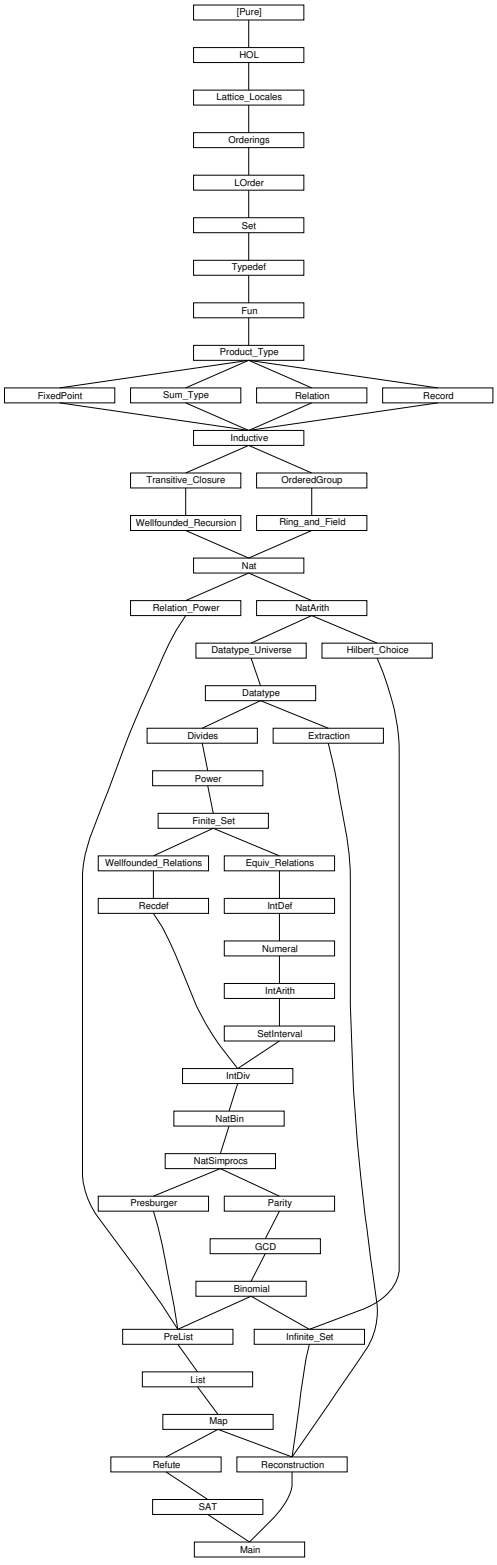
Isabelle/HOL — Higher-Order Logic

October 1, 2005

Contents


[Pure]
HOL
Lattice_Locales
Orderings
LOrder
Set
Typedef
Fun
Product_Type
FixedPoint
Sum_Type
Relation
Record
Inductive
Transitive_Closure
OrderedGroup
Wellfounded_Recursion
Ring_and_Field
Nat
Relation_Power
NatArith
Datatype_Universe
Hilbert_Choice
Datatype
Divides
Extraction
Power
Finite_Set
Wellfounded_Relations
Equiv_Relations
Recdef
IntDef
Numeral
IntArith
SetInterval
IntDiv
NatBin
NatSimprocs
Presburger
Parity
GCD
Binomial
PreList
Infinite_Set
List
Map
Refute
Reconstruction
SAT
Main

1 HOL: The basis of Higher-Order Logic

theory *HOL*
imports *CPure*
uses (*cladata.ML*) (*blastdata.ML*) (*simpdata.ML*) (*eqrule-HOL-data.ML*)
(~~/*src/Provers/eqsubst.ML*)

begin

1.1 Primitive logic

1.1.1 Core syntax

classes *type*
defaultsort *type*

global

typedecl *bool*

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

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

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

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*

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

local

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

1.1.2 Additional concrete syntax

nonterminals
letbinds letbind
case-syn cases-syn

syntax
-not-equal :: ['a, 'a] => *bool* (**infixl** ~= *50*)
-The :: [*pttrn*, *bool*] => 'a (("*3*THE -./ -) [*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
x ~= *y* == ~ (*x* = *y*)
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 (**output**)
= :: ['a, 'a] => *bool* (**infix** *50*)
-not-equal :: ['a, 'a] => *bool* (**infix** ~= *50*)

syntax (*xsymbols*)
Not :: *bool* => *bool* (¬ - [*40*] *40*)
op & :: [*bool*, *bool*] => *bool* (**infixr** ∧ *35*)
op | :: [*bool*, *bool*] => *bool* (**infixr** ∨ *30*)
op −−> :: [*bool*, *bool*] => *bool* (**infixr** ⟶ *25*)
-not-equal :: ['a, 'a] => *bool* (**infix** ≠ *50*)
ALL :: [*idts*, *bool*] => *bool* (("*3*∀ -./ -) [*0*, *10*] *10*)
EX :: [*idts*, *bool*] => *bool* (("*3*∃ -./ -) [*0*, *10*] *10*)
EX! :: [*idts*, *bool*] => *bool* (("*3*∃!-./ -) [*0*, *10*] *10*)
-case1 :: ['a, 'b] => *case-syn* (("*2*- ⇒/ -) *10*)

syntax (*xsymbols* **output**)

-not-equal :: [*'a*, *'a*] => *bool* (**infix** ≠ *50*)

syntax (*HTML* **output**)

-not-equal	:: [*'a*, *'a*] => *bool*	(**infix** ≠ *50*)
Not	:: *bool* => *bool*	(¬ - [*40*] *40*)
op &	:: [*bool*, *bool*] => *bool*	(**infixr** ∧ *35*)
op |	:: [*bool*, *bool*] => *bool*	(**infixr** ∨ *30*)
-not-equal	:: [*'a*, *'a*] => *bool*	(**infix** ≠ *50*)
ALL	:: [*idts*, *bool*] => *bool*	((*3*∀ -./ -) [*0*, *10*] *10*)
EX	:: [*idts*, *bool*] => *bool*	((*3*∃ -./ -) [*0*, *10*] *10*)
EX!	:: [*idts*, *bool*] => *bool*	((*3*∃!-./ -) [*0*, *10*] *10*)

syntax (*HOL*)

ALL	:: [*idts*, *bool*] => *bool*	((*3*! -./ -) [*0*, *10*] *10*)
EX	:: [*idts*, *bool*] => *bool*	((*3?* -./ -) [*0*, *10*] *10*)
EX!	:: [*idts*, *bool*] => *bool*	((*3?*! -./ -) [*0*, *10*] *10*)

1.1.3 Axioms and basic definitions

axioms

eq-reflection: $(x{=}y) \Longrightarrow (x{=}{=}y)$

refl: $t = (t::'a)$

ext: $(!!x::'a.\ (f\ x ::'b) = g\ x) \Longrightarrow (\%x.\ f\ x) = (\%x.\ g\ x)$

— 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: $(P \Longrightarrow Q) \Longrightarrow P{-}{-}{>}Q$

mp: $[|\ P{-}{-}{>}Q;\ \ P\ |] \Longrightarrow Q$

Thanks to Stephan Merz

theorem *subst*:
assumes *eq*: $s = t$ **and** *p*: $P(s)$
shows $P(t::'a)$
proof −
from *eq* **have** *meta*: $s \equiv t$
by (*rule eq-reflection*)
from *p* **show** *?thesis*
by (*unfold meta*)
qed

defs

True-def:	*True*	$== ((\%x::bool.\ x) = (\%x.\ x))$
All-def:	*All*(*P*)	$== (P = (\%x.\ True))$
Ex-def:	*Ex*(*P*)	$== !Q.\ (!x.\ P\ x\ {-}{-}{>}\ Q)\ {-}{-}{>}\ Q$

False-def: *False* == (!*P*. *P*)
not-def: ~ *P* == *P*−−>*False*
and-def: *P* & *Q* == !*R*. (*P*−−>*Q*−−>*R*) −−> *R*
or-def: *P* | *Q* == !*R*. (*P*−−>*R*) −−> (*Q*−−>*R*) −−> *R*
Ex1-def: *Ex1*(*P*) == *?* *x*. *P*(*x*) & (! *y*. *P*(*y*) −−> *y*=*x*)

axioms
iff: (*P*−−>*Q*) −−> (*Q*−−>*P*) −−> (*P*=*Q*)
True-or-False: (*P*=*True*) | (*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
arbitrary

1.1.4 Generic algebraic operations

axclass *zero* < *type*
axclass *one* < *type*
axclass *plus* < *type*
axclass *minus* < *type*
axclass *times* < *type*
axclass *inverse* < *type*

global

consts
0 :: ′*a*::*zero* (*0*)
1 :: ′*a*::*one* (*1*)
+ :: [′*a*::*plus*, ′*a*] => ′*a* (**infixl** *65*)
− :: [′*a*::*minus*, ′*a*] => ′*a* (**infixl** *65*)
uminus :: [′*a*::*minus*] => ′*a* (− - [*81*] *80*)
∗ :: [′*a*::*times*, ′*a*] => ′*a* (**infixl** *70*)

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

local

typed-print-translation ⟪
let

fun tr' c = (c, fn show-sorts => fn T => fn ts =>
if 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 [tr' 0, tr' 1] end;
⟩⟩ — show types that are presumably too general

consts
abs :: *'a::minus => 'a*
inverse :: *'a::inverse => 'a*
divide :: *['a::inverse, 'a] => 'a* (**infixl** *'/ 70*)

syntax (*xsymbols*)
abs :: *'a::minus => 'a* (|-|)
syntax (*HTML* **output**)
abs :: *'a::minus => 'a* (|-|)

1.2 Equality

lemma *sym*: *s=t ==> t=s*
apply (*erule subst*)
apply (*rule refl*)
done

lemmas *ssubst = sym [THEN subst, standard]*

lemma *trans*: *[| r=s; s=t |] ==> r=t*
apply (*erule subst , assumption*)
done

lemma *def-imp-eq*: **assumes** *meq*: *A == B* **shows** *A = B*
apply (*unfold meq*)
apply (*rule refl*)
done

lemma *box-equals*: *[| a=b; a=c; b=d |] ==> c=d*
apply (*rule trans*)
apply (*rule trans*)
apply (*rule sym*)
apply *assumption+*
done

For calculational reasoning:

lemma *forw-subst*: *a = b ==> P b ==> P a*
by (*rule ssubst*)

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

1.3 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*: $[\!|\ f = g;\ (x::'a) = y\ |\!] \Longrightarrow f(x) = g(y)$
apply (*erule subst*)+
apply (*rule refl*)
done

1.4 Equality of booleans – iff

lemma *iffI*: **assumes** *prems*: $P \Longrightarrow Q\ Q \Longrightarrow P$ **shows** $P=Q$
apply (*iprover intro*: *iff* [*THEN mp*, *THEN mp*] *impI prems*)
done

lemma *iffD2*: $[\!|\ P=Q;\ Q\ |\!] \Longrightarrow P$
apply (*erule ssubst*)
apply *assumption*
done

lemma *rev-iffD2*: $[\!|\ Q;\ P=Q\ |\!] \Longrightarrow P$
apply (*erule iffD2*)
apply *assumption*
done

lemmas *iffD1* = *sym* [*THEN iffD2*, *standard*]
lemmas *rev-iffD1* = *sym* [*THEN* [*2*] *rev-iffD2*, *standard*]

lemma *iffE*:
assumes *major*: $P=Q$
and *minor*: $[\!|\ P \longrightarrow Q;\ Q \longrightarrow P\ |\!] \Longrightarrow R$

shows *R*
by (*iprover intro*: *minor impI major* [*THEN iffD2*] *major* [*THEN iffD1*])

1.5 True

lemma *TrueI*: *True*
apply (*unfold True-def*)
apply (*rule refl*)
done

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

lemma *eqTrueE*: *P*=*True* ==> *P*
apply (*erule iffD2*)
apply (*rule TrueI*)
done

1.6 Universal quantifier

lemma *allI*: **assumes** *p*: !!*x*::'*a*. *P*(*x*) **shows** *ALL x*. *P*(*x*)
apply (*unfold All-def*)
apply (*iprover intro*: *ext eqTrueI p*)
done

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*])

1.7 False

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

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

1.8 Negation

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

lemmas *notI2* = *notE* [*THEN notI*, *standard*]

1.9 Implication

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

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

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

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

lemma *not-sym*: *t* ~= *s* ==> *s* ~= *t*
apply (*erule contrapos-nn*)
apply (*erule sym*)
done

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

1.10 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

1.11 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 *prems*: *P P* ==> *Q* **shows** *P* & *Q*
by (*iprover intro*: *conjI prems*)

1.12 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*])

1.13 Classical logic

lemma *classical*:
assumes *prem*: ~*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*: ~*R* ==> ~*P*
  shows *R*
apply (*rule ccontr*)
apply (*erule notE* [*OF premnot premp*])
done

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

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

1.14 Unique existence

lemma *ex1I*:
  assumes *prems*: *P a* !!*x*. *P*(*x*) ==> *x*=*a*
  shows *EX*! *x*. *P*(*x*)
by (*unfold Ex1-def*, *iprover intro*: *prems 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

