi-le-lemma* [*rule-format*]:
$n{>}0 \longrightarrow$
diff $0\ 0 =$
$(\sum m = 0..{<}n.\ \mathit{diff}\ m\ 0 * 0\ \hat{}\ m\ /\ \mathit{real}\ (\mathit{fact}\ m)) +$
$\mathit{diff}\ n\ 0 * 0\ \hat{}\ n\ /\ \mathit{real}\ (\mathit{fact}\ n)$
by (*induct n*, *auto*)

lemma *Maclaurin-bi-le*:

```
  [| diff 0 = f;
     ∀ m t. m < n & abs t ≤ abs x --> DERIV (diff m) t :> diff (Suc m) t |]
   ==> ∃ t. abs t ≤ abs x &
            f x =
            (∑ m=0..<n. diff m 0 / real (fact m) * x ^ m) +
            diff n t / real (fact n) * x ^ n
apply (case-tac n = 0, force)
apply (case-tac x = 0)
 apply (rule-tac x = 0 in exI)
 apply (force simp add: Maclaurin-bi-le-lemma)
apply (cut-tac x = x and y = 0 in linorder-less-linear, auto)
```

Case 1, where $x < 0$

```
apply (cut-tac f = diff 0 and diff = diff and h = x and n = n in Maclaurin-minus-objl,
safe)
  apply (simp add: abs-if)
 apply (rule-tac x = t in exI)
 apply (simp add: abs-if)
```

Case 2, where $0 < x$

```
apply (cut-tac f = diff 0 and diff = diff and h = x and n = n in Maclaurin-objl,
safe)
 apply (simp add: abs-if)
apply (rule-tac x = t in exI)
apply (simp add: abs-if)
done

lemma Maclaurin-all-lt:
     [| diff 0 = f;
        ∀ m x. DERIV (diff m) x :> diff(Suc m) x;
       x ~= 0; n > 0
      |] ==> ∃ t. 0 < abs t & abs t < abs x &
             f x = (∑ m=0..<n. (diff m 0 / real (fact m)) * x ^ m) +
                   (diff n t / real (fact n)) * x ^ n
apply (rule-tac x = x and y = 0 in linorder-cases)
prefer 2 apply blast
apply (drule-tac [2] diff=diff in Maclaurin)
apply (drule-tac diff=diff in Maclaurin-minus, simp-all, safe)
apply (rule-tac [!] x = t in exI, auto)
done

lemma Maclaurin-all-lt-objl:
     diff 0 = f &
     (∀ m x. DERIV (diff m) x :> diff(Suc m) x) &
     x ~= 0 & n > 0
     --> (∃ t. 0 < abs t & abs t < abs x &
             f x = (∑ m=0..<n. (diff m 0 / real (fact m)) * x ^ m) +
                   (diff n t / real (fact n)) * x ^ n)
by (blast intro: Maclaurin-all-lt)
```

lemma *Maclaurin-zero* [*rule-format*]:
$x = (0::real)$
$\Longrightarrow n \neq 0 \longrightarrow$
$(\sum m{=}0..{<}n.\ (diff\ m\ (0::real)\ /\ real\ (fact\ m)) * x\ \hat{}\ m) =$
$diff\ 0\ 0$
by (*induct n*, *auto*)

lemma *Maclaurin-all-le*: [| $diff\ 0 = f$;
$\forall m\ x.\ DERIV\ (diff\ m)\ x :> diff\ (Suc\ m)\ x$
|] ==> $\exists t.\ abs\ t \leq abs\ x\ \&$
$f\ x = (\sum m{=}0..{<}n.\ (diff\ m\ 0\ /\ real\ (fact\ m)) * x\ \hat{}\ m) +$
$(diff\ n\ t\ /\ real\ (fact\ n)) * x\ \hat{}\ n$
apply(*cases n=0*)
apply (*force*)
apply (*case-tac x = 0*)
apply (*frule-tac diff = diff* **and** *n = n* **in** *Maclaurin-zero*, *assumption*)
apply (*drule not0-implies-Suc*)
apply (*rule-tac x = 0* **in** *exI*, *force*)
apply (*frule-tac diff = diff* **and** *n = n* **in** *Maclaurin-all-lt*, *auto*)
apply (*rule-tac x = t* **in** *exI*, *auto*)
done

lemma *Maclaurin-all-le-objl*: $diff\ 0 = f\ \&$
$(\forall m\ x.\ DERIV\ (diff\ m)\ x :> diff\ (Suc\ m)\ x)$
$\longrightarrow (\exists t.\ abs\ t \leq abs\ x\ \&$
$f\ x = (\sum m{=}0..{<}n.\ (diff\ m\ 0\ /\ real\ (fact\ m)) * x\ \hat{}\ m) +$
$(diff\ n\ t\ /\ real\ (fact\ n)) * x\ \hat{}\ n)$
by (*blast intro*: *Maclaurin-all-le*)

67.3 Version for Exponential Function

lemma *Maclaurin-exp-lt*: [| $x\ \tilde{}{=}\ 0$; $n > 0$ |]
==> $(\exists t.\ 0 < abs\ t\ \&$
$abs\ t < abs\ x\ \&$
$exp\ x = (\sum m{=}0..{<}n.\ (x\ \hat{}\ m)\ /\ real\ (fact\ m)) +$
$(exp\ t\ /\ real\ (fact\ n)) * x\ \hat{}\ n)$
by (*cut-tac diff* = %*n*. *exp* **and** *f* = *exp* **and** *x* = *x* **and** *n* = *n* **in** *Maclaurin-all-lt-objl*, *auto*)

lemma *Maclaurin-exp-le*:
$\exists t.\ abs\ t \leq abs\ x\ \&$
$exp\ x = (\sum m{=}0..{<}n.\ (x\ \hat{}\ m)\ /\ real\ (fact\ m)) +$
$(exp\ t\ /\ real\ (fact\ n)) * x\ \hat{}\ n$
by (*cut-tac diff* = %*n*. *exp* **and** *f* = *exp* **and** *x* = *x* **and** *n* = *n* **in** *Maclaurin-all-le-objl*, *auto*)

67.4 Version for Sine Function

lemma *mod-exhaust-less-4*:
m mod 4 = 0 | m mod 4 = 1 | m mod 4 = 2 | m mod 4 = (3::nat)
by *auto*

lemma *Suc-Suc-mult-two-diff-two* [*rule-format*, *simp*]:
*n≠0 −−> Suc (Suc (2 * n − 2)) = 2*n*
by (*induct n*, *auto*)

lemma *lemma-Suc-Suc-4n-diff-2* [*rule-format*, *simp*]:
*n≠0 −−> Suc (Suc (4*n − 2)) = 4*n*
by (*induct n*, *auto*)

lemma *Suc-mult-two-diff-one* [*rule-format*, *simp*]:
*n≠0 −−> Suc (2 * n − 1) = 2*n*
by (*induct n*, *auto*)

It is unclear why so many variant results are needed.

lemma *Maclaurin-sin-expansion2*:
∃ t. abs t ≤ abs x &
sin x =
(∑ m=0..<n. (if even m then 0
*else (−1 ^ ((m − Suc 0) div 2)) / real (fact m)) **
x ^ m)
*+ ((sin(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)*
apply (*cut-tac f = sin* **and** *n = n* **and** *x = x*
and *diff = %n x. sin (x + 1/2*real n * pi)* **in** *Maclaurin-all-lt-objl*)
apply *safe*
apply (*simp* (*no-asm*))
apply (*simp* (*no-asm*))
apply (*case-tac n*, *clarify*, *simp*, *simp add*: *lemma-STAR-sin*)
apply (*rule ccontr*, *simp*)
apply (*drule-tac x = x* **in** *spec*, *simp*)
apply (*erule ssubst*)
apply (*rule-tac x = t* **in** *exI*, *simp*)
apply (*rule setsum-cong*[*OF refl*])
apply (*auto simp add*: *sin-zero-iff odd-Suc-mult-two-ex*)
done

lemma *Maclaurin-sin-expansion*:
∃ t. sin x =
(∑ m=0..<n. (if even m then 0
*else (−1 ^ ((m − Suc 0) div 2)) / real (fact m)) **
x ^ m)
*+ ((sin(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)*
apply (*insert Maclaurin-sin-expansion2* [*of x n*])
apply (*blast intro*: *elim*:)
done

lemma *Maclaurin-sin-expansion3*:
[| $n > 0$; $0 < x$ |] ==>
$\exists t.\ 0 < t \ \&\ t < x \ \&$
sin $x =$
$(\sum m{=}0..{<}n.$ (*if even m then 0*
else $(-1\ \hat{}\ ((m - Suc\ 0)\ div\ 2))\ /\ real\ (fact\ m)) *$
$x\ \hat{}\ m)$
$+ ((sin(t + 1/2 * real(n) * pi)\ /\ real\ (fact\ n)) * x\ \hat{}\ n)$
apply (*cut-tac f = sin* **and** $n = n$ **and** $h = x$ **and** *diff* $= \%n\ x.\ sin\ (x + 1/2{*}real$
$(n) * pi)$ **in** *Maclaurin-objl*)
apply *safe*
apply *simp*
apply (*simp* (*no-asm*))
apply (*erule ssubst*)
apply (*rule-tac* $x = t$ **in** *exI*, *simp*)
apply (*rule setsum-cong*[*OF refl*])
apply (*auto simp add*: *sin-zero-iff odd-Suc-mult-two-ex*)
done

lemma *Maclaurin-sin-expansion4*:
$0 < x$ ==>
$\exists t.\ 0 < t \ \&\ t \leq x \ \&$
sin $x =$
$(\sum m{=}0..{<}n.$ (*if even m then 0*
else $(-1\ \hat{}\ ((m - Suc\ 0)\ div\ 2))\ /\ real\ (fact\ m)) *$
$x\ \hat{}\ m)$
$+ ((sin(t + 1/2 * real\ (n) * pi)\ /\ real\ (fact\ n)) * x\ \hat{}\ n)$
apply (*cut-tac f = sin* **and** $n = n$ **and** $h = x$ **and** *diff* $= \%n\ x.\ sin\ (x + 1/2{*}real$
$(n) * pi)$ **in** *Maclaurin2-objl*)
apply *safe*
apply *simp*
apply (*simp* (*no-asm*))
apply (*erule ssubst*)
apply (*rule-tac* $x = t$ **in** *exI*, *simp*)
apply (*rule setsum-cong*[*OF refl*])
apply (*auto simp add*: *sin-zero-iff odd-Suc-mult-two-ex*)
done