1.15 THE: definite description operator

lemma *the-equality*:
assumes *prema*: $P\ a$
and *premx*: $!!x.\ P\ x \Longrightarrow x=a$
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** $!!x.\ P\ x \Longrightarrow x=a$
shows $P\ (THE\ x.\ P\ x)$
by (*iprover intro*: *prems the-equality* [*THEN ssubst*])

lemma *theI'*: $EX!\ x.\ P\ x \Longrightarrow P\ (THE\ x.\ P\ x)$
apply (*erule ex1E*)
apply (*erule theI*)
apply (*erule allE*)
apply (*erule mp*)
apply *assumption*
done

lemma *theI2*:
assumes $P\ a$ $!!x.\ P\ x \Longrightarrow x=a$ $!!x.\ P\ x \Longrightarrow Q\ x$
shows $Q\ (THE\ x.\ P\ x)$
by (*iprover intro*: *prems theI*)

lemma *the1-equality*: $[|\ EX!x.\ P\ x;\ P\ a\ |] \Longrightarrow (THE\ x.\ P\ x) = a$
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

1.16 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
assumes ~*Q* ==> *P* **shows** *P*|*Q*
apply (*rule classical*)
apply (*iprover intro*: *prems 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-thm*:
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*: *prems exI allI notI notE* [*of* ∃ *x*. *P x*])
done

1.17 Theory and package setup

ML
⟪
val eq-reflection = thm eq-reflection
val refl = thm refl
val subst = thm subst
val ext = thm ext
val impI = thm impI
val mp = thm mp
val True-def = thm True-def
val All-def = thm All-def
val Ex-def = thm Ex-def
val False-def = thm False-def
val not-def = thm not-def
val and-def = thm and-def
val or-def = thm or-def
val Ex1-def = thm Ex1-def
val iff = thm iff
val True-or-False = thm True-or-False
val Let-def = thm Let-def
val if-def = thm if-def
val sym = thm sym
val ssubst = thm ssubst
val trans = thm trans
val def-imp-eq = thm def-imp-eq
val box-equals = thm box-equals
val fun-cong = thm fun-cong
val arg-cong = thm arg-cong
val cong = thm cong
val iffI = thm iffI
val iffD2 = thm iffD2
val rev-iffD2 = thm rev-iffD2
val iffD1 = thm iffD1
val rev-iffD1 = thm rev-iffD1
val iffE = thm iffE
val TrueI = thm TrueI
val eqTrueI = thm eqTrueI
val eqTrueE = thm eqTrueE

val allI = thm allI
val spec = thm spec
val allE = thm allE
val all-dupE = thm all-dupE
val FalseE = thm FalseE
val False-neq-True = thm False-neq-True
val notI = thm notI
val False-not-True = thm False-not-True
val True-not-False = thm True-not-False
val notE = thm notE
val notI2 = thm notI2
val impE = thm impE
val rev-mp = thm rev-mp
val contrapos-nn = thm contrapos-nn
val contrapos-pn = thm contrapos-pn
val not-sym = thm not-sym
val rev-contrapos = thm rev-contrapos
val exI = thm exI
val exE = thm exE
val conjI = thm conjI
val conjunct1 = thm conjunct1
val conjunct2 = thm conjunct2
val conjE = thm conjE
val context-conjI = thm context-conjI
val disjI1 = thm disjI1
val disjI2 = thm disjI2
val disjE = thm disjE
val classical = thm classical
val ccontr = thm ccontr
val rev-notE = thm rev-notE
val notnotD = thm notnotD
val contrapos-pp = thm contrapos-pp
val ex1I = thm ex1I
val ex-ex1I = thm ex-ex1I
val ex1E = thm ex1E
val ex1-implies-ex = thm ex1-implies-ex
val the-equality = thm the-equality
val theI = thm theI
val theI' = thm theI'
val theI2 = thm theI2
val the1-equality = thm the1-equality
val the-sym-eq-trivial = thm the-sym-eq-trivial
val disjCI = thm disjCI
val excluded-middle = thm excluded-middle
val case-split-thm = thm case-split-thm
val impCE = thm impCE
val impCE = thm impCE
val iffCE = thm iffCE
val exCI = thm exCI

```
(* 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 = List.filter (fn t => not (wrong-prem (HOLogic.dest-Trueprop
(hd (Thm.prems-of t)))))
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


fun strip-tac i = REPEAT(resolve-tac [impI,allI] i)

(*Obsolete form of disjunctive case analysis*)
fun excluded-middle-tac sP =
    res-inst-tac [(Q,sP)] (excluded-middle RS disjE)

fun case-tac a = res-inst-tac [(P,a)] case-split-thm
\⟩

theorems case-split = case-split-thm [case-names True False]
```

1.17.1 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 ==> ALL x. P x ==> 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: ~ P
```

and *2*: ~ *P* ==> *P*
shows *R*
proof −
from *2* **and** *1* **have** *P* **.**
with *1* **show** *R* **by** (*rule notE*)
qed

lemmas [*Pure.elim*!] = *disjE iffE FalseE conjE exE*
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*

1.17.2 Atomizing meta-level connectives

lemma *atomize-all* [*atomize*]: (!!*x*. *P x*) == *Trueprop* (*ALL x*. *P x*)
proof
assume !!*x*. *P x*
show *ALL x*. *P x* **by** (*rule allI*)
next
assume *ALL x*. *P x*
thus !!*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*
thus *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*
thus *False* **by** (*rule notE*)
qed

lemma *atomize-eq* [*atomize*]: (*x* == *y*) == *Trueprop* (*x* = *y*)
proof
assume *x* == *y*
show *x* = *y* **by** (*unfold prems*) (*rule refl*)

next
  assume $x = y$
  thus $x == y$ **by** (*rule eq-reflection*)
qed

lemma *atomize-conj* [*atomize*]:
  $(!!C.\ (A ==> B ==> PROP\ C) ==> PROP\ C) == Trueprop\ (A \&\ B)$
proof
  assume $!!C.\ (A ==> B ==> PROP\ C) ==> PROP\ C$
  show $A \&\ B$ **by** (*rule conjI*)
next
  fix C
  assume $A \&\ B$
  assume $A ==> B ==> PROP\ C$
  thus $PROP\ C$
  proof *this*
    show A **by** (*rule conjunct1*)
    show B **by** (*rule conjunct2*)
  qed
qed

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

1.17.3 Classical Reasoner setup

use *cladata.ML*
setup *hypsubst-setup*

setup ⟪
  [*ContextRules.addSWrapper* (*fn tac => hyp-subst-tac' ORELSE' tac*)]
⟫

setup *Classical.setup*
setup *clasetup*

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

use *blastdata.ML*
setup *Blast.setup*

1.17.4 Simplifier setup

lemma *meta-eq-to-obj-eq*: $x == y ==> x = y$
proof −
  assume *r*: $x == y$
  show $x = y$ **by** (*unfold r*) (*rule refl*)
qed

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
$(P \sim= Q) = (P = (\sim Q))$
$(P \mid \sim P) = True \quad (\sim P \mid P) = True$
$(x = x) = True$
$(\sim True) = False \ \ (\sim False) = True$
$(\sim P) \sim= P \ \ P \sim= (\sim P)$
$(True=P) = P \ \ (P=True) = P \ \ (False=P) = (\sim P) \ \ (P=False) = (\sim P)$
$(True \dashrightarrow P) = P \ \ (False \dashrightarrow P) = True$
$(P \dashrightarrow True) = True \ \ (P \dashrightarrow P) = True$
$(P \dashrightarrow False) = (\sim P) \ \ (P \dashrightarrow \sim P) = (\sim P)$
$(P \ \& \ True) = P \ \ (True \ \& \ P) = P$
$(P \ \& \ False) = False \ \ (False \ \& \ P) = False$
$(P \ \& \ P) = P \ \ (P \ \& \ (P \ \& \ Q)) = (P \ \& \ Q)$
$(P \ \& \sim P) = False \quad (\sim P \ \& \ P) = False$
$(P \mid True) = True \ \ (True \mid P) = True$
$(P \mid False) = P \ \ (False \mid P) = P$
$(P \mid P) = P \ \ (P \mid (P \mid Q)) = (P \mid Q)$ **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**
$!!P.\ (EX\ x.\ x=t \ \& \ P(x)) = P(t)$
$!!P.\ (EX\ x.\ t=x \ \& \ P(x)) = P(t)$
$!!P.\ (ALL\ x.\ x=t \dashrightarrow P(x)) = P(t)$
$!!P.\ (ALL\ x.\ t=x \dashrightarrow P(x)) = P(t)$
by (*blast*, *blast*, *blast*, *blast*, *blast*, *iprover*+)

lemma *imp-cong*: $(P = P') \Longrightarrow (P' \Longrightarrow (Q = Q')) \Longrightarrow ((P \dashrightarrow Q) = (P' \dashrightarrow Q'))$
by *iprover*

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

lemma *all-simps*:
$!!P\ Q.\ (ALL\ x.\ P\ x \ \& \ Q) \ = ((ALL\ x.\ P\ x) \ \& \ Q)$
$!!P\ Q.\ (ALL\ x.\ P \ \& \ Q\ x) \ = (P \ \& \ (ALL\ x.\ Q\ x))$
$!!P\ Q.\ (ALL\ x.\ P\ x \mid Q) \ = ((ALL\ x.\ P\ x) \mid Q)$
$!!P\ Q.\ (ALL\ x.\ P \mid Q\ x) \ = (P \mid (ALL\ x.\ Q\ x))$

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

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*

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*

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

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*: $(\sim(P \dashrightarrow Q)) = (P \;\&\; \sim Q)$ **by** *blast*
lemma *not-iff*: $(P \sim= Q) = (P = (\sim Q))$ **by** *blast*
lemma *disj-not1*: $(\sim P \mid Q) = (P \dashrightarrow Q)$ **by** *blast*
lemma *disj-not2*: $(P \mid \sim Q) = (Q \dashrightarrow P)$ — changes orientation :-(
by *blast*
lemma *imp-conv-disj*: $(P \dashrightarrow Q) = ((\sim P) \mid Q)$ **by** *blast*

lemma *iff-conv-conj-imp*: $(P = Q) = ((P \dashrightarrow Q) \;\&\; (Q \dashrightarrow P))$ **by** *iprover*

lemma *cases-simp*: $((P \dashrightarrow Q) \;\&\; (\sim P \dashrightarrow Q)) = Q$
— 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*: $((!\ x.\ P\ x) \dashrightarrow Q) = (?\ x.\ P\ x \dashrightarrow Q)$ **by** *blast*
lemma *not-ex*: $(\sim\ (?\ x.\ P(x))) = (!\ x.\sim P(x))$ **by** *iprover*
lemma *imp-ex*: $((?\ x.\ P\ x) \dashrightarrow Q) = (!\ x.\ P\ x \dashrightarrow Q)$ **by** *iprover*

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*:
$(P = P') \Longrightarrow (P' \Longrightarrow (Q = Q')) \Longrightarrow ((P \;\&\; Q) = (P' \;\&\; Q'))$
by *iprover*

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

The | congruence rule: not included by default!

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

lemma *eq-sym-conv*: $(x = y) = (y = x)$
by *iprover*

if-then-else rules

lemma *if-True*: $(\mathit{if}\ \mathit{True}\ \mathit{then}\ x\ \mathit{else}\ y) = x$
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* = *split-if split-if-asm*

lemma *if-def2*: (*if Q then x else y*) = ((*Q* --> *x*) & (~ *Q* --> *y*))
by (*rule split-if*)

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

The following copy of the implication operator is useful for fine-tuning con-

gruence rules. It instructs the simplifier to simplify its premise.

constdefs
simp-implies :: [*prop*, *prop*] => *prop* (**infixr** =*simp*=> *1*)
simp-implies ≡ *op* ==>

lemma *simp-impliesI*:
assumes *PQ*: (*PROP P* ⟹ *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* ⟹ *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* ⟹ *PROP Q*
and *P'*: *PROP P'*
from *PP'* [*symmetric*] **and** *P'* **have** *PROP P*
by (*rule equal-elim-rule1*)
hence *PROP Q* **by** (*rule PQ*)
with *P'QQ'* [*OF P'*] **show** *PROP Q'* **by** (*rule equal-elim-rule1*)
next
assume *P'Q'*: *PROP P'* ⟹ *PROP Q'*
and *P*: *PROP P*
from *PP'* **and** *P* **have** *P'*: *PROP P'* **by** (*rule equal-elim-rule1*)
hence *PROP Q'* **by** (*rule P'Q'*)
with *P'QQ'* [*OF P'*, *symmetric*] **show** *PROP Q*
by (*rule equal-elim-rule1*)
qed

Actual Installation of the Simplifier.

use *simpdata.ML*
setup *Simplifier.method-setup Splitter.split-modifiers* **setup** *simpsetup*
setup *Splitter.setup* **setup** *Clasimp.setup*

Lucas Dixon's eqstep tactic.

use ~~/src/Provers/eqsubst.ML
use *eqrule-HOL-data.ML*
setup *EQSubstTac.setup*

1.17.5 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 = one-of [*false*, *true*];
⟫

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

ML ⟪
local

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

in

val eq-codegen-setup = [*Codegen.add-codegen eq-codegen eq-codegen*];

end;
⟩⟩

setup *eq-codegen-setup*

1.18 Other simple lemmas

declare *disj-absorb* [*simp*] *conj-absorb* [*simp*]

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

theorem *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 (*rule-tac P* = *xa* = *x* **in** *case-split-thm*)
apply (*drule-tac* [*3*] *x* = *x* **in** *fun-cong*, *simp-all*)
done

Needs only HOL-lemmas:

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

1.19 Generic cases and induction

constdefs
induct-forall :: ('*a* => *bool*) => *bool*
induct-forall P == ∀ *x*. *P x*
induct-implies :: *bool* => *bool* => *bool*
induct-implies A B == *A* −−> *B*
induct-equal :: '*a* => '*a* => *bool*
induct-equal x y == *x* = *y*
induct-conj :: *bool* => *bool* => *bool*
induct-conj A B == *A* & *B*

lemma *induct-forall-eq*: (!!*x*. *P x*) == *Trueprop* (*induct-forall* (λ*x*. *P x*))

by (*simp only*: *atomize-all induct-forall-def*)

lemma *induct-implies-eq*: (*A* ==> *B*) == *Trueprop* (*induct-implies A B*)
by (*simp only*: *atomize-imp induct-implies-def*)

lemma *induct-equal-eq*: (*x* == *y*) == *Trueprop* (*induct-equal x y*)
by (*simp only*: *atomize-eq induct-equal-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*)
next
assume *r*: *A* ==> *B* ==> *PROP C* **and** *induct-conj A B*
show *PROP C* **by** (*rule r*) (*simp*! *add*: *induct-conj-def*)+
qed

lemma *induct-impliesI*: (*A* ==> *B*) ==> *induct-implies A B*
by (*simp add*: *induct-implies-def*)

lemmas *induct-atomize* = *atomize-conj induct-forall-eq induct-implies-eq induct-equal-eq*
lemmas *induct-rulify1* [*symmetric*, *standard*] = *induct-forall-eq induct-implies-eq induct-equal-eq*
lemmas *induct-rulify2* = *induct-forall-def induct-implies-def induct-equal-def induct-conj-def*
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 InductMethod = *InductMethodFun*
(*struct*
val dest-concls = *HOLogic.dest-concls*
val cases-default = *thm case-split*
val local-impI = *thm induct-impliesI*
val conjI = *thm conjI*
val atomize = *thms induct-atomize*
val rulify1 = *thms induct-rulify1*
val rulify2 = *thms induct-rulify2*
val localize = [*Thm.symmetric* (*thm induct-implies-def*)]

end);
⟫

setup *InductMethod.setup*

1.19.1 Tags, for the ATP Linkup

constdefs
tag :: *bool* => *bool*
tag P == *P*

These label the distinguished literals of introduction and elimination rules.

lemma *tagI*: *P* ==> *tag P*
by (*simp add*: *tag-def*)

lemma *tagD*: *tag P* ==> *P*
by (*simp add*: *tag-def*)

Applications of "tag" to True and False must go!

lemma *tag-True*: *tag True* = *True*
by (*simp add*: *tag-def*)

lemma *tag-False*: *tag False* = *False*
by (*simp add*: *tag-def*)

end

2 Lattice-Locales: Lattices via Locales

theory *Lattice-Locales*
imports *HOL*
begin

2.1 Lattices

This theory of lattice locales only defines binary sup and inf operations. The extension to finite sets is done in theory *Finite-Set*. In the longer term it may be better to define arbitrary sups and infs via *THE*.

locale *partial-order* =
fixes *below* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** $\sqsubseteq$ *50*)
assumes *refl*[*iff*]: $x \sqsubseteq x$
and *trans*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq z \Longrightarrow x \sqsubseteq z$
and *antisym*: $x \sqsubseteq y \Longrightarrow y \sqsubseteq x \Longrightarrow x = y$

locale *lower-semilattice* = *partial-order* +
fixes *inf* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\sqcap$ *70*)

assumes *inf-le1*: $x \sqcap y \sqsubseteq x$ **and** *inf-le2*: $x \sqcap y \sqsubseteq y$
and *inf-least*: $x \sqsubseteq y \Longrightarrow x \sqsubseteq z \Longrightarrow x \sqsubseteq y \sqcap z$

locale *upper-semilattice* = *partial-order* +
fixes *sup* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\sqcup$ *65*)
assumes *sup-ge1*: $x \sqsubseteq x \sqcup y$ **and** *sup-ge2*: $y \sqsubseteq x \sqcup y$
and *sup-greatest*: $y \sqsubseteq x \Longrightarrow z \sqsubseteq x \Longrightarrow y \sqcup z \sqsubseteq x$

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

lemma (**in** *lower-semilattice*) *inf-commute*: $(x \sqcap y) = (y \sqcap x)$
by(*blast intro*: *antisym inf-le1 inf-le2 inf-least*)

lemma (**in** *upper-semilattice*) *sup-commute*: $(x \sqcup y) = (y \sqcup x)$
by(*blast intro*: *antisym sup-ge1 sup-ge2 sup-greatest*)

lemma (**in** *lower-semilattice*) *inf-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
by(*blast intro*: *antisym inf-le1 inf-le2 inf-least trans del*:*refl*)

lemma (**in** *upper-semilattice*) *sup-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
by(*blast intro*!: *antisym sup-ge1 sup-ge2 intro*: *sup-greatest trans del*:*refl*)

lemma (**in** *lower-semilattice*) *inf-idem*[*simp*]: $x \sqcap x = x$
by(*blast intro*: *antisym inf-le1 inf-le2 inf-least refl*)

lemma (**in** *upper-semilattice*) *sup-idem*[*simp*]: $x \sqcup x = x$
by(*blast intro*: *antisym sup-ge1 sup-ge2 sup-greatest refl*)

lemma (**in** *lower-semilattice*) *inf-left-idem*[*simp*]: $x \sqcap (x \sqcap y) = x \sqcap y$
by (*simp add*: *inf-assoc*[*symmetric*])

lemma (**in** *upper-semilattice*) *sup-left-idem*[*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
by (*simp add*: *sup-assoc*[*symmetric*])

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

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

lemma (**in** *lower-semilattice*) *inf-absorb*: $x \sqsubseteq y \Longrightarrow x \sqcap y = x$
by(*blast intro*: *antisym inf-le1 inf-least refl*)

lemma (**in** *upper-semilattice*) *sup-absorb*: $x \sqsubseteq y \Longrightarrow x \sqcup y = y$
by(*blast intro*: *antisym sup-ge2 sup-greatest refl*)

lemma (**in** *lower-semilattice*) *below-inf-conv*[*simp*]:
$x \sqsubseteq y \sqcap z = (x \sqsubseteq y \wedge x \sqsubseteq z)$

by(*blast intro*: *antisym inf-le1 inf-le2 inf-least refl trans*)

lemma (**in** *upper-semilattice*) *above-sup-conv*[*simp*]:
$x \sqcup y \sqsubseteq z = (x \sqsubseteq z \wedge y \sqsubseteq z)$
by(*blast intro*: *antisym sup-ge1 sup-ge2 sup-greatest refl trans*)

Towards distributivity: if you have one of them, you have them all.

lemma (**in** *lattice*) *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 (**in** *lattice*) *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

A package of rewrite rules for deciding equivalence wrt ACI:

lemma (**in** *lower-semilattice*) *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$
proof −
  have $x \sqcap (y \sqcap z) = (y \sqcap z) \sqcap x$ **by** (*simp only*: *inf-commute*)
  also have ... $= y \sqcap (z \sqcap x)$ **by** (*simp only*: *inf-assoc*)
  also have $z \sqcap x = x \sqcap z$ **by** (*simp only*: *inf-commute*)
  finally(*back-subst*) **show** *?thesis* **.**
qed

lemma (**in** *upper-semilattice*) *sup-left-commute*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
proof −
  have $x \sqcup (y \sqcup z) = (y \sqcup z) \sqcup x$ **by** (*simp only*: *sup-commute*)
  also have ... $= y \sqcup (z \sqcup x)$ **by** (*simp only*: *sup-assoc*)
  also have $z \sqcup x = x \sqcup z$ **by** (*simp only*: *sup-commute*)
  finally(*back-subst*) **show** *?thesis* **.**
qed

lemma (**in** *lower-semilattice*) *inf-left-idem*: $x \sqcap (x \sqcap y) = x \sqcap y$

proof −
have $x \sqcap (x \sqcap y) = (x \sqcap x) \sqcap y$ **by**(*simp only*:*inf-assoc*)
also have $\ldots = x \sqcap y$ **by**(*simp*)
finally show *?thesis* **.**
qed

lemma (**in** *upper-semilattice*) *sup-left-idem*: $x \sqcup (x \sqcup y) = x \sqcup y$
proof −
have $x \sqcup (x \sqcup y) = (x \sqcup x) \sqcup y$ **by**(*simp only*:*sup-assoc*)
also have $\ldots = x \sqcup y$ **by**(*simp*)
finally show *?thesis* **.**
qed

lemmas (**in** *lower-semilattice*) *inf-ACI* =
inf-commute inf-assoc inf-left-commute inf-left-idem

lemmas (**in** *upper-semilattice*) *sup-ACI* =
sup-commute sup-assoc sup-left-commute sup-left-idem

lemmas (**in** *lattice*) *ACI* = *inf-ACI sup-ACI*

2.2 Distributive lattices

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

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

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

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

lemmas (**in** *distrib-lattice*) *distrib* =
sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2

end

3 Orderings: Type classes for $\leq$

theory *Orderings*

imports *Lattice-Locales*
uses (*antisym-setup.ML*)
begin

3.1 Order signatures and orders

axclass
ord < *type*

syntax
op < :: ['*a*::*ord*, '*a*] => *bool* (*op* <)
op <= :: ['*a*::*ord*, '*a*] => *bool* (*op* <=)

global

consts
op < :: ['*a*::*ord*, '*a*] => *bool* ((-/ < -) [*50*, *51*] *50*)
op <= :: ['*a*::*ord*, '*a*] => *bool* ((-/ <= -) [*50*, *51*] *50*)

local

syntax (*xsymbols*)
op <= :: ['*a*::*ord*, '*a*] => *bool* (*op* ≤)
op <= :: ['*a*::*ord*, '*a*] => *bool* ((-/ ≤ -) [*50*, *51*] *50*)

syntax (*HTML* **output**)
op <= :: ['*a*::*ord*, '*a*] => *bool* (*op* ≤)
op <= :: ['*a*::*ord*, '*a*] => *bool* ((-/ ≤ -) [*50*, *51*] *50*)

Syntactic sugar:

syntax
-*gt* :: '*a*::*ord* => '*a* => *bool* (**infixl** > *50*)
-*ge* :: '*a*::*ord* => '*a* => *bool* (**infixl** >= *50*)
translations
x > *y* => *y* < *x*
x >= *y* => *y* <= *x*

syntax (*xsymbols*)
-*ge* :: '*a*::*ord* => '*a* => *bool* (**infixl** ≥ *50*)

syntax (*HTML* **output**)
-*ge* :: ['*a*::*ord*, '*a*] => *bool* (**infixl** ≥ *50*)

3.2 Monotonicity

locale *mono* =
fixes *f*
assumes *mono*: *A* <= *B* ==> *f A* <= *f B*

lemmas *monoI* [*intro?*] = *mono.intro*
and *monoD* [*dest?*] = *mono.mono*

constdefs
min :: ['*a*::*ord*, '*a*] => '*a*
min a b == (*if a* <= *b then a else b*)
max :: ['*a*::*ord*, '*a*] => '*a*
max a b == (*if a* <= *b then b else a*)

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

lemma *min-of-mono*:
ALL x y. (*f x* <= *f y*) = (*x* <= *y*) ==> *min* (*f m*) (*f n*) = *f* (*min m n*)
by (*simp add*: *min-def*)

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

lemma *max-of-mono*:
ALL x y. (*f x* <= *f y*) = (*x* <= *y*) ==> *max* (*f m*) (*f n*) = *f* (*max m n*)
by (*simp add*: *max-def*)

3.3 Orders

axclass *order* < *ord*
order-refl [*iff*]: *x* <= *x*
order-trans: *x* <= *y* ==> *y* <= *z* ==> *x* <= *z*
order-antisym: *x* <= *y* ==> *y* <= *x* ==> *x* = *y*
order-less-le: (*x* < *y*) = (*x* <= *y* & *x* ~= *y*)

Connection to locale:

interpretation *order*:
partial-order[*op* ≤ :: '*a*::*order* ⇒ '*a* ⇒ *bool*]
apply(*rule partial-order.intro*)
apply(*rule order-refl*, *erule* (*1*) *order-trans*, *erule* (*1*) *order-antisym*)
done

Reflexivity.

lemma *order-eq-refl*: !!*x*::'*a*::*order*. *x* = *y* ==> *x* <= *y*
— This form is useful with the classical reasoner.
apply (*erule ssubst*)
apply (*rule order-refl*)
done

lemma *order-less-irrefl* [*iff*]: ~ *x* < (*x*::'*a*::*order*)
by (*simp add*: *order-less-le*)

lemma *order-le-less*: ((*x*::'*a*::*order*) <= *y*) = (*x* < *y* | *x* = *y*)

— NOT suitable for iff, since it can cause PROOF FAILED.
apply (*simp add*: *order-less-le*, *blast*)
done

lemmas *order-le-imp-less-or-eq* = *order-le-less* [*THEN iffD1*, *standard*]

lemma *order-less-imp-le*: !!x::'a::*order*. $x < y$ ==> $x <= y$
by (*simp add*: *order-less-le*)

Asymmetry.

lemma *order-less-not-sym*: (x::'a::*order*) $< y$ ==> ~ ($y < x$)
by (*simp add*: *order-less-le order-antisym*)

lemma *order-less-asym*: $x < (y$::'a::*order*) ==> (~P ==> $y < x$) ==> P
apply (*drule order-less-not-sym*)
apply (*erule contrapos-np*, *simp*)
done

lemma *order-eq-iff*: !!x::'a::*order*. ($x = y$) = ($x \leq y$ & $y \leq x$)
by (*blast intro*: *order-antisym*)

lemma *order-antisym-conv*: (y::'a::*order*) <= x ==> ($x <= y$) = ($x = y$)
by(*blast intro*:*order-antisym*)

Transitivity.

lemma *order-less-trans*: !!x::'a::*order*. [| $x < y$; $y < z$ |] ==> $x < z$
apply (*simp add*: *order-less-le*)
apply (*blast intro*: *order-trans order-antisym*)
done

lemma *order-le-less-trans*: !!x::'a::*order*. [| $x <= y$; $y < z$ |] ==> $x < z$
apply (*simp add*: *order-less-le*)
apply (*blast intro*: *order-trans order-antisym*)
done

lemma *order-less-le-trans*: !!x::'a::*order*. [| $x < y$; $y <= z$ |] ==> $x < z$
apply (*simp add*: *order-less-le*)
apply (*blast intro*: *order-trans order-antisym*)
done

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

lemma *order-less-imp-not-less*: (x::'a::*order*) $< y$ ==> (~ $y < x$) = *True*
by (*blast elim*: *order-less-asym*)

lemma *order-less-imp-triv*: (x::'a::*order*) $< y$ ==> ($y < x$ --> P) = *True*
by (*blast elim*: *order-less-asym*)

lemma *order-less-imp-not-eq*: (x::'a::*order*) $< y$ ==> ($x = y$) = *False*
by *auto*

lemma *order-less-imp-not-eq2*: $(x::'a::order) < y \Longrightarrow (y = x) = False$
by *auto*

Other operators.

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

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

3.4 Transitivity rules for calculational reasoning

lemma *order-neq-le-trans*: $a \sim= b \Longrightarrow (a::'a::order) <= b \Longrightarrow a < b$
by (*simp add*: *order-less-le*)

lemma *order-le-neq-trans*: $(a::'a::order) <= b \Longrightarrow a \sim= b \Longrightarrow a < b$
by (*simp add*: *order-less-le*)

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

3.5 Least value operator

constdefs
Least :: $('a::ord \Rightarrow bool) \Rightarrow 'a$ (**binder** *LEAST* *10*)
$Least\ P == THE\ x.\ P\ x\ \&\ (ALL\ y.\ P\ y \longrightarrow x <= y)$
— We can no longer use LeastM because the latter requires Hilbert-AC.

lemma *LeastI2-order*:
$[|\ P\ (x::'a::order);$
$!!y.\ P\ y \Longrightarrow x <= y;$
$!!x.\ [|\ P\ x;\ ALL\ y.\ P\ y \longrightarrow x \leq y\ |] \Longrightarrow Q\ x\ |]$
$\Longrightarrow Q\ (Least\ P)$
apply (*unfold Least-def*)
apply (*rule theI2*)
apply (*blast intro*: *order-antisym*)+
done

lemma *Least-equality*:
$[|\ P\ (k::'a::order);\ !!x.\ P\ x \Longrightarrow k <= x\ |] \Longrightarrow (LEAST\ x.\ P\ x) = k$
apply (*simp add*: *Least-def*)
apply (*rule the-equality*)
apply (*auto intro*!: *order-antisym*)
done

3.6 Linear / total orders

axclass *linorder* < *order*
linorder-linear: $x <= y \mid y <= x$

lemma *linorder-less-linear*: !!x::'a::*linorder*. $x<y \mid x=y \mid y<x$
apply (*simp add*: *order-less-le*)
apply (*insert linorder-linear*, *blast*)
done

lemma *linorder-le-less-linear*: !!x::'a::*linorder*. $x\leq y \mid y<x$
by (*simp add*: *order-le-less linorder-less-linear*)

lemma *linorder-le-cases* [*case-names le ge*]:
$((x::'a::linorder) \leq y \Longrightarrow P) \Longrightarrow (y \leq x \Longrightarrow P) \Longrightarrow P$
by (*insert linorder-linear*, *blast*)

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

lemma *linorder-not-less*: !!x::'a::*linorder*. $(\sim x < y) = (y <= x)$
apply (*simp add*: *order-less-le*)
apply (*insert linorder-linear*)
apply (*blast intro*: *order-antisym*)
done

lemma *linorder-not-le*: !!x::'a::*linorder*. $(\sim x <= y) = (y < x)$
apply (*simp add*: *order-less-le*)
apply (*insert linorder-linear*)
apply (*blast intro*: *order-antisym*)
done

lemma *linorder-neq-iff*: !!x::'a::*linorder*. $(x \sim= y) = (x<y \mid y<x)$
by (*cut-tac* $x = x$ **and** $y = y$ **in** *linorder-less-linear*, *auto*)

lemma *linorder-neqE*: $x \sim= (y::'a::linorder) \Longrightarrow (x < y \Longrightarrow R) \Longrightarrow (y < x \Longrightarrow R) \Longrightarrow R$
by (*simp add*: *linorder-neq-iff*, *blast*)

lemma *linorder-antisym-conv1*: $\sim (x::'a::linorder) < y \Longrightarrow (x <= y) = (x = y)$
by(*blast intro*:*order-antisym dest*:*linorder-not-less*[*THEN iffD1*])

lemma *linorder-antisym-conv2*: $(x::'a::linorder) <= y \Longrightarrow (\sim x < y) = (x = y)$
by(*blast intro*:*order-antisym dest*:*linorder-not-less*[*THEN iffD1*])

lemma *linorder-antisym-conv3*: $\sim (y::'a::linorder) < x \Longrightarrow (\sim x < y) = (x = y)$
by(*blast intro*:*order-antisym dest*:*linorder-not-less*[*THEN iffD1*])

Replacing the old Nat.leI

lemma *leI*: ~ *x* < *y* ==> *y* <= (*x*::'*a*::*linorder*)
by (*simp only*: *linorder-not-less*)

lemma *leD*: *y* <= (*x*::'*a*::*linorder*) ==> ~ *x* < *y*
by (*simp only*: *linorder-not-less*)

lemma *not-leE*: ~ *y* <= (*x*::'*a*::*linorder*) ==> *x* < *y*
by (*simp only*: *linorder-not-le*)

use *antisym-setup.ML*
setup *antisym-setup*

3.7 Setup of transitivity reasoner as Solver

lemma *less-imp-neq*: [| (*x*::'*a*::*order*) < *y* |] ==> *x* ~= *y*
by (*erule contrapos-pn*, *erule subst*, *rule order-less-irrefl*)

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

ML-setup ⟪