67.5 Maclaurin Expansion for Cosine Function

lemma *sumr-cos-zero-one* [*simp*]:
$(\sum m{=}0..{<}(Suc\ n).$
(*if even m then* $-1\ \hat{}\ (m\ div\ 2)/(real\ \ (fact\ m))$ *else 0*) $* 0\ \hat{}\ m) = 1$
by (*induct n*, *auto*)

lemma *Maclaurin-cos-expansion*:
$\exists t.\ abs\ t \leq abs\ x \ \&$
cos $x =$

```
      (∑ m=0..<n. (if even m
                   then -1 ^ (m div 2)/(real (fact m))
                   else 0) *
                   x ^ m)
    + ((cos(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)
apply (cut-tac f = cos and n = n and x = x and diff = %n x. cos (x + 1/2*real
(n) *pi) in Maclaurin-all-lt-objl)
apply safe
apply (simp (no-asm))
apply (simp (no-asm))
apply (case-tac n, simp)
apply (simp del: setsum-op-ivl-Suc)
apply (rule ccontr, simp)
apply (drule-tac x = x in spec, simp)
apply (erule ssubst)
apply (rule-tac x = t in exI, simp)
apply (rule setsum-cong[OF refl])
apply (auto simp add: cos-zero-iff even-mult-two-ex)
done

lemma Maclaurin-cos-expansion2:
    [| 0 < x; n > 0 |] ==>
     ∃ t. 0 < t & t < x &
     cos x =
     (∑ m=0..<n. (if even m
                  then -1 ^ (m div 2)/(real (fact m))
                  else 0) *
                  x ^ m)
    + ((cos(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)
apply (cut-tac f = cos and n = n and h = x and diff = %n x. cos (x + 1/2*real
(n) *pi) in Maclaurin-objl)
apply safe
apply simp
apply (simp (no-asm))
apply (erule ssubst)
apply (rule-tac x = t in exI, simp)
apply (rule setsum-cong[OF refl])
apply (auto simp add: cos-zero-iff even-mult-two-ex)
done

lemma Maclaurin-minus-cos-expansion:
    [| x < 0; n > 0 |] ==>
     ∃ t. x < t & t < 0 &
     cos x =
     (∑ m=0..<n. (if even m
                  then -1 ^ (m div 2)/(real (fact m))
                  else 0) *
                  x ^ m)
    + ((cos(t + 1/2 * real (n) *pi) / real (fact n)) * x ^ n)
```

apply (*cut-tac f = cos* **and** *n = n* **and** *h = x* **and** *diff = %n x. cos (x + 1/2*real (n) *pi)* **in** *Maclaurin-minus-objl*)
apply *safe*
apply *simp*
apply (*simp (no-asm)*)
apply (*erule ssubst*)
apply (*rule-tac x = t* **in** *exI*, *simp*)
apply (*rule setsum-cong*[*OF refl*])
apply (*auto simp add*: *cos-zero-iff even-mult-two-ex*)
done

lemma *sin-bound-lemma*:
  [|*x = y*; *abs u ≤ (v*::*real)* |] ==> |(*x + u*) − *y*| ≤ *v*
by *auto*

TODO: move to Parity.thy

lemma *nat-odd-1* [*simp*]: *odd* (*1*::*nat*)
  unfolding *even-nat-def* **by** *simp*

lemma *Maclaurin-sin-bound*:
  abs(*sin x* − (∑ *m*=*0*..<*n*. (*if even m then 0 else* (−*1* ^ ((*m* − *Suc 0*) *div 2*)) / *real* (*fact m*)) *
  x ^ *m*)) ≤ *inverse*(*real* (*fact n*)) * |*x*| ^ *n*
proof −
  have !! *x* (*y*::*real*). *x ≤ 1* ⟹ *0 ≤ y* ⟹ *x * y ≤ 1 * y*
    by (*rule-tac mult-right-mono*,*simp-all*)
  note *est = this*[*simplified*]
  let *?diff = λ*(*n*::*nat*) *x*. *if n mod 4 = 0 then sin*(*x*) *else if n mod 4 = 1 then cos*(*x*) *else if n mod 4 = 2 then* −*sin*(*x*) *else* −*cos*(*x*)
  have *diff-0*: *?diff 0 = sin* **by** *simp*
  have *DERIV-diff*: ∀ *m x*. *DERIV* (*?diff m*) *x* :> *?diff* (*Suc m*) *x*
    apply (*clarify*)
    apply (*subst* (*1 2 3*) *mod-Suc-eq-Suc-mod*)
    apply (*cut-tac m*=*m* **in** *mod-exhaust-less-4*)
    apply (*safe*, *simp-all*)
    apply (*rule DERIV-minus*, *simp*)
    apply (*rule lemma-DERIV-subst*, *rule DERIV-minus*, *rule DERIV-cos*, *simp*)
    done
  from *Maclaurin-all-le* [*OF diff-0 DERIV-diff*]
  obtain *t* **where** *t1*: |*t*| ≤ |*x*| **and**
    t2: *sin x* = (∑ *m* = *0*..<*n*. *?diff m 0* / *real* (*fact m*) * *x* ^ *m*) +
      ?diff n t / *real* (*fact n*) * *x* ^ *n* **by** *fast*
  have *diff-m-0*:
    ⋀*m*. *?diff m 0* = (*if even m then 0*
      else −*1* ^ ((*m* − *Suc 0*) *div 2*))

apply (*subst even-even-mod-4-iff*)
apply (*cut-tac m=m* **in** *mod-exhaust-less-4*)
apply (*elim disjE*, *simp-all*)
apply (*safe dest*!: *mod-eqD*, *simp-all*)
done
show *?thesis*
apply (*subst t2*)
apply (*rule sin-bound-lemma*)
apply (*rule setsum-cong*[*OF refl*])
apply (*subst diff-m-0*, *simp*)
apply (*auto intro*: *mult-right-mono* [**where** *b=1*, *simplified*] *mult-right-mono*
simp add: *est mult-nonneg-nonneg mult-ac divide-inverse*
power-abs [*symmetric*] *abs-mult*)
done
qed

end

68 Taylor: Taylor series

theory *Taylor*
imports *MacLaurin*
begin

We use MacLaurin and the translation of the expansion point c to 0 to prove Taylor's theorem.

lemma *taylor-up*:
assumes *INIT*: $n>0$ *diff* $0 = f$
and *DERIV*: $(\forall\ m\ t.\ m < n\ \&\ a \leq t\ \&\ t \leq b \longrightarrow DERIV\ (diff\ m)\ t :> (diff\ (Suc\ m)\ t))$
and *INTERV*: $a \leq c\ c < b$
shows $\exists\ t.\ c < t\ \&\ t < b\ \&$
$f\ b = setsum\ (\%m.\ (diff\ m\ c\ /\ real\ (fact\ m)) * (b - c)\ \hat{}\ m)\ \{0..<n\} +$
$(diff\ n\ t\ /\ real\ (fact\ n)) * (b - c)\ \hat{}\ n$
proof −
from *INTERV* **have** $0 < b-c$ **by** *arith*
moreover
from *INIT* **have** $n>0\ ((\lambda m\ x.\ diff\ m\ (x + c))\ 0) = (\lambda x.\ f\ (x + c))$ **by** *auto*
moreover
have *ALL m t. m < n & 0 <= t & t <= b - c −−> DERIV (%x. diff m (x + c)) t :> diff (Suc m) (t + c)*
proof (*intro strip*)
fix *m t*
assume *m < n & 0 <= t & t <= b - c*
with *DERIV* **and** *INTERV* **have** *DERIV (diff m) (t + c) :> diff (Suc m) (t + c)* **by** *auto*
moreover

from *DERIV-ident* **and** *DERIV-const* **have** *DERIV* (%*x*. *x* + *c*) *t* :> *1+0*
by (*rule DERIV-add*)
ultimately have *DERIV* (%*x*. *diff m* (*x* + *c*)) *t* :> *diff* (*Suc m*) (*t* + *c*) ∗ (*1+0*)
by (*rule DERIV-chain2*)
thus *DERIV* (%*x*. *diff m* (*x* + *c*)) *t* :> *diff* (*Suc m*) (*t* + *c*) **by** *simp*
qed
ultimately
have *EX*:*EX t*>*0*. *t* < *b* − *c* &
f (*b* − *c* + *c*) = (*SUM m* = *0*..<*n*. *diff m* (*0* + *c*) / *real* (*fact m*) ∗ (*b* − *c*) ^ *m*) +
diff n (*t* + *c*) / *real* (*fact n*) ∗ (*b* − *c*) ^ *n*
by (*rule Maclaurin*)
show *?thesis*
proof −
from *EX* **obtain** *x* **where**
X: *0* < *x* & *x* < *b* − *c* &
f (*b* − *c* + *c*) = (∑ *m* = *0*..<*n*. *diff m* (*0* + *c*) / *real* (*fact m*) ∗ (*b* − *c*) ^ *m*) +
diff n (*x* + *c*) / *real* (*fact n*) ∗ (*b* − *c*) ^ *n* **..**
let *?H* = *x* + *c*
from *X* **have** *c*<*?H* & *?H*<*b* ∧ *f b* = (∑ *m* = *0*..<*n*. *diff m c* / *real* (*fact m*) ∗ (*b* − *c*) ^ *m*) +
diff n ?H / *real* (*fact n*) ∗ (*b* − *c*) ^ *n*
by *fastsimp*
thus *?thesis* **by** *fastsimp*
qed
qed