```
(* The setting up of Quasi-Tac serves as a demo.  Since there is no
   class for quasi orders, the tactics Quasi-Tac.trans-tac and
   Quasi-Tac.quasi-tac are not of much use. *)

fun decomp-gen sort sign (Trueprop $ t) =
  let fun of-sort t = let val T = type-of t in
        (* exclude numeric types: linear arithmetic subsumes transitivity *)
        T <> HOLogic.natT andalso T <> HOLogic.intT andalso
        T <> HOLogic.realT andalso Sign.of-sort sign (T, sort) end
  fun dec (Const (Not, -) $ t) = (
        case dec t of
          NONE => NONE
        | SOME (t1, rel, t2) => SOME (t1, ~ ^ rel, t2))
      | dec (Const (op =,  -) $ t1 $ t2) =
          if of-sort t1
          then SOME (t1, =, t2)
          else NONE
      | dec (Const (op <=,  -) $ t1 $ t2) =
          if of-sort t1
          then SOME (t1, <=, t2)
          else NONE
      | dec (Const (op <,  -) $ t1 $ t2) =
          if of-sort t1
          then SOME (t1, <, t2)
          else NONE
```

```
      | dec - = NONE
  in dec t end;

structure Quasi-Tac = Quasi-Tac-Fun (
  struct
    val le-trans = thm order-trans;
    val le-refl = thm order-refl;
    val eqD1 = thm order-eq-refl;
    val eqD2 = thm sym RS thm order-eq-refl;
    val less-reflE = thm order-less-irrefl RS thm notE;
    val less-imp-le = thm order-less-imp-le;
    val le-neq-trans = thm order-le-neq-trans;
    val neq-le-trans = thm order-neq-le-trans;
    val less-imp-neq = thm less-imp-neq;
    val decomp-trans = decomp-gen [Orderings.order];
    val decomp-quasi = decomp-gen [Orderings.order];

  end); (* struct *)

structure Order-Tac = Order-Tac-Fun (
  struct
    val less-reflE = thm order-less-irrefl RS thm notE;
    val le-refl = thm order-refl;
    val less-imp-le = thm order-less-imp-le;
    val not-lessI = thm linorder-not-less RS thm iffD2;
    val not-leI = thm linorder-not-le RS thm iffD2;
    val not-lessD = thm linorder-not-less RS thm iffD1;
    val not-leD = thm linorder-not-le RS thm iffD1;
    val eqI = thm order-antisym;
    val eqD1 = thm order-eq-refl;
    val eqD2 = thm sym RS thm order-eq-refl;
    val less-trans = thm order-less-trans;
    val less-le-trans = thm order-less-le-trans;
    val le-less-trans = thm order-le-less-trans;
    val le-trans = thm order-trans;
    val le-neq-trans = thm order-le-neq-trans;
    val neq-le-trans = thm order-neq-le-trans;
    val less-imp-neq = thm less-imp-neq;
    val eq-neq-eq-imp-neq = thm eq-neq-eq-imp-neq;
    val not-sym = thm not-sym;
    val decomp-part = decomp-gen [Orderings.order];
    val decomp-lin = decomp-gen [Orderings.linorder];

  end); (* struct *)

simpset-ref() := simpset ()
    addSolver (mk-solver Trans-linear (fn - => Order-Tac.linear-tac))
    addSolver (mk-solver Trans-partial (fn - => Order-Tac.partial-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). ∗)
⟩⟩

3.8 Min and max on (linear) orders

Instantiate locales:

interpretation *min-max*:
lower-semilattice[*op* ≤ *min* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a*]
apply(*rule lower-semilattice-axioms.intro*)
apply(*simp add*:*min-def linorder-not-le order-less-imp-le*)
apply(*simp add*:*min-def linorder-not-le order-less-imp-le*)
apply(*simp add*:*min-def linorder-not-le order-less-imp-le*)
done

interpretation *min-max*:
upper-semilattice[*op* ≤ *max* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a*]
apply −
apply(*rule upper-semilattice-axioms.intro*)
apply(*simp add*: *max-def linorder-not-le order-less-imp-le*)
apply(*simp add*: *max-def linorder-not-le order-less-imp-le*)
apply(*simp add*: *max-def linorder-not-le order-less-imp-le*)
done

interpretation *min-max*:
lattice[*op* ≤ *min* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a max*]
.

interpretation *min-max*:
distrib-lattice[*op* ≤ *min* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a max*]
apply(*rule distrib-lattice-axioms.intro*)
apply(*rule-tac x*=*x* **and** *y*=*y* **in** *linorder-le-cases*)
apply(*rule-tac x*=*x* **and** *y*=*z* **in** *linorder-le-cases*)
apply(*rule-tac x*=*y* **and** *y*=*z* **in** *linorder-le-cases*)
apply(*simp add*:*min-def max-def*)
apply(*simp add*:*min-def max-def*)
apply(*rule-tac x*=*y* **and** *y*=*z* **in** *linorder-le-cases*)
apply(*simp add*:*min-def max-def*)
apply(*simp add*:*min-def max-def*)
apply(*rule-tac x*=*x* **and** *y*=*z* **in** *linorder-le-cases*)
apply(*rule-tac x*=*y* **and** *y*=*z* **in** *linorder-le-cases*)
apply(*simp add*:*min-def max-def*)
apply(*simp add*:*min-def max-def*)
apply(*rule-tac x*=*y* **and** *y*=*z* **in** *linorder-le-cases*)
apply(*simp add*:*min-def max-def*)
apply(*simp add*:*min-def max-def*)
done

lemma *le-max-iff-disj*: !!*z*::'*a*::*linorder*. (*z* <= *max x y*) = (*z* <= *x* | *z* <= *y*)
apply(*simp add*:*max-def*)
apply (*insert linorder-linear*)
apply (*blast intro*: *order-trans*)
done

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

lemma *less-max-iff-disj*: !!*z*::'*a*::*linorder*. (*z* < *max x y*) = (*z* < *x* | *z* < *y*)
apply (*simp add*: *max-def order-le-less*)
apply (*insert linorder-less-linear*)
apply (*blast intro*: *order-less-trans*)
done

lemma *max-less-iff-conj* [*simp*]:
!!*z*::'*a*::*linorder*. (*max x y* < *z*) = (*x* < *z* & *y* < *z*)
apply (*simp add*: *order-le-less max-def*)
apply (*insert linorder-less-linear*)
apply (*blast intro*: *order-less-trans*)
done

lemma *min-less-iff-conj* [*simp*]:
!!*z*::'*a*::*linorder*. (*z* < *min x y*) = (*z* < *x* & *z* < *y*)
apply (*simp add*: *order-le-less min-def*)
apply (*insert linorder-less-linear*)
apply (*blast intro*: *order-less-trans*)
done

lemma *min-le-iff-disj*: !!*z*::'*a*::*linorder*. (*min x y* <= *z*) = (*x* <= *z* | *y* <= *z*)
apply (*simp add*: *min-def*)
apply (*insert linorder-linear*)
apply (*blast intro*: *order-trans*)
done

lemma *min-less-iff-disj*: !!*z*::'*a*::*linorder*. (*min x y* < *z*) = (*x* < *z* | *y* < *z*)
apply (*simp add*: *min-def order-le-less*)
apply (*insert linorder-less-linear*)
apply (*blast intro*: *order-less-trans*)
done

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*]

lemma *split-min*:
P (*min* (*i*::'*a*::*linorder*) *j*) = ((*i* <= *j* --> *P*(*i*)) & (~ *i* <= *j* --> *P*(*j*)))

by (*simp add*: *min-def*)

lemma *split-max*:
P (*max* (*i*::'*a*::*linorder*) *j*) = ((*i* <= *j* −−> *P*(*j*)) & (~ *i* <= *j* −−> *P*(*i*)))
by (*simp add*: *max-def*)