lemma *taylor-down*:
assumes *INIT*: *n*>*0 diff 0* = *f*
and *DERIV*: (∀ *m t*. *m* < *n* & *a* ≤ *t* & *t* ≤ *b* ⟶ *DERIV* (*diff m*) *t* :> (*diff* (*Suc m*) *t*))
and *INTERV*: *a* < *c c* ≤ *b*
shows ∃ *t*. *a* < *t* & *t* < *c* &
f a = *setsum* (% *m*. (*diff m c* / *real* (*fact m*)) ∗ (*a* − *c*)^*m*) {*0*..<*n*} +
(*diff n t* / *real* (*fact n*)) ∗ (*a* − *c*)^*n*
proof −
from *INTERV* **have** *a*−*c* < *0* **by** *arith*
moreover
from *INIT* **have** *n*>*0* ((λ*m x*. *diff m* (*x* + *c*)) *0*) = (λ*x*. *f* (*x* + *c*)) **by** *auto*
moreover
have *ALL m t*. *m* < *n* & *a*−*c* <= *t* & *t* <= *0* −−> *DERIV* (%*x*. *diff m* (*x* + *c*)) *t* :> *diff* (*Suc m*) (*t* + *c*)
proof (*rule allI impI*)+
fix *m t*
assume *m* < *n* & *a*−*c* <= *t* & *t* <= *0*
with *DERIV* **and** *INTERV* **have** *DERIV* (*diff m*) (*t* + *c*) :> *diff* (*Suc m*) (*t* + *c*) **by** *auto*

```
    moreover
    from DERIV-ident and DERIV-const have DERIV (%x. x + c) t :> 1+0
by (rule DERIV-add)
    ultimately have DERIV (%x. diff m (x + c)) t :> diff (Suc m) (t + c) *
(1+0) by (rule DERIV-chain2)
    thus DERIV (%x. diff m (x + c)) t :> diff (Suc m) (t + c) by simp
  qed
  ultimately
  have EX: EX t>a - c. t < 0 &
    f (a - c + c) = (SUM m = 0..<n. diff m (0 + c) / real (fact m) * (a - c)
^ m) +
      diff n (t + c) / real (fact n) * (a - c) ^ n
    by (rule Maclaurin-minus)
  show ?thesis
  proof -
    from EX obtain x where X: a - c < x & x < 0 &
      f (a - c + c) = (SUM m = 0..<n. diff m (0 + c) / real (fact m) * (a - c)
^ m) +
        diff n (x + c) / real (fact n) * (a - c) ^ n ..
    let ?H = x + c
    from X have a<?H & ?H<c ∧ f a = (∑ m = 0..<n. diff m c / real (fact
m) * (a - c) ^ m) +
      diff n ?H / real (fact n) * (a - c) ^ n
      by fastsimp
    thus ?thesis by fastsimp
  qed
qed

lemma taylor:
  assumes INIT: n>0 diff 0 = f
  and DERIV: (∀ m t. m < n & a ≤ t & t ≤ b ⟶ DERIV (diff m) t :> (diff
(Suc m) t))
  and INTERV: a ≤ c  c ≤ b a ≤ x x ≤ b x ≠ c
  shows ∃ t. (if x<c then (x < t & t < c) else (c < t & t < x)) &
    f x = setsum (% m. (diff m c / real (fact m)) * (x - c)^m) {0..<n} +
      (diff n t / real (fact n)) * (x - c)^n
proof (cases x<c)
  case True
  note INIT
  moreover from DERIV and INTERV
  have ∀ m t. m < n ∧ x ≤ t ∧ t ≤ b ⟶ DERIV (diff m) t :> diff (Suc m) t
    by fastsimp
  moreover note True
  moreover from INTERV have c ≤ b by simp
  ultimately have EX: ∃ t>x. t < c ∧ f x =
    (∑ m = 0..<n. diff m c / real (fact m) * (x - c) ^ m) +
      diff n t / real (fact n) * (x - c) ^ n
    by (rule taylor-down)
  with True show ?thesis by simp
```

next
case *False*
note *INIT*
moreover from *DERIV* **and** *INTERV*
have $\forall m\ t.\ m < n \wedge a \leq t \wedge t \leq x \longrightarrow DERIV\ (diff\ m)\ t :> diff\ (Suc\ m)\ t$
by *fastsimp*
moreover from *INTERV* **have** $a \leq c$ **by** *arith*
moreover from *False* **and** *INTERV* **have** $c < x$ **by** *arith*
ultimately have *EX*: $\exists t{>}c.\ t < x \wedge f\ x =$
$(\sum m = 0..{<}n.\ diff\ m\ c\ /\ real\ (fact\ m) * (x - c)\ \hat{}\ m) +$
$diff\ n\ t\ /\ real\ (fact\ n) * (x - c)\ \hat{}\ n$
by (*rule taylor-up*)
with *False* **show** *?thesis* **by** *simp*
qed

end

69 Integration: Theory of Integration

theory *Integration*
imports *Deriv ATP-Linkup*
begin

We follow John Harrison in formalizing the Gauge integral.

definition
— Partitions and tagged partitions etc.

partition :: [(*real*∗*real*),*nat* => *real*] => *bool* **where**
[*code del*]: *partition* = (%(*a*,*b*) *D*. *D* 0 = *a* &
(∃ *N*. (∀ *n* < *N*. *D*(*n*) < *D*(*Suc n*)) &
(∀ *n* ≥ *N*. *D*(*n*) = *b*)))

definition
psize :: (*nat* => *real*) => *nat* **where**
[*code del*]:*psize D* = (*SOME N*. (∀ *n* < *N*. *D*(*n*) < *D*(*Suc n*)) &
(∀ *n* ≥ *N*. *D*(*n*) = *D*(*N*)))

definition
tpart :: [(*real*∗*real*),((*nat* => *real*)∗(*nat* =>*real*))] => *bool* **where**
[*code del*]:*tpart* = (%(*a*,*b*) (*D*,*p*). *partition*(*a*,*b*) *D* &
(∀ *n*. *D*(*n*) ≤ *p*(*n*) & *p*(*n*) ≤ *D*(*Suc n*)))

— Gauges and gauge-fine divisions

definition
gauge :: [*real* => *bool*, *real* => *real*] => *bool* **where**
[*code del*]:*gauge E g* = (∀ *x*. *E x* −−> 0 < *g*(*x*))

definition
fine :: [*real* => *real*, ((*nat* => *real*)*(*nat* => *real*))] => *bool* **where**
[*code del*]:*fine* = (%*g* (*D*,*p*). ∀ *n*. *n* < (*psize D*) −−> *D*(*Suc n*) − *D*(*n*) < *g*(*p n*))

— Riemann sum

definition
rsum :: [((*nat*=>*real*)*(*nat*=>*real*)),*real*=>*real*] => *real* **where**
rsum = (%(*D*,*p*) *f*. ∑ *n*=*0*..<*psize*(*D*). *f*(*p n*) * (*D*(*Suc n*) − *D*(*n*)))

— Gauge integrability (definite)

definition
Integral :: [(*real*∗*real*),*real*=>*real*,*real*] => *bool* **where**
[*code del*]: *Integral* = (%(*a*,*b*) *f k*. ∀ *e* > *0*.
(∃ *g*. *gauge*(%*x*. *a* ≤ *x* & *x* ≤ *b*) *g* &
(∀ *D p*. *tpart*(*a*,*b*) (*D*,*p*) & *fine*(*g*)(*D*,*p*) −−>
|*rsum*(*D*,*p*) *f* − *k*| < *e*)))

lemma *psize-unique*:
assumes *1*: ∀ *n* < *N*. *D*(*n*) < *D*(*Suc n*)
assumes *2*: ∀ *n* ≥ *N*. *D*(*n*) = *D*(*N*)
shows *psize D* = *N*
unfolding *psize-def*
proof (*rule some-equality*)
show (∀ *n*<*N*. *D*(*n*) < *D*(*Suc n*)) ∧ (∀ *n*≥*N*. *D*(*n*) = *D*(*N*)) **using** *prems* **..**
next
fix *M* **assume** (∀ *n*<*M*. *D*(*n*) < *D*(*Suc n*)) ∧ (∀ *n*≥*M*. *D*(*n*) = *D*(*M*))
hence *3*: ∀ *n*<*M*. *D*(*n*) < *D*(*Suc n*) **and** *4*: ∀ *n*≥*M*. *D*(*n*) = *D*(*M*) **by** *fast+*
show *M* = *N*
proof (*rule linorder-cases*)
assume *M* < *N*
hence *D*(*M*) < *D*(*Suc M*) **by** (*rule 1* [*rule-format*])
also have *D*(*Suc M*) = *D*(*M*) **by** (*rule 4* [*rule-format*], *simp*)
finally show *M* = *N* **by** *simp*
next
assume *N* < *M*
hence *D*(*N*) < *D*(*Suc N*) **by** (*rule 3* [*rule-format*])
also have *D*(*Suc N*) = *D*(*N*) **by** (*rule 2* [*rule-format*], *simp*)
finally show *M* = *N* **by** *simp*
next
assume *M* = *N* **thus** *M* = *N* **.**
qed
qed