3.9 Bounded quantifiers

syntax
-lessAll :: [*idt*, '*a*, *bool*] => *bool* ((*3ALL -<-./ -*) [*0*, *0*, *10*] *10*)
-lessEx :: [*idt*, '*a*, *bool*] => *bool* ((*3EX -<-./ -*) [*0*, *0*, *10*] *10*)
-leAll :: [*idt*, '*a*, *bool*] => *bool* ((*3ALL -<=-./ -*) [*0*, *0*, *10*] *10*)
-leEx :: [*idt*, '*a*, *bool*] => *bool* ((*3EX -<=-./ -*) [*0*, *0*, *10*] *10*)

-gtAll :: [*idt*, '*a*, *bool*] => *bool* ((*3ALL ->-./ -*) [*0*, *0*, *10*] *10*)
-gtEx :: [*idt*, '*a*, *bool*] => *bool* ((*3EX ->-./ -*) [*0*, *0*, *10*] *10*)
-geAll :: [*idt*, '*a*, *bool*] => *bool* ((*3ALL ->=-./ -*) [*0*, *0*, *10*] *10*)
-geEx :: [*idt*, '*a*, *bool*] => *bool* ((*3EX ->=-./ -*) [*0*, *0*, *10*] *10*)

syntax (*xsymbols*)
-lessAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ *-<-./ -*) [*0*, *0*, *10*] *10*)
-lessEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ *-<-./ -*) [*0*, *0*, *10*] *10*)
-leAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ *-≤-./ -*) [*0*, *0*, *10*] *10*)
-leEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ *-≤-./ -*) [*0*, *0*, *10*] *10*)

-gtAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ *->-./ -*) [*0*, *0*, *10*] *10*)
-gtEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ *->-./ -*) [*0*, *0*, *10*] *10*)
-geAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ *-≥-./ -*) [*0*, *0*, *10*] *10*)
-geEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ *-≥-./ -*) [*0*, *0*, *10*] *10*)

syntax (*HOL*)
-lessAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*! *-<-./ -*) [*0*, *0*, *10*] *10*)
-lessEx :: [*idt*, '*a*, *bool*] => *bool* ((*3?* *-<-./ -*) [*0*, *0*, *10*] *10*)
-leAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*! *-<=-./ -*) [*0*, *0*, *10*] *10*)
-leEx :: [*idt*, '*a*, *bool*] => *bool* ((*3?* *-<=-./ -*) [*0*, *0*, *10*] *10*)

syntax (*HTML* **output**)
-lessAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ *-<-./ -*) [*0*, *0*, *10*] *10*)
-lessEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ *-<-./ -*) [*0*, *0*, *10*] *10*)
-leAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ *-≤-./ -*) [*0*, *0*, *10*] *10*)
-leEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ *-≤-./ -*) [*0*, *0*, *10*] *10*)

-gtAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ *->-./ -*) [*0*, *0*, *10*] *10*)
-gtEx :: [*idt*, '*a*, *bool*] => *bool* ((*3*∃ *->-./ -*) [*0*, *0*, *10*] *10*)
-geAll :: [*idt*, '*a*, *bool*] => *bool* ((*3*∀ *-≥-./ -*) [*0*, *0*, *10*] *10*)
-geEx :: [*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
  fun mk v v' q n P =
    if v=v' andalso not (v mem (map fst (Term.add-frees n [])))
    then Syntax.const q $ Syntax.mark-bound v' $ n $ P else raise Match;
  fun all-tr' [Const (-bound,-) $ Free (v,-),
            Const(op -->,-) $ (Const (op <,-) $ (Const (-bound,-) $ Free (v',-))
$ n ) $ P] =
    mk v v' -lessAll n P

  | all-tr' [Const (-bound,-) $ Free (v,-),
               Const(op -->,-) $ (Const (op <=,-) $ (Const (-bound,-) $ Free
(v',-)) $ n ) $ P] =
    mk v v' -leAll n P

  | all-tr' [Const (-bound,-) $ Free (v,-),
              Const(op -->,-) $ (Const (op <,-) $ n $ (Const (-bound,-) $ Free
(v',-))) $ P] =
    mk v v' -gtAll n P

  | all-tr' [Const (-bound,-) $ Free (v,-),
             Const(op -->,-) $ (Const (op <=,-) $ n $ (Const (-bound,-) $ Free
(v',-))) $ P] =
    mk v v' -geAll n P;

  fun ex-tr' [Const (-bound,-) $ Free (v,-),
              Const(op &,-) $ (Const (op <,-) $ (Const (-bound,-) $ Free (v',-)) $
n ) $ P] =
    mk v v' -lessEx n P

  | ex-tr' [Const (-bound,-) $ Free (v,-),
              Const(op &,-) $ (Const (op <=,-) $ (Const (-bound,-) $ Free (v',-))
$ n ) $ P] =
    mk v v' -leEx n P

  | ex-tr' [Const (-bound,-) $ Free (v,-),
            Const(op &,-) $ (Const (op <,-) $ n $ (Const (-bound,-) $ Free (v',-)))
$ P] =
    mk v v' -gtEx n P

  | ex-tr' [Const (-bound,-) $ Free (v,-),
```

Const(*op &*,-) $ (*Const* (*op <=*,-) $ *n* $ (*Const* (*-bound*,-) $ *Free* (*v'*,-))) $ *P*] =
  mk v v' -geEx n P
in
[(*ALL* , *all-tr'*), (*EX* , *ex-tr'*)]
end
⟩⟩

3.10 Extra transitivity rules

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

lemma *xt1*: $a = b \Longrightarrow b > c \Longrightarrow a > c$
by *simp*

lemma *xt2*: $a > b \Longrightarrow b = c \Longrightarrow a > c$
by *simp*

lemma *xt3*: $a = b \Longrightarrow b >= c \Longrightarrow a >= c$
by *simp*

lemma *xt4*: $a >= b \Longrightarrow b = c \Longrightarrow a >= c$
by *simp*

lemma *xt5*: $(x::'a::order) >= y \Longrightarrow y >= x \Longrightarrow x = y$
by *simp*

lemma *xt6*: $(x::'a::order) >= y \Longrightarrow y >= z \Longrightarrow x >= z$
by *simp*

lemma *xt7*: $(x::'a::order) > y \Longrightarrow y >= z \Longrightarrow x > z$
by *simp*

lemma *xt8*: $(x::'a::order) >= y \Longrightarrow y > z \Longrightarrow x > z$
by *simp*

lemma *xt9*: $(a::'a::order) > b \Longrightarrow b > a \Longrightarrow ?P$
by *simp*

lemma *xt10*: $(x::'a::order) > y \Longrightarrow y > z \Longrightarrow x > z$
by *simp*

lemma *xt11*: $(a::'a::order) >= b \Longrightarrow a \sim= b \Longrightarrow a > b$
by *simp*

lemma *xt12*: $(a::'a::order) \sim= b \Longrightarrow a >= b \Longrightarrow a > b$
by *simp*

lemma *xt13*: $a = f\ b \Longrightarrow b > c \Longrightarrow (!!x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow$

$a > f\ c$
by *simp*

lemma *xt14*: $a > b \Longrightarrow f\ b = c \Longrightarrow (!!x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow$
$f\ a > c$
by *auto*

lemma *xt15*: $a = f\ b \Longrightarrow b >= c \Longrightarrow (!!x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow$
$a >= f\ c$
by *simp*

lemma *xt16*: $a >= b \Longrightarrow f\ b = c \Longrightarrow (!!\ x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow$
$f\ a >= c$
by *auto*

lemma *xt17*: $(a::'a::order) >= f\ b \Longrightarrow b >= c \Longrightarrow$
$(!!x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow a >= f\ c$
by (*subgoal-tac* $f\ b >= f\ c$, *force*, *force*)

lemma *xt18*: $(a::'a::order) >= b \Longrightarrow (f\ b::'b::order) >= c \Longrightarrow$
$(!!x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow f\ a >= c$
by (*subgoal-tac* $f\ a >= f\ b$, *force*, *force*)

lemma *xt19*: $(a::'a::order) > f\ b \Longrightarrow (b::'b::order) >= c \Longrightarrow$
$(!!x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow a > f\ c$
by (*subgoal-tac* $f\ b >= f\ c$, *force*, *force*)

lemma *xt20*: $(a::'a::order) > b \Longrightarrow (f\ b::'b::order) >= c \Longrightarrow$
$(!!x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow f\ a > c$
by (*subgoal-tac* $f\ a > f\ b$, *force*, *force*)

lemma *xt21*: $(a::'a::order) >= f\ b \Longrightarrow b > c \Longrightarrow$
$(!!x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow a > f\ c$
by (*subgoal-tac* $f\ b > f\ c$, *force*, *force*)

lemma *xt22*: $(a::'a::order) >= b \Longrightarrow (f\ b::'b::order) > c \Longrightarrow$
$(!!x\ y.\ x >= y \Longrightarrow f\ x >= f\ y) \Longrightarrow f\ a > c$
by (*subgoal-tac* $f\ a >= f\ b$, *force*, *force*)

lemma *xt23*: $(a::'a::order) > f\ b \Longrightarrow (b::'b::order) > c \Longrightarrow$
$(!!x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow a > f\ c$
by (*subgoal-tac* $f\ b > f\ c$, *force*, *force*)

lemma *xt24*: $(a::'a::order) > b \Longrightarrow (f\ b::'b::order) > c \Longrightarrow$
$(!!x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow f\ a > c$
by (*subgoal-tac* $f\ a > f\ b$, *force*, *force*)

lemmas *xtrans = xt1 xt2 xt3 xt4 xt5 xt6 xt7 xt8 xt9 xt10 xt11 xt12*

xt13 xt14 xt15 xt15 xt17 xt18 xt19 xt20 xt21 xt22 xt23 xt24

end

4 LOrder: Lattice Orders

theory *LOrder*
imports *Orderings*
begin

The theory of lattices developed here is taken from the book:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979.

constdefs
is-meet :: $(('a::order) \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$
is-meet m == ! *a b x*. *m a b* $\leq$ *a* $\wedge$ *m a b* $\leq$ *b* $\wedge$ ($x \leq a \wedge x \leq b \longrightarrow x \leq m\ a\ b$)
is-join :: $(('a::order) \Rightarrow 'a \Rightarrow 'a) \Rightarrow bool$
is-join j == ! *a b x*. *a* $\leq$ *j a b* $\wedge$ *b* $\leq$ *j a b* $\wedge$ ($a \leq x \wedge b \leq x \longrightarrow j\ a\ b \leq x$)

lemma *is-meet-unique*:
assumes *is-meet u is-meet v* **shows** *u* = *v*
proof −
{
fix *a b* :: $'a \Rightarrow 'a \Rightarrow 'a$
assume *a*: *is-meet a*
assume *b*: *is-meet b*
{
fix *x y*
let *?za* = *a x y*
let *?zb* = *b x y*
from *a* **have** *za-le*: *?za* <= *x* & *?za* <= *y* **by** (*auto simp add*: *is-meet-def*)
with *b* **have** *?za* <= *?zb* **by** (*auto simp add*: *is-meet-def*)
}
}
note *f-le* = *this*
show *u* = *v* **by** ((*rule ext*)+, *simp-all add*: *order-antisym prems f-le*)
qed

lemma *is-join-unique*:
assumes *is-join u is-join v* **shows** *u* = *v*
proof −
{
fix *a b* :: $'a \Rightarrow 'a \Rightarrow 'a$

```
    assume a: is-join a
    assume b: is-join b
    {
      fix x y
      let ?za = a x y
      let ?zb = b x y
      from a have za-le: x <= ?za & y <= ?za by (auto simp add: is-join-def)
      with b have ?zb <= ?za by (auto simp add: is-join-def)
    }
  }
  note f-le = this
  show u = v by ((rule ext)+, simp-all add: order-antisym prems f-le)
qed

axclass join-semilorder < order
  join-exists: ? j. is-join j

axclass meet-semilorder < order
  meet-exists: ? m. is-meet m

axclass lorder < join-semilorder, meet-semilorder

constdefs
  meet :: ('a::meet-semilorder) ⇒ 'a ⇒ 'a
  meet == THE m. is-meet m
  join :: ('a::join-semilorder) ⇒ 'a ⇒ 'a
  join ==  THE j. is-join j

lemma is-meet-meet: is-meet (meet::'a ⇒ 'a ⇒ ('a::meet-semilorder))
proof −
  from meet-exists obtain k::'a ⇒ 'a ⇒ 'a where is-meet k ..
  with is-meet-unique[of - k] show ?thesis
    by (simp add: meet-def theI[of is-meet])
qed

lemma meet-unique: (is-meet m) = (m = meet)
by (insert is-meet-meet, auto simp add: is-meet-unique)

lemma is-join-join: is-join (join::'a ⇒ 'a ⇒ ('a::join-semilorder))
proof −
  from join-exists obtain k::'a ⇒ 'a ⇒ 'a where is-join k ..
  with is-join-unique[of - k] show ?thesis
    by (simp add: join-def theI[of is-join])
qed

lemma join-unique: (is-join j) = (j = join)
by (insert is-join-join, auto simp add: is-join-unique)