lemma *partition-zero* [*simp*]: *a* = *b* ==> *psize* (%*n*. *if n* = *0 then a else b*) = *0*
by (*rule psize-unique*, *simp-all*)

lemma *partition-one* [*simp*]: $a < b \Longrightarrow psize\ (\%n.\ if\ n = 0\ then\ a\ else\ b) = 1$
by (*rule psize-unique*, *simp-all*)

lemma *partition-single* [*simp*]:
$a \leq b \Longrightarrow partition(a,b)(\%n.\ if\ n = 0\ then\ a\ else\ b)$
by (*auto simp add*: *partition-def order-le-less*)

lemma *partition-lhs*: $partition(a,b)\ D \Longrightarrow (D(0) = a)$
by (*simp add*: *partition-def*)

lemma *partition*:
$(partition(a,b)\ D) =$
$((D\ 0 = a)\ \&$
$(\forall n < psize\ D.\ D\ n < D(Suc\ n))\ \&$
$(\forall n \geq psize\ D.\ D\ n = b))$
apply (*simp add*: *partition-def*)
apply (*rule iffI*, *clarify*)
apply (*subgoal-tac psize D = N*, *simp*)
apply (*rule psize-unique*, *assumption*, *simp*)
apply (*simp*, *rule-tac x=psize D* **in** *exI*, *simp*)
done

lemma *partition-rhs*: $partition(a,b)\ D \Longrightarrow (D(psize\ D) = b)$
by (*simp add*: *partition*)

lemma *partition-rhs2*: $[|partition(a,b)\ D;\ psize\ D \leq n\ |] \Longrightarrow (D\ n = b)$
by (*simp add*: *partition*)

lemma *lemma-partition-lt-gen* [*rule-format*]:
$partition(a,b)\ D\ \&\ m + Suc\ d \leq n\ \&\ n \leq (psize\ D) \longrightarrow D(m) < D(m + Suc\ d)$
apply (*induct d*, *auto simp add*: *partition*)
apply (*blast dest*: *Suc-le-lessD intro*: *less-le-trans order-less-trans*)
done

lemma *less-eq-add-Suc*: $m < n \Longrightarrow \exists d.\ n = m + Suc\ d$
by (*auto simp add*: *less-iff-Suc-add*)

lemma *partition-lt-gen*:
$[|partition(a,b)\ D;\ m < n;\ n \leq (psize\ D)|] \Longrightarrow D(m) < D(n)$
by (*auto dest*: *less-eq-add-Suc intro*: *lemma-partition-lt-gen*)

lemma *partition-lt*: $partition(a,b)\ D \Longrightarrow n < (psize\ D) \Longrightarrow D(0) < D(Suc\ n)$
apply (*induct n*)
apply (*auto simp add*: *partition*)
done

lemma *partition-le*: $partition(a,b)\ D \Longrightarrow a \leq b$

apply (*frule partition* [*THEN iffD1*], *safe*)
apply (*drule-tac x = psize D* **and** *P=%n. psize D ≤ n −−> ?P n* **in** *spec*, *safe*)
apply (*case-tac psize D = 0*)
apply (*drule-tac* [*2*] *n = psize D − Suc 0* **in** *partition-lt*, *auto*)
done

lemma *partition-gt*: [|*partition*(*a*,*b*) *D*; *n* < (*psize D*)|] ==> *D*(*n*) < *D*(*psize D*)
by (*auto intro*: *partition-lt-gen*)

lemma *partition-eq*: *partition*(*a*,*b*) *D* ==> ((*a* = *b*) = (*psize D* = *0*))
apply (*frule partition* [*THEN iffD1*], *safe*)
apply (*rotate-tac 2*)
apply (*drule-tac x = psize D* **in** *spec*)
apply (*rule ccontr*)
apply (*drule-tac n = psize D − Suc 0* **in** *partition-lt*)
apply *auto*
done

lemma *partition-lb*: *partition*(*a*,*b*) *D* ==> *a* ≤ *D*(*r*)
apply (*frule partition* [*THEN iffD1*], *safe*)
apply (*induct r*)
apply (*cut-tac* [*2*] *y = Suc r* **and** *x = psize D* **in** *linorder-le-less-linear*)
apply (*auto intro*: *partition-le*)
apply (*drule-tac x = r* **in** *spec*)
apply *arith*
done

lemma *partition-lb-lt*: [| *partition*(*a*,*b*) *D*; *psize D* ~= *0* |] ==> *a* < *D*(*Suc n*)
apply (*rule-tac t = a* **in** *partition-lhs* [*THEN subst*], *assumption*)
apply (*cut-tac x = Suc n* **and** *y = psize D* **in** *linorder-le-less-linear*)
apply (*frule partition* [*THEN iffD1*], *safe*)
apply (*blast intro*: *partition-lt less-le-trans*)
apply (*rotate-tac 3*)
apply (*drule-tac x = Suc n* **in** *spec*)
apply (*erule impE*)
apply (*erule less-imp-le*)
apply (*frule partition-rhs*)
apply (*drule partition-gt*[*of - - - 0*], *arith*)
apply (*simp* (*no-asm-simp*))
done

lemma *partition-ub*: *partition*(*a*,*b*) *D* ==> *D*(*r*) ≤ *b*
apply (*frule partition* [*THEN iffD1*])
apply (*cut-tac x = psize D* **and** *y = r* **in** *linorder-le-less-linear*, *safe*, *blast*)
apply (*subgoal-tac* ∀ *x*. *D* ((*psize D*) − *x*) ≤ *b*)
apply (*rotate-tac 4*)
apply (*drule-tac x = psize D − r* **in** *spec*)
apply (*subgoal-tac psize D − (psize D − r) = r*)
apply *simp*

apply *arith*
apply *safe*
apply (*induct-tac x*)
apply (*simp* (*no-asm*), *blast*)
apply (*case-tac psize D − Suc n = 0*)
apply (*erule-tac V = ∀ n. psize D ≤ n −−> D n = b* **in** *thin-rl*)
apply (*simp* (*no-asm-simp*) *add*: *partition-le*)
apply (*rule order-trans*)
prefer *2* **apply** *assumption*
apply (*subgoal-tac psize D − n = Suc* (*psize D − Suc n*))
prefer *2* **apply** *arith*
apply (*drule-tac x = psize D − Suc n* **in** *spec*, *simp*)
done

lemma *partition-ub-lt*: [| *partition*(*a*,*b*) *D*; *n < psize D* |] ==> *D*(*n*) < *b*
by (*blast intro*: *partition-rhs* [*THEN subst*] *partition-gt*)

lemma *lemma-partition-append1*:
[| *partition* (*a*, *b*) *D1*; *partition* (*b*, *c*) *D2* |]
==> (∀ *n < psize D1 + psize D2*.
(*if n < psize D1 then D1 n else D2* (*n − psize D1*))
< (*if Suc n < psize D1 then D1* (*Suc n*)
else D2 (*Suc n − psize D1*))) &
(∀ *n ≥ psize D1 + psize D2*.
(*if n < psize D1 then D1 n else D2* (*n − psize D1*)) =
(*if psize D1 + psize D2 < psize D1 then D1* (*psize D1 + psize D2*)
else D2 (*psize D1 + psize D2 − psize D1*)))
apply (*auto intro*: *partition-lt-gen*)
apply (*subgoal-tac psize D1 = Suc n*)
apply (*auto intro*!: *partition-lt-gen simp add*: *partition-lhs partition-ub-lt*)
apply (*auto intro*!: *partition-rhs2 simp add*: *partition-rhs*
split: *nat-diff-split*)
done

lemma *lemma-psize1*:
[| *partition* (*a*, *b*) *D1*; *partition* (*b*, *c*) *D2*; *N < psize D1* |]
==> *D1*(*N*) < *D2* (*psize D2*)
apply (*rule-tac y = D1* (*psize D1*) **in** *order-less-le-trans*)
apply (*erule partition-gt*)
apply (*auto simp add*: *partition-rhs partition-le*)
done

lemma *lemma-partition-append2*:
[| *partition* (*a*, *b*) *D1*; *partition* (*b*, *c*) *D2* |]
==> *psize* (%*n*. *if n < psize D1 then D1 n else D2* (*n − psize D1*)) =
psize D1 + psize D2
apply (*rule psize-unique*)
apply (*erule* (*1*) *lemma-partition-append1* [*THEN conjunct1*])
apply (*erule* (*1*) *lemma-partition-append1* [*THEN conjunct2*])

done

lemma *tpart-eq-lhs-rhs*: [|*psize D = 0*; *tpart*(*a*,*b*) (*D*,*p*)|] ==> *a = b*
by (*auto simp add*: *tpart-def partition-eq*)

lemma *tpart-partition*: *tpart*(*a*,*b*) (*D*,*p*) ==> *partition*(*a*,*b*) *D*
by (*simp add*: *tpart-def*)