lemma meet-left-le: meet a b ≤ (a::'a::meet-semilorder)
```

by (*insert is-meet-meet*, *auto simp add*: *is-meet-def*)

lemma *meet-right-le*: *meet a b* ≤ (*b*::'*a*::*meet-semilorder*)
by (*insert is-meet-meet*, *auto simp add*: *is-meet-def*)

lemma *meet-imp-le*: *x* ≤ *a* ⟹ *x* ≤ *b* ⟹ *x* ≤ *meet a* (*b*::'*a*::*meet-semilorder*)
by (*insert is-meet-meet*, *auto simp add*: *is-meet-def*)

lemma *join-left-le*: *a* ≤ *join a* (*b*::'*a*::*join-semilorder*)
by (*insert is-join-join*, *auto simp add*: *is-join-def*)

lemma *join-right-le*: *b* ≤ *join a* (*b*::'*a*::*join-semilorder*)
by (*insert is-join-join*, *auto simp add*: *is-join-def*)

lemma *join-imp-le*: *a* ≤ *x* ⟹ *b* ≤ *x* ⟹ *join a b* ≤ (*x*::'*a*::*join-semilorder*)
by (*insert is-join-join*, *auto simp add*: *is-join-def*)

lemmas *meet-join-le* = *meet-left-le meet-right-le join-left-le join-right-le*

lemma *is-meet-min*: *is-meet* (*min*::'*a* ⇒ '*a* ⇒ ('*a*::*linorder*))
by (*auto simp add*: *is-meet-def min-def*)

lemma *is-join-max*: *is-join* (*max*::'*a* ⇒ '*a* ⇒ ('*a*::*linorder*))
by (*auto simp add*: *is-join-def max-def*)

instance *linorder* ⊆ *meet-semilorder*
proof
  from *is-meet-min* **show** *?* (*m*::'*a*⇒'*a*⇒('*a*::*linorder*)). *is-meet m* **by** *auto*
qed

instance *linorder* ⊆ *join-semilorder*
proof
  from *is-join-max* **show** *?* (*j*::'*a*⇒'*a*⇒('*a*::*linorder*)). *is-join j* **by** *auto*
qed

instance *linorder* ⊆ *lorder* **..**

lemma *meet-min*: *meet* = (*min* :: '*a*⇒'*a*⇒('*a*::*linorder*))
by (*simp add*: *is-meet-meet is-meet-min is-meet-unique*)

lemma *join-max*: *join* = (*max* :: '*a*⇒'*a*⇒('*a*::*linorder*))
by (*simp add*: *is-join-join is-join-max is-join-unique*)

lemma *meet-idempotent*[*simp*]: *meet x x* = *x*
by (*rule order-antisym*, *simp-all add*: *meet-left-le meet-imp-le*)

lemma *join-idempotent*[*simp*]: *join x x* = *x*
by (*rule order-antisym*, *simp-all add*: *join-left-le join-imp-le*)

lemma *meet-comm*: *meet x y = meet y x*
by (*rule order-antisym*, (*simp add*: *meet-left-le meet-right-le meet-imp-le*)+)

lemma *join-comm*: *join x y = join y x*
by (*rule order-antisym*, (*simp add*: *join-right-le join-left-le join-imp-le*)+)

lemma *meet-assoc*: *meet* (*meet x y*) *z* = *meet x* (*meet y z*) (**is** *?l=?r*)
proof −
have *?l <= meet x y & meet x y <= x & ?l <= z & meet x y <= y* **by** (*simp add*: *meet-left-le meet-right-le*)
hence *?l <= x & ?l <= y & ?l <= z* **by** *auto*
hence *?l <= ?r* **by** (*simp add*: *meet-imp-le*)
hence *a*:*?l <= meet x* (*meet y z*) **by** (*simp add*: *meet-imp-le*)
have *?r <= meet y z & meet y z <= y & meet y z <= z & ?r <= x* **by** (*simp add*: *meet-left-le meet-right-le*)
hence *?r <= x & ?r <= y & ?r <= z* **by** (*auto*)
hence *?r <= meet x y & ?r <= z* **by** (*simp add*: *meet-imp-le*)
hence *b*:*?r <= ?l* **by** (*simp add*: *meet-imp-le*)
from *a b* **show** *?l = ?r* **by** *auto*
qed

lemma *join-assoc*: *join* (*join x y*) *z* = *join x* (*join y z*) (**is** *?l=?r*)
proof −
have *join x y <= ?l & x <= join x y & z <= ?l & y <= join x y* **by** (*simp add*: *join-left-le join-right-le*)
hence *x <= ?l & y <= ?l & z <= ?l* **by** *auto*
hence *join y z <= ?l & x <= ?l* **by** (*simp add*: *join-imp-le*)
hence *a*:*?r <= ?l* **by** (*simp add*: *join-imp-le*)
have *join y z <= ?r & y <= join y z & z <= join y z & x <= ?r* **by** (*simp add*: *join-left-le join-right-le*)
hence *y <= ?r & z <= ?r & x <= ?r* **by** *auto*
hence *join x y <= ?r & z <= ?r* **by** (*simp add*: *join-imp-le*)
hence *b*:*?l <= ?r* **by** (*simp add*: *join-imp-le*)
from *a b* **show** *?l = ?r* **by** *auto*
qed

lemma *meet-left-comm*: *meet a* (*meet b c*) = *meet b* (*meet a c*)
by (*simp add*: *meet-assoc*[*symmetric*, *of a b c*], *simp add*: *meet-comm*[*of a b*], *simp add*: *meet-assoc*)

lemma *meet-left-idempotent*: *meet y* (*meet y x*) = *meet y x*
by (*simp add*: *meet-assoc meet-comm meet-left-comm*)

lemma *join-left-comm*: *join a* (*join b c*) = *join b* (*join a c*)
by (*simp add*: *join-assoc*[*symmetric*, *of a b c*], *simp add*: *join-comm*[*of a b*], *simp add*: *join-assoc*)

lemma *join-left-idempotent*: *join y* (*join y x*) = *join y x*
by (*simp add*: *join-assoc join-comm join-left-comm*)

lemmas *meet-aci = meet-assoc meet-comm meet-left-comm meet-left-idempotent*

lemmas *join-aci = join-assoc join-comm join-left-comm join-left-idempotent*

lemma *le-def-meet*: $(x <= y) = (meet\ x\ y = x)$
proof −
  have *u*: $x <= y \longrightarrow meet\ x\ y = x$
  proof
    assume $x <= y$
   hence $x <= meet\ x\ y\ \&\ meet\ x\ y <= x$ **by** (*simp add*: *meet-imp-le meet-left-le*)
    thus $meet\ x\ y = x$ **by** *auto*
  qed
  have *v*:$meet\ x\ y = x \longrightarrow x <= y$
  proof
    have *a*:$meet\ x\ y <= y$ **by** (*simp add*: *meet-right-le*)
    assume $meet\ x\ y = x$
    hence $x = meet\ x\ y$ **by** *auto*
    with *a* **show** $x <= y$ **by** (*auto*)
  qed
  from *u v* **show** *?thesis* **by** *blast*
qed

lemma *le-def-join*: $(x <= y) = (join\ x\ y = y)$
proof −
  have *u*: $x <= y \longrightarrow join\ x\ y = y$
  proof
    assume $x <= y$
    hence $join\ x\ y <= y\ \&\ y <= join\ x\ y$ **by** (*simp add*: *join-imp-le join-right-le*)
    thus $join\ x\ y = y$ **by** *auto*
  qed
  have *v*:$join\ x\ y = y \longrightarrow x <= y$
  proof
    have *a*:$x <= join\ x\ y$ **by** (*simp add*: *join-left-le*)
    assume $join\ x\ y = y$
    hence $y = join\ x\ y$ **by** *auto*
    with *a* **show** $x <= y$ **by** (*auto*)
  qed
  from *u v* **show** *?thesis* **by** *blast*
qed

lemma *meet-join-absorp*: $meet\ x\ (join\ x\ y) = x$
proof −
  have *a*:$meet\ x\ (join\ x\ y) <= x$ **by** (*simp add*: *meet-left-le*)
  have *b*:$x <= meet\ x\ (join\ x\ y)$ **by** (*rule meet-imp-le*, *simp-all add*: *join-left-le*)
  from *a b* **show** *?thesis* **by** *auto*
qed

lemma *join-meet-absorp*: $join\ x\ (meet\ x\ y) = x$

proof −
have *a*:*x* <= *join x* (*meet x y*) **by** (*simp add*: *join-left-le*)
have *b*:*join x* (*meet x y*) <= *x* **by** (*rule join-imp-le*, *simp-all add*: *meet-left-le*)
from *a b* **show** *?thesis* **by** *auto*
qed

lemma *meet-mono*: $y \leq z \Longrightarrow$ *meet x y* $\leq$ *meet x z*
proof −
assume *a*: *y* <= *z*
have *meet x y* <= *x* & *meet x y* <= *y* **by** (*simp add*: *meet-left-le meet-right-le*)
with *a* **have** *meet x y* <= *x* & *meet x y* <= *z* **by** *auto*
thus *meet x y* <= *meet x z* **by** (*simp add*: *meet-imp-le*)
qed

lemma *join-mono*: $y \leq z \Longrightarrow$ *join x y* $\leq$ *join x z*
proof −
assume *a*: $y \leq z$
have *x* <= *join x z* & *z* <= *join x z* **by** (*simp add*: *join-left-le join-right-le*)
with *a* **have** *x* <= *join x z* & *y* <= *join x z* **by** *auto*
thus *join x y* <= *join x z* **by** (*simp add*: *join-imp-le*)
qed

lemma *distrib-join-le*: *join x* (*meet y z*) $\leq$ *meet* (*join x y*) (*join x z*) (**is** - <= *?r*)
proof −
have *a*: *x* <= *?r* **by** (*rule meet-imp-le*, *simp-all add*: *join-left-le*)
from *meet-join-le* **have** *b*: *meet y z* <= *?r*
by (*rule-tac meet-imp-le*, (*blast intro*: *order-trans*)+)
from *a b* **show** *?thesis* **by** (*simp add*: *join-imp-le*)
qed

lemma *distrib-meet-le*: *join* (*meet x y*) (*meet x z*) $\leq$ *meet x* (*join y z*) (**is** *?l* <= -)
proof −
have *a*: *?l* <= *x* **by** (*rule join-imp-le*, *simp-all add*: *meet-left-le*)
from *meet-join-le* **have** *b*: *?l* <= *join y z*
by (*rule-tac join-imp-le*, (*blast intro*: *order-trans*)+)
from *a b* **show** *?thesis* **by** (*simp add*: *meet-imp-le*)
qed

lemma *meet-join-eq-imp-le*: $a = c \vee a = d \vee b = c \vee b = d \Longrightarrow$ *meet a b* $\leq$ *join c d*
by (*insert meet-join-le*, *blast intro*: *order-trans*)

lemma *modular-le*: $x \leq z \Longrightarrow$ *join x* (*meet y z*) $\leq$ *meet* (*join x y*) *z* (**is** - $\Longrightarrow$ *?t* <= -)
proof −
assume *a*: *x* <= *z*
have *b*: *?t* <= *join x y* **by** (*rule join-imp-le*, *simp-all add*: *meet-join-le meet-join-eq-imp-le*)
have *c*: *?t* <= *z* **by** (*rule join-imp-le*, *simp-all add*: *meet-join-le a*)

from *b c* **show** *?thesis* **by** (*simp add*: *meet-imp-le*)
qed

end

5 Set: Set theory for higher-order logic

theory *Set*
imports *LOrder*
begin

A set in HOL is simply a predicate.

5.1 Basic syntax

global

typedecl *'a set*
arities *set* :: (*type*) *type*

consts
{} :: *'a set* ({})
UNIV :: *'a set*
insert :: *'a => 'a set => 'a set*
Collect :: *('a => bool) => 'a set* — comprehension
Int :: *'a set => 'a set => 'a set* (**infixl** *70*)
Un :: *'a set => 'a set => 'a set* (**infixl** *65*)
UNION :: *'a set => ('a => 'b set) => 'b set* — general union
INTER :: *'a set => ('a => 'b set) => 'b set* — general intersection
Union :: *'a set set => 'a set* — union of a set
Inter :: *'a set set => 'a set* — intersection of a set
Pow :: *'a set => 'a set set* — powerset
Ball :: *'a set => ('a => bool) => bool* — bounded universal quantifiers

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

syntax
op : :: *'a => 'a set => bool* (*op :*)
consts
op : :: *'a => 'a set => bool* ((-/ : -) [*50*, *51*] *50*) — membership

local

instance *set* :: (*type*) {*ord*, *minus*} **..**

5.2 Additional concrete syntax

syntax
range :: $('a \Rightarrow 'b) \Rightarrow 'b\ set$ — of function

op ~: :: $'a \Rightarrow 'a\ set \Rightarrow bool$ $(op\ \sim:)$ — non-membership
op ~: :: $'a \Rightarrow 'a\ set \Rightarrow bool$ $((-/\ \sim:\ -)\ [50,\ 51]\ 50)$

@Finset :: $args \Rightarrow 'a\ set$ $(\{(-)\})$
@Coll :: $pttrn \Rightarrow bool \Rightarrow 'a\ set$ $((1\{-./\ -\}))$
@SetCompr :: $'a \Rightarrow idts \Rightarrow bool \Rightarrow 'a\ set$ $((1\{-\ |/-./\ -\}))$
@Collect :: $idt \Rightarrow 'a\ set \Rightarrow bool \Rightarrow 'a\ set$ $((1\{-\ :/\ -./\ -\}))$
@INTER1 :: $pttrns \Rightarrow 'b\ set \Rightarrow 'b\ set$ $((3INT\ -./\ -)\ 10)$
@UNION1 :: $pttrns \Rightarrow 'b\ set \Rightarrow 'b\ set$ $((3UN\ -./\ -)\ 10)$
@INTER :: $pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow 'b\ set$ $((3INT\ -:-./\ -)\ 10)$
@UNION :: $pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow 'b\ set$ $((3UN\ -:-./\ -)\ 10)$

-Ball :: $pttrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((3ALL\ -:-./\ -)\ [0,\ 0,\ 10]\ 10)$
-Bex :: $pttrn \Rightarrow 'a\ set \Rightarrow bool \Rightarrow bool$ $((3EX\ -:-./\ -)\ [0,\ 0,\ 10]\ 10)$

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

translations
range f $== f`UNIV$
$x \sim: y$ $== \sim (x : y)$
$\{x,\ xs\}$ $== insert\ x\ \{xs\}$
$\{x\}$ $== insert\ x\ \{\}$
$\{x.\ P\}$ $== Collect\ (\%x.\ P)$
$\{x{:}A.\ P\}$ $=> \{x.\ x{:}A\ \&\ P\}$
$UN\ x\ y.\ B$ $== UN\ x.\ UN\ y.\ B$
$UN\ x.\ B$ $== UNION\ UNIV\ (\%x.\ B)$
$UN\ x.\ B$ $== UN\ x{:}UNIV.\ B$
$INT\ x\ y.\ B$ $== INT\ x.\ INT\ y.\ B$
$INT\ x.\ B$ $== INTER\ UNIV\ (\%x.\ B)$
$INT\ x.\ B$ $== INT\ x{:}UNIV.\ B$
$UN\ x{:}A.\ B$ $== UNION\ A\ (\%x.\ B)$
$INT\ x{:}A.\ B$ $== INTER\ A\ (\%x.\ B)$
$ALL\ x{:}A.\ P$ $== Ball\ A\ (\%x.\ P)$
$EX\ x{:}A.\ P$ $== Bex\ A\ (\%x.\ P)$

syntax (**output**)
-setle :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ $(op <=)$
-setle :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ $((-/\ <=\ -)\ [50,\ 51]\ 50)$
-setless :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ $(op <)$
-setless :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ $((-/\ <\ -)\ [50,\ 51]\ 50)$

syntax (*xsymbols*)
-setle :: $'a\ set \Rightarrow 'a\ set \Rightarrow bool$ $(op \subseteq)$

-setle :: *'a set => 'a set => bool* (*(-/ ⊆ -) [50, 51] 50*)
-setless :: *'a set => 'a set => bool* (*op ⊂*)
-setless :: *'a set => 'a set => bool* (*(-/ ⊂ -) [50, 51] 50*)
op Int :: *'a set => 'a set => 'a set* (**infixl** ∩ *70*)
op Un :: *'a set => 'a set => 'a set* (**infixl** ∪ *65*)
op : :: *'a => 'a set => bool* (*op ∈*)
op : :: *'a => 'a set => bool* (*(-/ ∈ -) [50, 51] 50*)
op ~: :: *'a => 'a set => bool* (*op ∉*)
op ~: :: *'a => 'a set => bool* (*(-/ ∉ -) [50, 51] 50*)
Union :: *'a set set => 'a set* (⋃ *- [90] 90*)
Inter :: *'a set set => 'a set* (⋂ *- [90] 90*)
-Ball :: *pttrn => 'a set => bool => bool* (*(3∀ -∈-./ -) [0, 0, 10] 10*)
-Bex :: *pttrn => 'a set => bool => bool* (*(3∃ -∈-./ -) [0, 0, 10] 10*)

syntax (*HTML* **output**)

-setle :: *'a set => 'a set => bool* (*op ⊆*)
-setle :: *'a set => 'a set => bool* (*(-/ ⊆ -) [50, 51] 50*)
-setless :: *'a set => 'a set => bool* (*op ⊂*)
-setless :: *'a set => 'a set => bool* (*(-/ ⊂ -) [50, 51] 50*)
op Int :: *'a set => 'a set => 'a set* (**infixl** ∩ *70*)
op Un :: *'a set => 'a set => 'a set* (**infixl** ∪ *65*)
op : :: *'a => 'a set => bool* (*op ∈*)
op : :: *'a => 'a set => bool* (*(-/ ∈ -) [50, 51] 50*)
op ~: :: *'a => 'a set => bool* (*op ∉*)
op ~: :: *'a => 'a set => bool* (*(-/ ∉ -) [50, 51] 50*)
-Ball :: *pttrn => 'a set => bool => bool* (*(3∀ -∈-./ -) [0, 0, 10] 10*)
-Bex :: *pttrn => 'a set => bool => bool* (*(3∃ -∈-./ -) [0, 0, 10] 10*)

syntax (*xsymbols*)

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

syntax (*latex* **output**)

@UNION1 :: *pttrns => 'b set => 'b set* (*(3⋃(00_)/ -) 10*)
@INTER1 :: *pttrns => 'b set => 'b set* (*(3⋂(00_)/ -) 10*)
@UNION :: *pttrn => 'a set => 'b set => 'b set* (*(3⋃(00_∈_)/ -) 10*)
@INTER :: *pttrn => 'a set => 'b set => 'b set* (*(3⋂(00_∈_)/ -) 10*)

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.

translations

op ⊆ => op <= :: - set => - set => bool
op ⊂ => op < :: - set => - set => bool

typed-print-translation ⟪
let
fun le-tr′ - (Type (fun, (Type (set, -) :: -))) ts =
list-comb (Syntax.const -setle, ts)
| le-tr′ - - - = raise Match;

fun less-tr′ - (Type (fun, (Type (set, -) :: -))) ts =
list-comb (Syntax.const -setless, ts)
| less-tr′ - - - = raise Match;
in [(op <=, le-tr′), (op <, less-tr′)] end
⟫

5.2.1 Bounded quantifiers

syntax
-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)

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)

syntax (*HOL*)
-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)

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)

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

print-translation ⟪
let
fun
all-tr′ [Const (-bound,-) $ Free (v,Type(T,-)),