lemma *partition-append*:
[| *tpart*(*a*,*b*) (*D1*,*p1*); *fine*(*g*) (*D1*,*p1*);
tpart(*b*,*c*) (*D2*,*p2*); *fine*(*g*) (*D2*,*p2*) |]
==> ∃ *D p*. *tpart*(*a*,*c*) (*D*,*p*) & *fine*(*g*) (*D*,*p*)
apply (*rule-tac x = %n. if n < psize D1 then D1 n else D2 (n − psize D1)*
in *exI*)
apply (*rule-tac x = %n. if n < psize D1 then p1 n else p2 (n − psize D1)*
in *exI*)
apply (*case-tac psize D1 = 0*)
apply (*auto dest*: *tpart-eq-lhs-rhs*)
prefer *2*
apply (*simp add*: *fine-def*
lemma-partition-append2 [*OF tpart-partition tpart-partition*])
— But must not expand *fine* in other subgoals
apply *auto*
apply (*subgoal-tac psize D1 = Suc n*)
prefer *2* **apply** *arith*
apply (*drule tpart-partition* [*THEN partition-rhs*])
apply (*drule tpart-partition* [*THEN partition-lhs*])
apply (*auto split*: *nat-diff-split*)
apply (*auto simp add*: *tpart-def*)
defer *1*
apply (*subgoal-tac psize D1 = Suc n*)
prefer *2* **apply** *arith*
apply (*drule partition-rhs*)
apply (*drule partition-lhs*, *auto*)
apply (*simp split*: *nat-diff-split*)
apply (*subst partition*)
apply (*subst* (*1 2*) *lemma-partition-append2*, *assumption+*)
apply (*rule conjI*)
apply (*simp add*: *partition-lhs*)
apply (*drule lemma-partition-append1*)
apply *assumption*
apply (*simp add*: *partition-rhs*)
done

We can always find a division that is fine wrt any gauge

lemma *partition-exists*:
[| *a* ≤ *b*; *gauge*(%*x*. *a* ≤ *x* & *x* ≤ *b*) *g* |]
==> ∃ *D p*. *tpart*(*a*,*b*) (*D*,*p*) & *fine g* (*D*,*p*)
apply (*cut-tac P = %*(*u*,*v*). *a* ≤ *u* & *v* ≤ *b* −−>

```
                (∃ D p. tpart (u,v) (D,p) & fine (g) (D,p))
      in lemma-BOLZANO2)
apply safe
apply (blast intro: order-trans)+
apply (auto intro: partition-append)
apply (case-tac a ≤ x & x ≤ b)
apply (rule-tac [2] x = 1 in exI, auto)
apply (rule-tac x = g x in exI)
apply (auto simp add: gauge-def)
apply (rule-tac x = %n. if n = 0 then aa else ba in exI)
apply (rule-tac x = %n. if n = 0 then x else ba in exI)
apply (auto simp add: tpart-def fine-def)
done
```

Lemmas about combining gauges

```
lemma gauge-min:
    [| gauge(E) g1; gauge(E) g2 |]
     ==> gauge(E) (%x. if g1(x) < g2(x) then g1(x) else g2(x))
by (simp add: gauge-def)

lemma fine-min:
     fine (%x. if g1(x) < g2(x) then g1(x) else g2(x)) (D,p)
      ==> fine(g1) (D,p) & fine(g2) (D,p)
by (auto simp add: fine-def split: split-if-asm)
```

The integral is unique if it exists

```
lemma Integral-unique:
   [| a ≤ b; Integral(a,b) f k1; Integral(a,b) f k2 |] ==> k1 = k2
apply (simp add: Integral-def)
apply (drule-tac x = |k1 - k2| /2 in spec)+
apply auto
apply (drule gauge-min, assumption)
apply (drule-tac g = %x. if g x < ga x then g x else ga x
      in partition-exists, assumption, auto)
apply (drule fine-min)
apply (drule spec)+
apply auto
apply (subgoal-tac |(rsum (D,p) f - k2) - (rsum (D,p) f - k1)| < |k1 - k2|)
apply arith
apply (drule add-strict-mono, assumption)
apply (auto simp only: left-distrib [symmetric] mult-2-right [symmetric]
            mult-less-cancel-right)
done

lemma Integral-zero [simp]: Integral(a,a) f 0
apply (auto simp add: Integral-def)
apply (rule-tac x = %x. 1 in exI)
apply (auto dest: partition-eq simp add: gauge-def tpart-def rsum-def)
done
```

lemma *sumr-partition-eq-diff-bounds* [*simp*]:
$(\sum n{=}0..{<}m.\ D\ (Suc\ n) - D\ n{::}real) = D(m) - D\ 0$
by (*induct m*, *auto*)

lemma *Integral-eq-diff-bounds*: $a \leq b$ ==> *Integral*(*a*,*b*) (%*x*. *1*) ($b - a$)
apply (*auto simp add*: *order-le-less rsum-def Integral-def*)
apply (*rule-tac x* = %*x*. $b - a$ **in** *exI*)
apply (*auto simp add*: *gauge-def abs-less-iff tpart-def partition*)
done

lemma *Integral-mult-const*: $a \leq b$ ==> *Integral*(*a*,*b*) (%*x*. *c*) ($c{*}(b - a)$)
apply (*auto simp add*: *order-le-less rsum-def Integral-def*)
apply (*rule-tac x* = %*x*. $b - a$ **in** *exI*)
apply (*auto simp add*: *setsum-right-distrib* [*symmetric*] *gauge-def abs-less-iff*
right-diff-distrib [*symmetric*] *partition tpart-def*)
done

lemma *Integral-mult*:
[| $a \leq b$; *Integral*(*a*,*b*) *f k* |] ==> *Integral*(*a*,*b*) (%*x*. $c * f\ x$) ($c * k$)
apply (*auto simp add*: *order-le-less*
dest: *Integral-unique* [*OF order-refl Integral-zero*])
apply (*auto simp add*: *rsum-def Integral-def setsum-right-distrib*[*symmetric*] *mult-assoc*)
apply (*rule-tac a2* = *c* **in** *abs-ge-zero* [*THEN order-le-imp-less-or-eq*, *THEN disjE*])
prefer *2* **apply** *force*
apply (*drule-tac x* = *e*/*abs c* **in** *spec*, *auto*)
apply (*simp add*: *zero-less-mult-iff divide-inverse*)
apply (*rule exI*, *auto*)
apply (*drule spec*)+
apply *auto*
apply (*rule-tac z1* = *inverse* (*abs c*) **in** *real-mult-less-iff1* [*THEN iffD1*])
apply (*auto simp add*: *abs-mult divide-inverse* [*symmetric*] *right-diff-distrib* [*symmetric*])
done

Fundamental theorem of calculus (Part I)

"Straddle Lemma" : Swartz and Thompson: AMM 95(7) 1988

lemma *choiceP*: $\forall x.\ P(x)$ --> $(\exists y.\ Q\ x\ y)$ ==> $\exists f.\ (\forall x.\ P(x)$ --> $Q\ x\ (f\ x))$
by (*insert bchoice* [*of Collect P Q*], *simp*)

lemma *strad1*:
$[\![\forall xa{::}real.\ xa \neq x \wedge |xa - x| < s \longrightarrow$
$|(f\ xa - f\ x)\ /\ (xa - x) - f'\ x| * 2 < e;$
$0 < e;\ a \leq x;\ x \leq b;\ 0 < s]\!]$
$\Longrightarrow \forall z.\ |z - x| < s$ -->$|f\ z - f\ x - f'\ x * (z - x)| * 2 \leq e * |z - x|$

apply *auto*
apply (*case-tac* $0 < |z - x|$)
prefer *2* **apply** (*simp add*: *zero-less-abs-iff*)
apply (*drule-tac* $x = z$ **in** *spec*)
apply (*rule-tac* $z1 = |inverse\ (z - x)|$
in *real-mult-le-cancel-iff2* [*THEN iffD1*])
apply *simp*
apply (*simp del*: *abs-inverse abs-mult add*: *abs-mult* [*symmetric*]
mult-assoc [*symmetric*])
apply (*subgoal-tac inverse* $(z - x) * (f\,z - f\,x - f'\,x * (z - x))$
$= (f\,z - f\,x) / (z - x) - f'\,x$)
apply (*simp add*: *abs-mult* [*symmetric*] *mult-ac diff-minus*)
apply (*subst mult-commute*)
apply (*simp add*: *left-distrib diff-minus*)
apply (*simp add*: *mult-assoc divide-inverse*)
apply (*simp add*: *left-distrib*)
done

lemma *lemma-straddle*:
$[|\ \forall x.\ a \leq x\ \&\ x \leq b\ \text{--}\!\text{>}\ DERIV\ f\ x\ :>\ f'(x);\ 0 < e\ |]$
$\text{==}\!\text{>}\ \exists g.\ gauge(\%x.\ a \leq x\ \&\ x \leq b)\ g\ \&$
$(\forall x\ u\ v.\ a \leq u\ \&\ u \leq x\ \&\ x \leq v\ \&\ v \leq b\ \&\ (v - u) < g(x)$
$\text{--}\!\text{>}\ |(f(v) - f(u)) - (f'(x) * (v - u))| \leq e * (v - u))$
apply (*simp add*: *gauge-def*)
apply (*subgoal-tac* $\forall x.\ a \leq x\ \&\ x \leq b\ \text{--}\!\text{>}$
$(\exists d > 0.\ \forall u\ v.\ u \leq x\ \&\ x \leq v\ \&\ (v - u) < d\ \text{--}\!\text{>}$
$|(f\ (v) - f\ (u)) - (f'\,(x) * (v - u))| \leq e * (v - u)))$
apply (*drule choiceP*, *auto*)
apply (*drule spec*, *auto*)
apply (*auto simp add*: *DERIV-iff2 LIM-def*)
apply (*drule-tac* $x = e/2$ **in** *spec*, *auto*)
apply (*frule strad1*, *assumption+*)
apply (*rule-tac* $x = s$ **in** *exI*, *auto*)
apply (*rule-tac* $x = u$ **and** $y = v$ **in** *linorder-cases*, *auto*)
apply (*rule-tac* $y = |(f\ (v) - f\ (x)) - (f'\,(x) * (v - x))| +$
$|(f\ (x) - f\ (u)) - (f'\,(x) * (x - u))|$
in *order-trans*)
apply (*rule abs-triangle-ineq* [*THEN* [*2*] *order-trans*])
apply (*simp add*: *right-diff-distrib*)
apply (*rule-tac* $t = e*\ (v - u)$ **in** *real-sum-of-halves* [*THEN subst*])
apply (*rule add-mono*)
apply (*rule-tac* $y = (e/2) * |v - x|$ **in** *order-trans*)
prefer *2* **apply** *simp*
apply (*erule-tac* [!] $V = \forall x'.\ x' \sim= x\ \&\ |x' - x| < s\ \text{--}\!\text{>}\ ?P\ x'$ **in** *thin-rl*)
apply (*drule-tac* $x = v$ **in** *spec*, *simp add*: *times-divide-eq*)
apply (*drule-tac* $x = u$ **in** *spec*, *auto*)
apply (*subgoal-tac* $|f\,u - f\,x - f'\,x * (u - x)| = |f\,x - f\,u - f'\,x * (x - u)|$)
apply (*rule order-trans*)
apply (*auto simp add*: *abs-le-iff*)

apply (*simp add*: *right-diff-distrib*)
done

lemma *FTC1*: [|$a \leq b$; $\forall x.\ a \leq x\ \&\ x \leq b$ −−> *DERIV f x* :> $f'(x)$ |]
==> *Integral*(*a*,*b*) f' ($f(b) - f(a)$))
apply (*drule order-le-imp-less-or-eq*, *auto*)
apply (*auto simp add*: *Integral-def*)
apply (*rule ccontr*)
apply (*subgoal-tac* $\forall e > 0.\ \exists g.$ *gauge* (%$x.\ a \leq x\ \&\ x \leq b$) *g* & ($\forall D\ p.$ *tpart* (*a*, *b*) (*D*, *p*) & *fine g* (*D*, *p*) −−> |*rsum* (*D*, *p*) f' − (*f b* − *f a*)| ≤ *e*))
apply (*rotate-tac 3*)
apply (*drule-tac x* = *e*/*2* **in** *spec*, *auto*)
apply (*drule spec*, *auto*)
apply ((*drule spec*)+, *auto*)
apply (*drule-tac e* = *ea*/ (*b* − *a*) **in** *lemma-straddle*)
apply (*auto simp add*: *zero-less-divide-iff*)
apply (*rule exI*)
apply (*auto simp add*: *tpart-def rsum-def*)
apply (*subgoal-tac* ($\sum$ *n*=*0*..<*psize D*. *f*(*D*(*Suc n*)) − *f*(*D n*)) = *f b* − *f a*)
prefer *2*
apply (*cut-tac D* = %*n*. *f* (*D n*) **and** *m* = *psize D*
in *sumr-partition-eq-diff-bounds*)
apply (*simp add*: *partition-lhs partition-rhs*)
apply (*drule sym*, *simp*)
apply (*simp* (*no-asm*) *add*: *setsum-subtractf*[*symmetric*])
apply (*rule setsum-abs* [*THEN order-trans*])
apply (*subgoal-tac ea* = ($\sum$ *n*=*0*..<*psize D*. (*ea* / (*b* − *a*)) ∗ (*D* (*Suc n*) − (*D n*))))
apply (*simp add*: *abs-minus-commute*)
apply (*rule-tac t* = *ea* **in** *ssubst*, *assumption*)
apply (*rule setsum-mono*)
apply (*rule-tac* [*2*] *setsum-right-distrib* [*THEN subst*])
apply (*auto simp add*: *partition-rhs partition-lhs partition-lb partition-ub fine-def*)
done

lemma *Integral-subst*: [| *Integral*(*a*,*b*) *f k1*; *k2*=*k1* |] ==> *Integral*(*a*,*b*) *f k2*
by *simp*

lemma *Integral-add*:
[| $a \leq b$; $b \leq c$; *Integral*(*a*,*b*) f' *k1*; *Integral*(*b*,*c*) f' *k2*;
$\forall x.\ a \leq x\ \&\ x \leq c$ −−> *DERIV f x* :> f' *x* |]
==> *Integral*(*a*,*c*) f' (*k1* + *k2*)
apply (*rule FTC1* [*THEN Integral-subst*], *auto*)
apply (*frule FTC1*, *auto*)
apply (*frule-tac a* = *b* **in** *FTC1*, *auto*)
apply (*drule-tac x* = *x* **in** *spec*, *auto*)
apply (*drule-tac* ?*k2.0* = *f b* − *f a* **in** *Integral-unique*)

apply (*drule-tac* [*3*] *?k2.0* = *f c* − *f b* **in** *Integral-unique*, *auto*)
done

lemma *partition-psize-Least*:
  partition(*a*,*b*) *D* ==> *psize D* = (*LEAST n*. *D*(*n*) = *b*)
apply (*auto intro*!: *Least-equality* [*symmetric*] *partition-rhs*)
apply (*auto dest*: *partition-ub-lt simp add*: *linorder-not-less* [*symmetric*])
done

lemma *lemma-partition-bounded*: *partition* (*a*, *c*) *D* ==> ~ (∃ *n*. *c* < *D*(*n*))
apply *safe*
apply (*drule-tac r* = *n* **in** *partition-ub*, *auto*)
done

lemma *lemma-partition-eq*:
  partition (*a*, *c*) *D* ==> *D* = (%*n*. *if D n* < *c then D n else c*)
apply (*rule ext*, *auto*)
apply (*auto dest*!: *lemma-partition-bounded*)
apply (*drule-tac x* = *n* **in** *spec*, *auto*)
done

lemma *lemma-partition-eq2*:
  partition (*a*, *c*) *D* ==> *D* = (%*n*. *if D n* ≤ *c then D n else c*)
apply (*rule ext*, *auto*)
apply (*auto dest*!: *lemma-partition-bounded*)
apply (*drule-tac x* = *n* **in** *spec*, *auto*)
done

lemma *partition-lt-Suc*:
  [| *partition*(*a*,*b*) *D*; *n* < *psize D* |] ==> *D n* < *D* (*Suc n*)
by (*auto simp add*: *partition*)

lemma *tpart-tag-eq*: *tpart*(*a*,*c*) (*D*,*p*) ==> *p* = (%*n*. *if D n* < *c then p n else c*)
apply (*rule ext*)
apply (*auto simp add*: *tpart-def*)
apply (*drule linorder-not-less* [*THEN iffD1*])
apply (*drule-tac r* = *Suc n* **in** *partition-ub*)
apply (*drule-tac x* = *n* **in** *spec*, *auto*)
done

69.1 Lemmas for Additivity Theorem of Gauge Integral

lemma *lemma-additivity1*:
  [| *a* ≤ *D n*; *D n* < *b*; *partition*(*a*,*b*) *D* |] ==> *n* < *psize D*
by (*auto simp add*: *partition linorder-not-less* [*symmetric*])

lemma *lemma-additivity2*: [| *a* ≤ *D n*; *partition*(*a*,*D n*) *D* |] ==> *psize D* ≤ *n*
apply (*rule ccontr*, *drule not-leE*)
apply (*frule partition* [*THEN iffD1*], *safe*)

apply (*frule-tac r = Suc n* **in** *partition-ub*)
apply (*auto dest*!: *spec*)
done

lemma *partition-eq-bound*:
[| *partition*(*a*,*b*) *D*; *psize D < m* |] ==> *D*(*m*) = *D*(*psize D*)
by (*auto simp add*: *partition*)

lemma *partition-ub2*: [| *partition*(*a*,*b*) *D*; *psize D < m* |] ==> *D*(*r*) ≤ *D*(*m*)
by (*simp add*: *partition partition-ub*)

lemma *tag-point-eq-partition-point*:
[| *tpart*(*a*,*b*) (*D*,*p*); *psize D* ≤ *m* |] ==> *p*(*m*) = *D*(*m*)
apply (*simp add*: *tpart-def*, *auto*)
apply (*drule-tac x = m* **in** *spec*)
apply (*auto simp add*: *partition-rhs2*)
done

lemma *partition-lt-cancel*: [| *partition*(*a*,*b*) *D*; *D m < D n* |] ==> *m < n*
apply (*cut-tac less-linear* [*of n psize D*], *auto*)
apply (*cut-tac less-linear* [*of m n*])
apply (*cut-tac less-linear* [*of m psize D*])
apply (*auto dest*: *partition-gt*)
apply (*drule-tac n = m* **in** *partition-lt-gen*, *auto*)
apply (*frule partition-eq-bound*)
apply (*drule-tac* [*2*] *partition-gt*, *auto*)
apply (*metis linear not-less partition-rhs partition-rhs2*)
apply (*metis lemma-additivity1 order-less-trans partition-eq-bound partition-lb partition-rhs*)
done

lemma *lemma-additivity4-psize-eq*:
[| *a* ≤ *D n*; *D n < b*; *partition* (*a*, *b*) *D* |]
==> *psize* (%*x*. *if D x < D n then D*(*x*) *else D n*) = *n*
apply (*frule* (*2*) *lemma-additivity1*)
apply (*rule psize-unique*, *auto*)
apply (*erule partition-lt-Suc*, *erule* (*1*) *less-trans*)
apply (*erule notE*)
apply (*erule* (*1*) *partition-lt-gen*, *erule less-imp-le*)
apply (*drule* (*1*) *partition-lt-cancel*, *simp*)
done