```
          Const(op -->,-) $ (Const (op <,-) $ (Const (-bound,-) $ Free (v',-))
$ n ) $ P] =
  (if v=v' andalso T=set
   then Syntax.const -setlessAll $ Syntax.mark-bound v' $ n $ P
   else raise Match)

  | all-tr' [Const (-bound,-) $ Free (v,Type(T,-)),
          Const(op -->,-) $ (Const (op <=,-) $ (Const (-bound,-) $ Free (v',-))
$ n ) $ P] =
  (if v=v' andalso T=set
   then Syntax.const -setleAll $ Syntax.mark-bound v' $ n $ P
   else raise Match);

  fun
    ex-tr' [Const (-bound,-) $ Free (v,Type(T,-)),
          Const(op &,-) $ (Const (op <,-) $ (Const (-bound,-) $ Free (v',-)) $ n
) $ P] =
  (if v=v' andalso T=set
   then Syntax.const -setlessEx $ Syntax.mark-bound v' $ n $ P
   else raise Match)

  | ex-tr' [Const (-bound,-) $ Free (v,Type(T,-)),
         Const(op &,-) $ (Const (op <=,-) $ (Const (-bound,-) $ Free (v',-)) $ n
) $ P] =
  (if v=v' andalso T=set
   then Syntax.const -setleEx $ Syntax.mark-bound v' $ n $ P
   else raise Match)
in
[(ALL , all-tr'), (EX , ex-tr')]
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 $ Abs (, 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
[(*Ball*, *btr' -Ball*),(*Bex*, *btr' -Bex*),
 (*UNION*, *btr' @UNION*),(*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.

axioms
  mem-Collect-eq: (*a* : {*x*. *P*(*x*)}) = *P*(*a*)
  Collect-mem-eq: {*x*. *x*:*A*} = *A*

finalconsts
Collect
op :

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

defs (**overloaded**)
subset-def: *A <= B* == *ALL x:A. x:B*
psubset-def: *A < B* == *(A::'a set) <= B & ~ A=B*
Compl-def: *- A* == *{x. ~x:A}*
set-diff-def: *A - B* == *{x. x:A & ~x:B}*

defs
Un-def: *A Un B* == *{x. x:A | x:B}*
Int-def: *A Int B* == *{x. x:A & x:B}*
INTER-def: *INTER A B* == *{y. ALL x:A. y: B(x)}*
UNION-def: *UNION A B* == *{y. EX x:A. y: B(x)}*
Inter-def: *Inter S* == *(INT x:S. x)*
Union-def: *Union S* == *(UN x:S. x)*
Pow-def: *Pow A* == *{B. B <= A}*
empty-def: *{}* == *{x. False}*
UNIV-def: *UNIV* == *{x. True}*
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

declare *mem-Collect-eq* [*iff*] *Collect-mem-eq* [*simp*]

lemma *CollectI*: *P(a)* ==> *a : {x. P(x)}*
by *simp*

lemma *CollectD*: *a : {x. P(x)}* ==> *P(a)*
by *simp*

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

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

5.4.2 Bounded quantifiers

lemma *ballI* [*intro!*]: (!!*x. x:A* ==> *P x*) ==> *ALL x:A. P x*
by (*simp add*: *Ball-def*)

lemmas *strip = impI allI ballI*

lemma *bspec* [*dest?*]: *ALL x:A. P x ==> x:A ==> P x*
by (*simp add*: *Ball-def*)

lemma *ballE* [*elim*]: *ALL x:A. P x ==> (P x ==> Q) ==> (x ~: A ==> Q) ==> 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 ⟪
local val ballE = thm ballE
in fun ball-tac i = etac ballE i THEN contr-tac (*i + 1*) *end*;
⟫

Gives better instantiation for bound:

ML-setup ⟪
claset-ref() := *claset*() *addbefore* (*bspec*, *datac* (*thm bspec*) *1*);
⟫

lemma *bexI* [*intro*]: *P x ==> x:A ==> 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 ==> P x ==> 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. ~P x ==> P a*) ==> *a:A ==> EX x:A. P x*
by (*unfold Bex-def*) *blast*

lemma *bexE* [*elim!*]: *EX x:A. P x ==> (!!x. x:A ==> P x ==> Q) ==> Q*
by (*unfold Bex-def*) *blast*

lemma *ball-triv* [*simp*]: (*ALL x:A. P*) = ((*EX x. x:A*) −−> *P*)
— Trival rewrite rule.
by (*simp add*: *Ball-def*)

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

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

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

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

ML-setup ⟪
local
val Ball-def = thm Ball-def;
val Bex-def = thm Bex-def;

val simpset = Simplifier.clear-ss HOL-basic-ss;
fun unfold-tac ss th =
ALLGOALS (full-simp-tac (Simplifier.inherit-bounds ss simpset addsimps [*th*]));

fun prove-bex-tac ss =
unfold-tac ss Bex-def THEN Quantifier1.prove-one-point-ex-tac;
val rearrange-bex = Quantifier1.rearrange-bex prove-bex-tac;

fun prove-ball-tac ss =
unfold-tac ss Ball-def THEN Quantifier1.prove-one-point-all-tac;
val rearrange-ball = Quantifier1.rearrange-ball prove-ball-tac;
in
val defBEX-regroup = Simplifier.simproc (Theory.sign-of (the-context ()))
defined BEX [*EX x:A. P x & Q x*] *rearrange-bex*;
val defBALL-regroup = Simplifier.simproc (Theory.sign-of (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*:
$A = B\ \text{==>}\ (!!x.\ x{:}B\ \text{==>}\ P\ x = Q\ x)\ \text{==>}$
$(ALL\ x{:}A.\ P\ x) = (ALL\ x{:}B.\ Q\ x)$
by (*simp add*: *Ball-def*)

lemma *strong-ball-cong* [*cong*]:
$A = B\ \text{==>}\ (!!x.\ x{:}B\ \text{=simp=>}\ P\ x = Q\ x)\ \text{==>}$

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

5.4.4 Subsets

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

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

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

declare *subsetD* [*intro?*] — FIXME

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

declare *rev-subsetD* [*intro?*] — FIXME

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

ML ⟪
local val rev-subsetD = thm rev-subsetD
in fun impOfSubs th = th RSN (*2*, *rev-subsetD*) *end*;
⟫

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 subset-def*) *blast*

Takes assumptions $A \subseteq B$; $c \in A$ and creates the assumption $c \in B$.