lemma *lemma-psize-left-less-psize*:
partition (*a*, *b*) *D*
==> *psize* (%*x*. *if D x < D n then D*(*x*) *else D n*) ≤ *psize D*
apply (*frule-tac r = n* **in** *partition-ub*)
apply (*drule-tac x = D n* **in** *order-le-imp-less-or-eq*)
apply (*auto simp add*: *lemma-partition-eq* [*symmetric*])
apply (*frule-tac r = n* **in** *partition-lb*)
apply (*drule* (*2*) *lemma-additivity4-psize-eq*)

apply (*rule ccontr*, *auto*)
apply (*frule-tac not-leE* [*THEN* [*2*] *partition-eq-bound*])
apply (*auto simp add*: *partition-rhs*)
done

lemma *lemma-psize-left-less-psize2*:
[| *partition*(*a*,*b*) *D*; *na* < *psize* (%*x*. *if D x* < *D n then D*(*x*) *else D n*) |]
==> *na* < *psize D*
by (*erule lemma-psize-left-less-psize* [*THEN* [*2*] *less-le-trans*])

lemma *lemma-additivity3*:
[| *partition*(*a*,*b*) *D*; *D na* < *D n*; *D n* < *D* (*Suc na*);
n < *psize D* |]
==> *False*
by (*metis not-less-eq partition-lt-cancel real-of-nat-less-iff*)

lemma *psize-const* [*simp*]: *psize* (%*x*. *k*) = *0*
by (*auto simp add*: *psize-def*)

lemma *lemma-additivity3a*:
[| *partition*(*a*,*b*) *D*; *D na* < *D n*; *D n* < *D* (*Suc na*);
na < *psize D* |]
==> *False*
apply (*frule-tac m* = *n* **in** *partition-lt-cancel*)
apply (*auto intro*: *lemma-additivity3*)
done

lemma *better-lemma-psize-right-eq1*:
[| *partition*(*a*,*b*) *D*; *D n* < *b* |] ==> *psize* (%*x*. *D* (*x* + *n*)) ≤ *psize D* − *n*
apply (*simp add*: *psize-def* [*of* (%*x*. *D* (*x* + *n*))])
apply (*rule-tac a* = *psize D* − *n* **in** *someI2*, *auto*)
apply (*simp add*: *partition less-diff-conv*)
apply (*simp add*: *le-diff-conv partition-rhs2 split*: *nat-diff-split*)
apply (*drule-tac x* = *psize D* − *n* **in** *spec*, *auto*)
apply (*frule partition-rhs*, *safe*)
apply (*frule partition-lt-cancel*, *assumption*)
apply (*drule partition* [*THEN iffD1*], *safe*)
apply (*subgoal-tac* ~ *D* (*psize D* − *n* + *n*) < *D* (*Suc* (*psize D* − *n* + *n*)))
apply *blast*
apply (*drule-tac x* = *Suc* (*psize D*) **and** *P*=%*n*. *?P n* ⟶ *D n* = *D* (*psize D*)
in *spec*)
apply *simp*
done

lemma *psize-le-n*: *partition* (*a*, *D n*) *D* ==> *psize D* ≤ *n*
apply (*rule ccontr*, *drule not-leE*)
apply (*frule partition-lt-Suc*, *assumption*)

apply (*frule-tac r = Suc n* **in** *partition-ub*, *auto*)
done

lemma *better-lemma-psize-right-eq1a*:
    partition(*a*,*D n*) *D* ==> *psize* (%*x*. *D* (*x* + *n*)) ≤ *psize D* − *n*
apply (*simp add*: *psize-def* [*of* (%*x*. *D* (*x* + *n*))])
apply (*rule-tac a = psize D − n* **in** *someI2*, *auto*)
 apply (*simp add*: *partition less-diff-conv*)
apply (*simp add*: *le-diff-conv*)
apply (*case-tac psize D ≤ n*)
 apply (*force intro*: *partition-rhs2*)
apply (*simp add*: *partition linorder-not-le*)
apply (*rule ccontr*, *drule not-leE*)
apply (*frule psize-le-n*)
apply (*drule-tac x = psize D − n* **in** *spec*, *simp*)
apply (*drule partition* [*THEN iffD1*], *safe*)
apply (*drule-tac x = Suc n* **and** *P*=%*na*. *?s* ≤ *na* ⟶ *D na* = *D n* **in** *spec*, *auto*)
done

lemma *better-lemma-psize-right-eq*:
    partition(*a*,*b*) *D* ==> *psize* (%*x*. *D* (*x* + *n*)) ≤ *psize D* − *n*
apply (*frule-tac r1 = n* **in** *partition-ub* [*THEN order-le-imp-less-or-eq*])
apply (*blast intro*: *better-lemma-psize-right-eq1a better-lemma-psize-right-eq1*)
done

lemma *lemma-psize-right-eq1*:
    [| *partition*(*a*,*b*) *D*; *D n* < *b* |] ==> *psize* (%*x*. *D* (*x* + *n*)) ≤ *psize D*
apply (*simp add*: *psize-def* [*of* (%*x*. *D* (*x* + *n*))])
apply (*rule-tac a = psize D − n* **in** *someI2*, *auto*)
 apply (*simp add*: *partition less-diff-conv*)
apply (*subgoal-tac n ≤ psize D*)
 apply (*simp add*: *partition le-diff-conv*)
apply (*rule ccontr*, *drule not-leE*)
apply (*drule-tac less-imp-le* [*THEN* [*2*] *partition-rhs2*], *assumption*, *simp*)
apply (*drule-tac x = psize D* **in** *spec*)
apply (*simp add*: *partition*)
done

lemma *lemma-psize-right-eq1a*:
    partition(*a*,*D n*) *D* ==> *psize* (%*x*. *D* (*x* + *n*)) ≤ *psize D*
apply (*simp add*: *psize-def* [*of* (%*x*. *D* (*x* + *n*))])
apply (*rule-tac a = psize D − n* **in** *someI2*, *auto*)
 apply (*simp add*: *partition less-diff-conv*)
apply (*case-tac psize D ≤ n*)
 apply (*force intro*: *partition-rhs2 simp add*: *le-diff-conv*)
apply (*simp add*: *partition le-diff-conv*)
apply (*rule ccontr*, *drule not-leE*)
apply (*drule-tac x = psize D* **in** *spec*)

apply (*simp add*: *partition*)
done

lemma *lemma-psize-right-eq*:
[| *partition*(*a*,*b*) *D* |] ==> *psize* (%*x*. *D* (*x* + *n*)) ≤ *psize D*
apply (*frule-tac r1* = *n* **in** *partition-ub* [*THEN order-le-imp-less-or-eq*])
apply (*blast intro*: *lemma-psize-right-eq1a lemma-psize-right-eq1*)
done

lemma *tpart-left1*:
[| *a* ≤ *D n*; *tpart* (*a*, *b*) (*D*, *p*) |]
==> *tpart*(*a*, *D n*) (%*x*. *if D x* < *D n then D*(*x*) *else D n*,
%*x*. *if D x* < *D n then p*(*x*) *else D n*)
apply (*frule-tac r* = *n* **in** *tpart-partition* [*THEN partition-ub*])
apply (*drule-tac x* = *D n* **in** *order-le-imp-less-or-eq*)
apply (*auto simp add*: *tpart-partition* [*THEN lemma-partition-eq*, *symmetric*] *tpart-tag-eq* [*symmetric*])
apply (*frule-tac tpart-partition* [*THEN* [*3*] *lemma-additivity1*])
apply (*auto simp add*: *tpart-def*)
apply (*drule-tac* [*2*] *linorder-not-less* [*THEN iffD1*, *THEN order-le-imp-less-or-eq*], *auto*)
prefer *3* **apply** (*drule-tac x*=*na* **in** *spec*, *arith*)
prefer *2* **apply** (*blast dest*: *lemma-additivity3*)
apply (*frule* (*2*) *lemma-additivity4-psize-eq*)
apply (*rule partition* [*THEN iffD2*])
apply (*frule partition* [*THEN iffD1*])
apply *safe*
apply (*auto simp add*: *partition-lt-gen*)
apply (*drule* (*1*) *partition-lt-cancel*, *arith*)
done

lemma *fine-left1*:
[| *a* ≤ *D n*; *tpart* (*a*, *b*) (*D*, *p*); *gauge* (%*x*. *a* ≤ *x* & *x* ≤ *D n*) *g*;
fine (%*x*. *if x* < *D n then min* (*g x*) ((*D n* − *x*)/ *2*)
else if x = *D n then min* (*g* (*D n*)) (*ga* (*D n*))
else min (*ga x*) ((*x* − *D n*)/ *2*)) (*D*, *p*) |]
==> *fine g*
(%*x*. *if D x* < *D n then D*(*x*) *else D n*,
%*x*. *if D x* < *D n then p*(*x*) *else D n*)
apply (*auto simp add*: *fine-def tpart-def gauge-def*)
apply (*frule-tac* [!] *na*=*na* **in** *lemma-psize-left-less-psize2*)
apply (*drule-tac* [!] *x* = *na* **in** *spec*, *auto*)
apply (*drule-tac* [!] *x* = *na* **in** *spec*, *auto*)
apply (*auto dest*: *lemma-additivity3a simp add*: *split-if-asm*)
done

lemma *tpart-right1*:
[| *a* ≤ *D n*; *tpart* (*a*, *b*) (*D*, *p*) |]
==> *tpart*(*D n*, *b*) (%*x*. *D*(*x* + *n*),%*x*. *p*(*x* + *n*))

apply (*simp add*: *tpart-def partition-def*, *safe*)
apply (*rule-tac* $x = N - n$ **in** *exI*, *auto*)
done

lemma *fine-right1*:
[| $a \leq D\ n$; *tpart* (a, b) (D, p); *gauge* (%x. $D\ n \leq x$ & $x \leq b$) *ga*;
fine (%x. *if* $x < D\ n$ *then min* (*g* x) (($D\ n - x$)/ 2)
else if $x = D\ n$ *then min* (*g* ($D\ n$)) (*ga* ($D\ n$))
else min (*ga* x) (($x - D\ n$)/ 2)) (D, p) |]
==> *fine ga* (%x. $D(x + n)$,%x. $p(x + n)$)
apply (*auto simp add*: *fine-def gauge-def*)
apply (*drule-tac* $x = na + n$ **in** *spec*)
apply (*frule-tac* $n = n$ **in** *tpart-partition* [*THEN better-lemma-psize-right-eq*], *auto*)
apply (*simp add*: *tpart-def*, *safe*)
apply (*subgoal-tac* $D\ n \leq p\ (na + n)$)
apply (*drule-tac* $y = p\ (na + n)$ **in** *order-le-imp-less-or-eq*)
apply *safe*
apply (*simp split*: *split-if-asm*, *simp*)
apply (*drule less-le-trans*, *assumption*)
apply (*rotate-tac 5*)
apply (*drule-tac* $x = na + n$ **in** *spec*, *safe*)
apply (*rule-tac* y=$D\ (na + n)$ **in** *order-trans*)
apply (*case-tac* $na = 0$, *auto*)
apply (*erule partition-lt-gen* [*THEN order-less-imp-le*])
apply *arith*
apply *arith*
done

lemma *rsum-add*: *rsum* (D, p) (%x. $f\ x + g\ x$) = *rsum* (D, p) f + *rsum*(D, p) *g*
by (*simp add*: *rsum-def setsum-addf left-distrib*)

Bartle/Sherbert: Theorem 10.1.5 p. 278

lemma *Integral-add-fun*:
[| $a \leq b$; *Integral*(a,b) *f k1*; *Integral*(a,b) *g k2* |]
==> *Integral*(a,b) (%x. $f\ x + g\ x$) (*k1* + *k2*)
apply (*simp add*: *Integral-def*, *auto*)
apply ((*drule-tac* $x = e/2$ **in** *spec*)+)
apply *auto*
apply (*drule gauge-min*, *assumption*)
apply (*rule-tac* $x =$ (%x. *if ga* $x <$ *gaa* x *then ga* x *else gaa* x) **in** *exI*)
apply *auto*
apply (*drule fine-min*)
apply ((*drule spec*)+, *auto*)
apply (*drule-tac* $a = |rsum\ (D, p)\ f - k1| * 2$ **and** $c = |rsum\ (D, p)\ g - k2| * 2$ **in** *add-strict-mono*, *assumption*)
apply (*auto simp only*: *rsum-add left-distrib* [*symmetric*]
mult-2-right [*symmetric*] *real-mult-less-iff1*)
done

lemma *partition-lt-gen2*:
[| *partition*(*a*,*b*) *D*; *r* < *psize D* |] ==> 0 < *D* (*Suc r*) − *D r*
by (*auto simp add*: *partition*)

lemma *lemma-Integral-le*:
[| ∀ *x*. *a* ≤ *x* & *x* ≤ *b* −−> *f x* ≤ *g x*;
tpart(*a*,*b*) (*D*,*p*)
|] ==> ∀ *n* ≤ *psize D*. *f* (*p n*) ≤ *g* (*p n*)
apply (*simp add*: *tpart-def*)
apply (*auto*, *frule partition* [*THEN iffD1*], *auto*)
apply (*drule-tac x* = *p n* **in** *spec*, *auto*)
apply (*case-tac n* = *0*, *simp*)
apply (*rule partition-lt-gen* [*THEN order-less-le-trans*, *THEN order-less-imp-le*], *auto*)
apply (*drule le-imp-less-or-eq*, *auto*)
apply (*drule-tac* [*2*] *x* = *psize D* **in** *spec*, *auto*)
apply (*drule-tac r* = *Suc n* **in** *partition-ub*)
apply (*drule-tac x* = *n* **in** *spec*, *auto*)
done

lemma *lemma-Integral-rsum-le*:
[| ∀ *x*. *a* ≤ *x* & *x* ≤ *b* −−> *f x* ≤ *g x*;
tpart(*a*,*b*) (*D*,*p*)
|] ==> *rsum*(*D*,*p*) *f* ≤ *rsum*(*D*,*p*) *g*
apply (*simp add*: *rsum-def*)
apply (*auto intro*!: *setsum-mono dest*: *tpart-partition* [*THEN partition-lt-gen2*]
dest!: *lemma-Integral-le*)
done

lemma *Integral-le*:
[| *a* ≤ *b*;
∀ *x*. *a* ≤ *x* & *x* ≤ *b* −−> *f*(*x*) ≤ *g*(*x*);
Integral(*a*,*b*) *f k1*; *Integral*(*a*,*b*) *g k2*
|] ==> *k1* ≤ *k2*
apply (*simp add*: *Integral-def*)
apply (*rotate-tac 2*)
apply (*drule-tac x* = |*k1* − *k2*| */2* **in** *spec*)
apply (*drule-tac x* = |*k1* − *k2*| */2* **in** *spec*, *auto*)
apply (*drule gauge-min*, *assumption*)
apply (*drule-tac g* = %*x*. *if ga x* < *gaa x then ga x else gaa x*
in *partition-exists*, *assumption*, *auto*)
apply (*drule fine-min*)
apply (*drule-tac x* = *D* **in** *spec*, *drule-tac x* = *D* **in** *spec*)
apply (*drule-tac x* = *p* **in** *spec*, *drule-tac x* = *p* **in** *spec*, *auto*)
apply (*frule lemma-Integral-rsum-le*, *assumption*)
apply (*subgoal-tac* |(*rsum* (*D*,*p*) *f* − *k1*) − (*rsum* (*D*,*p*) *g* − *k2*)| < |*k1* − *k2*|)
apply *arith*
apply (*drule add-strict-mono*, *assumption*)

apply (*auto simp only*: *left-distrib* [*symmetric*] *mult-2-right* [*symmetric*]
real-mult-less-iff1)
done

lemma *Integral-imp-Cauchy*:
(∃ *k*. *Integral*(*a*,*b*) *f k*) ==>
(∀ *e* > *0*. ∃ *g*. *gauge* (%*x*. *a* ≤ *x* & *x* ≤ *b*) *g* &
(∀ *D1 D2 p1 p2*.
tpart(*a*,*b*) (*D1*, *p1*) & *fine g* (*D1*,*p1*) &
tpart(*a*,*b*) (*D2*, *p2*) & *fine g* (*D2*,*p2*) −−>
|*rsum*(*D1*,*p1*) *f* − *rsum*(*D2*,*p2*) *f*| < *e*))
apply (*simp add*: *Integral-def*, *auto*)
apply (*drule-tac x* = *e*/*2* **in** *spec*, *auto*)
apply (*rule exI*, *auto*)
apply (*frule-tac x* = *D1* **in** *spec*)
apply (*frule-tac x* = *D2* **in** *spec*)
apply ((*drule spec*)+, *auto*)
apply (*erule-tac V* = *0* < *e* **in** *thin-rl*)
apply (*drule add-strict-mono*, *assumption*)
apply (*auto simp only*: *left-distrib* [*symmetric*] *mult-2-right* [*symmetric*]
real-mult-less-iff1)
done

lemma *Cauchy-iff2*:
Cauchy X =
(∀ *j*. (∃ *M*. ∀ *m* ≥ *M*. ∀ *n* ≥ *M*. |*X m* − *X n*| < *inverse*(*real* (*Suc j*))))
apply (*simp add*: *Cauchy-def*, *auto*)
apply (*drule reals-Archimedean*, *safe*)
apply (*drule-tac x* = *n* **in** *spec*, *auto*)
apply (*rule-tac x* = *M* **in** *exI*, *auto*)
apply (*drule-tac x* = *m* **in** *spec*, *simp*)
apply (*drule-tac x* = *na* **in** *spec*, *auto*)
done

lemma *partition-exists2*:
[| *a* ≤ *b*; ∀ *n*. *gauge* (%*x*. *a* ≤ *x* & *x* ≤ *b*) (*fa n*) |]
==> ∀ *n*. ∃ *D p*. *tpart* (*a*, *b*) (*D*, *p*) & *fine* (*fa n*) (*D*, *p*)
by (*blast dest*: *partition-exists*)

lemma *monotonic-anti-derivative*:
fixes *f g* :: *real* => *real* **shows**
[| *a* ≤ *b*; ∀ *c*. *a* ≤ *c* & *c* ≤ *b* −−> *f' c* ≤ *g' c*;
∀ *x*. *DERIV f x* :> *f' x*; ∀ *x*. *DERIV g x* :> *g' x* |]
==> *f b* − *f a* ≤ *g b* − *g a*
apply (*rule Integral-le*, *assumption*)
apply (*auto intro*: *FTC1*)
done

end

70 Complex-Main: Comprehensive Complex Theory

theory *Complex-Main*
imports
Main
Real
Complex
Log
Ln
Taylor
Integration
begin

end

References

[1] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.