ML ⟪
local val subsetCE = thm subsetCE
in fun set-mp-tac i = etac subsetCE i THEN mp-tac i end;

⟫

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

lemma *subset-refl*: $A \subseteq A$
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*)

lemmas *equalityI* [*intro*!] = *subset-antisym*

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 for creating induction formulae – for "pattern matching" on p. To make the induction hypotheses usable, apply *spec* or *bspec* to put universal

quantifiers over the free variables in p.

lemma *setup-induction*: p:A ==> (!!z. z:A ==> $p = z$ --> R) ==> R
by *simp*

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

lemma *eqelem-imp-iff*: $x = y$ ==> $(x : A) = (y : A)$
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*: $A \subseteq$ *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*)

5.4.7 The empty set

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

lemma *emptyE* [*elim!*]: $a : \{\}$ ==> 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* ~= {}
by (*blast elim*: *equalityE*)

5.4.8 The Powerset operator – Pow

lemma *Pow-iff* [*iff*]: (*A* ∈ *Pow B*) = (*A* ⊆ *B*)
by (*simp add*: *Pow-def*)

lemma *PowI*: *A* ⊆ *B* ==> *A* ∈ *Pow B*
by (*simp add*: *Pow-def*)

lemma *PowD*: *A* ∈ *Pow B* ==> *A* ⊆ *B*
by (*simp add*: *Pow-def*)

lemma *Pow-bottom*: {} ∈ *Pow B*
by *simp*

lemma *Pow-top*: *A* ∈ *Pow A*
by (*simp add*: *subset-refl*)

5.4.9 Set complement

lemma *Compl-iff* [*simp*]: (*c* ∈ −*A*) = (*c* ∉ *A*)
by (*unfold Compl-def*) *blast*

lemma *ComplI* [*intro*!]: (*c* ∈ *A* ==> *False*) ==> *c* ∈ −*A*
by (*unfold 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*~:*A*
by (*unfold Compl-def*) *blast*

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

5.4.10 Binary union – Un

lemma *Un-iff* [*simp*]: (*c* : *A Un B*) = (*c*:*A* | *c*:*B*)
by (*unfold Un-def*) *blast*

lemma *UnI1* [*elim?*]: *c*:*A* ==> *c* : *A Un B*
by *simp*

lemma *UnI2* [*elim?*]: *c*:*B* ==> *c* : *A Un B*
by *simp*

Classical introduction rule: no commitment to A vs B.

lemma *UnCI* [*intro*!]: (*c*~:*B* ==> *c*:*A*) ==> *c* : *A Un B*
by *auto*

lemma *UnE* [*elim*!]: *c* : *A Un B* ==> (*c*:*A* ==> *P*) ==> (*c*:*B* ==> *P*) ==> *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* ==> *c*:*B* ==> *c* : *A Int B*
by *simp*

lemma *IntD1*: *c* : *A Int B* ==> *c*:*A*
by *simp*

lemma *IntD2*: *c* : *A Int B* ==> *c*:*B*
by *simp*

lemma *IntE* [*elim*!]: *c* : *A Int B* ==> (*c*:*A* ==> *c*:*B* ==> *P*) ==> *P*
by *simp*

5.4.12 Set difference

lemma *Diff-iff* [*simp*]: (*c* : *A* − *B*) = (*c*:*A* & *c*~:*B*)
by (*unfold set-diff-def*) *blast*

lemma *DiffI* [*intro*!]: *c* : *A* ==> *c* ~: *B* ==> *c* : *A* − *B*
by *simp*

lemma *DiffD1*: *c* : *A* − *B* ==> *c* : *A*
by *simp*

lemma *DiffD2*: *c* : *A* − *B* ==> *c* : *B* ==> *P*
by *simp*

lemma *DiffE* [*elim*!]: *c* : *A* − *B* ==> (*c*:*A* ==> *c*~:*B* ==> *P*) ==> *P*
by *simp*

5.4.13 Augmenting a set – insert

lemma *insert-iff* [*simp*]: $(a : \mathit{insert}\ b\ A) = (a = b \mid a{:}A)$
by (*unfold insert-def*) *blast*

lemma *insertI1*: $a : \mathit{insert}\ a\ B$
by *simp*

lemma *insertI2*: $a : B \Longrightarrow a : \mathit{insert}\ b\ B$
by *simp*

lemma *insertE* [*elim*!]: $a : \mathit{insert}\ b\ A \Longrightarrow (a = b \Longrightarrow P) \Longrightarrow (a{:}A \Longrightarrow P) \Longrightarrow P$
by (*unfold insert-def*) *blast*

lemma *insertCI* [*intro*!]: $(a^{\sim}{:}B \Longrightarrow a = b) \Longrightarrow a{:}\ \mathit{insert}\ b\ B$
— Classical introduction rule.
by *auto*

lemma *subset-insert-iff*: $(A \subseteq \mathit{insert}\ x\ B) = (\mathit{if}\ x{:}A\ \mathit{then}\ A - \{x\} \subseteq B\ \mathit{else}\ A \subseteq B)$
by *auto*

5.4.14 Singletons, using insert

lemma *singletonI* [*intro*!]: $a : \{a\}$
— Redundant? But unlike *insertCI*, it proves the subgoal immediately!
by (*rule insertI1*)

lemma *singletonD* [*dest*!]: $b : \{a\} \Longrightarrow b = a$
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*]: $(\{b\} = \mathit{insert}\ a\ A) = (a = b\ \&\ A \subseteq \{b\})$
by *blast*

lemma *singleton-insert-inj-eq'* [*iff*]: $(\mathit{insert}\ a\ A = \{b\}) = (a = b\ \&\ A \subseteq \{b\})$
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*

5.4.15 Unions of families

$\mathit{UN}\ x{:}A.\ B\ x$ is $\bigcup B$ ' A.

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*!]: $b : (\mathit{UN}\ x{:}A.\ B\ x) \Longrightarrow (!!x.\ x{:}A \Longrightarrow b{:}\ B\ x \Longrightarrow R) \Longrightarrow R$
by (*unfold UNION-def*) *blast*

lemma *UN-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow C\ x = D\ x) \Longrightarrow (\mathit{UN}\ x{:}A.\ C\ x) = (\mathit{UN}\ x{:}B.\ D\ x)$
by (*simp add*: *UNION-def*)

5.4.16 Intersections of families

$\mathit{INT}\ x{:}A.\ B\ x$ is $\bigcap B$ ' A.

lemma *INT-iff* [*simp*]: $(b{:}\ (\mathit{INT}\ x{:}A.\ B\ x)) = (\mathit{ALL}\ x{:}A.\ b{:}\ B\ x)$
by (*unfold INTER-def*) *blast*

lemma *INT-I* [*intro*!]: $(!!x.\ x{:}A \Longrightarrow b{:}\ B\ x) \Longrightarrow b : (\mathit{INT}\ x{:}A.\ B\ x)$
by (*unfold INTER-def*) *blast*

lemma *INT-D* [*elim*]: $b : (\mathit{INT}\ x{:}A.\ B\ x) \Longrightarrow a{:}A \Longrightarrow b{:}\ B\ a$
by *auto*

lemma *INT-E* [*elim*]: $b : (\mathit{INT}\ x{:}A.\ B\ x) \Longrightarrow (b{:}\ B\ a \Longrightarrow R) \Longrightarrow (a^{\sim}{:}A \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination – by the Excluded Middle on $a \in A$.
by (*unfold INTER-def*) *blast*

lemma *INT-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow C\ x = D\ x) \Longrightarrow (\mathit{INT}\ x{:}A.\ C\ x) = (\mathit{INT}\ x{:}B.\ D\ x)$
by (*simp add*: *INTER-def*)

5.4.17 Union

lemma *Union-iff* [*simp*]: $(A : Union\ C) = (EX\ X{:}C.\ A{:}X)$
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*]: $(A : Inter\ C) = (ALL\ X{:}C.\ A{:}X)$
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 $f\ x$.

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-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*: $(a : (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y)) = ((Q \dashrightarrow a : x)\ \&\ (\sim Q \dashrightarrow a : y))$
by (*rule split-if*)

lemmas *split-ifs = if-bool-eq-conj split-if-eq1 split-if-eq2 split-if-mem1 split-if-mem2*

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-setup ⟪
val mksimps-pairs = [(Ball, [thm bspec])] @ mksimps-pairs;
simpset-ref() := simpset() setmksimps (mksimps mksimps-pairs);
⟫

declare *subset-UNIV* [*simp*] *subset-refl* [*simp*]

5.4.20 The "proper subset" relation

lemma *psubsetI* [*intro*!]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$
by (*unfold psubset-def*) *blast*

lemma *psubsetE* [*elim*!]:
$[|A \subset B;\ [|A \subseteq B;\ \sim (B \subseteq A)|] \Longrightarrow R|] \Longrightarrow R$
by (*unfold psubset-def*) *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*: *psubset-def subset-insert-iff*)

lemma *psubset-eq*: $(A \subset B) = (A \subseteq B\ \&\ A \neq B)$
by (*simp only*: *psubset-def*)

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 psubset-def*)
apply (*auto dest*: *subset-antisym*)
done

lemma *psubsetD*: $[|\ A \subset B;\ c \in A\ |] \Longrightarrow c \in B$
apply (*unfold psubset-def*)
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 psubset-def*) *blast*

lemma *atomize-ball*:
$(!!x.\ x \in A \Longrightarrow P\ x) \equiv \mathit{Trueprop}\ (\forall x{\in}A.\ P\ x)$
by (*simp only*: *Ball-def atomize-all atomize-imp*)

declare *atomize-ball* [*symmetric*, *rulify*]

5.5 Further set-theory lemmas

5.5.1 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq \mathit{insert}\ a\ B$
apply (*rule subsetI*)
apply (*erule insertI2*)
done

lemma *subset-insertI2*: $A \subseteq B \Longrightarrow A \subseteq \mathit{insert}\ b\ B$
by *blast*

lemma *subset-insert*: $x \notin A \Longrightarrow (A \subseteq \mathit{insert}\ x\ B) = (A \subseteq B)$
by *blast*

Big Union – least upper bound of a set.

lemma *Union-upper*: $B \in A \Longrightarrow B \subseteq \mathit{Union}\ A$
by (*iprover intro*: *subsetI UnionI*)

lemma *Union-least*: $(!!X.\ X \in A \Longrightarrow X \subseteq C) \Longrightarrow \mathit{Union}\ A \subseteq C$
by (*iprover intro*: *subsetI elim*: *UnionE dest*: *subsetD*)

General union.

lemma *UN-upper*: $a \in A \Longrightarrow B\ a \subseteq (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *UN-least*: $(!!x.\ x \in A \Longrightarrow B\ x \subseteq C) \Longrightarrow (\bigcup x{\in}A.\ B\ x) \subseteq C$
by (*iprover intro*: *subsetI elim*: *UN-E dest*: *subsetD*)

Big Intersection – greatest lower bound of a set.

lemma *Inter-lower*: $B \in A \Longrightarrow Inter\ A \subseteq B$
by *blast*

lemma *Inter-subset*:
$[\!|\ !!X.\ X \in A \Longrightarrow X \subseteq B;\ A \mathrel{\sim=} \{\}\ |\!] \Longrightarrow \bigcap A \subseteq B$
by *blast*

lemma *Inter-greatest*: $(!!X.\ X \in A \Longrightarrow C \subseteq X) \Longrightarrow C \subseteq Inter\ A$
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*

Monotonicity.

lemma *mono-Un*: $mono\ f \Longrightarrow f\ A \cup f\ B \subseteq f\ (A \cup B)$
by (*auto simp add*: *mono-def*)

lemma *mono-Int*: *mono f* ==> *f* $(A \cap B) \subseteq f\ A \cap f\ B$
by (*auto simp add*: *mono-def*)

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 psubset-def*) *blast*

lemma *Collect-empty-eq* [*simp*]: $(\mathit{Collect}\ P = \{\}) = (\forall x.\ \neg\ P\ x)$
by *auto*

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*: $\{x.\ \exists y.\ P\ x\ y\} = (\bigcup y.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-bex-eq*: $\{x.\ \exists y{\in}A.\ P\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ P\ x\ y\})$
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* $\subseteq$ *B*) = (*x* $\in$ *B* & *A* $\subseteq$ *B*)
by *blast*

lemma *mk-disjoint-insert*: *a* $\in$ *A* ==> $\exists$ *B*. *A* = *insert a B* & *a* $\notin$ *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* $\neq$ *a* --> *P u*}
by *auto*

lemma *UN-insert-distrib*: *u* $\in$ *A* ==> ($\bigcup$ *x*$\in$*A*. *insert a* (*B x*)) = *insert a* ($\bigcup$ *x*$\in$*A*. *B x*)
by *blast*

lemma *insert-inter-insert*[*simp*]: *insert a A* $\cap$ *insert a B* = *insert a* (*A* $\cap$ *B*)
by *blast*

lemma *insert-disjoint*[*simp*]:
(*insert a A* $\cap$ *B* = {}) = (*a* $\notin$ *B* $\wedge$ *A* $\cap$ *B* = {})
({} = *insert a A* $\cap$ *B*) = (*a* $\notin$ *B* $\wedge$ {} = *A* $\cap$ *B*)
by *auto*

lemma *disjoint-insert*[*simp*]:
(*B* $\cap$ *insert a A* = {}) = (*a* $\notin$ *B* $\wedge$ *B* $\cap$ *A* = {})
({} = *A* $\cap$ *insert b B*) = (*b* $\notin$ *A* $\wedge$ {} = *A* $\cap$ *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*: $x \in A$ ==> ($\lambda x.\ c$) ‘ $A = \{c\}$
by *auto*

lemma *image-image*: f ‘ (g ‘ A) = ($\lambda x.\ f\ (g\ x)$) ‘ A
by *blast*

lemma *insert-image* [*simp*]: $x \in A$ ==> *insert* (f x) (f‘A) = f‘A
by *blast*

lemma *image-is-empty* [*iff*]: (f‘$A = \{\}$) = ($A = \{\}$)
by *blast*

lemma *image-Collect*: 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*: $M = N$ ==> (!!$x.\ x \in N$ ==> $f\ x = g\ x$) ==> f‘$M = g$‘N
by (*simp add*: *image-def*)

range.

lemma *full-SetCompr-eq*: $\{u.\ \exists x.\ u = f\ x\}$ = *range* f
by *auto*

lemma *range-composition* [*simp*]: *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*]: $(A \cap B = UNIV) = (A = UNIV\ \&\ B = UNIV)$
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*

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* [*iff*]: $(\bigcup A = \{\}) = (\forall x{\in}A.\ x = \{\})$
by *blast*

lemma *empty-Union-conv* [*iff*]: $(\{\} = \bigcup A) = (\forall x{\in}A.\ x = \{\})$
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*]: $\bigcap (insert\ a\ B) = a \cap \bigcap B$
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* [*iff*]:
$(\bigcap A = UNIV) = (\forall x{\in}A.\ x = UNIV)$
$(UNIV = \bigcap A) = (\forall x{\in}A.\ x = UNIV)$
by *blast+*

UN and *INT*.

Basic identities:

lemma *UN-empty* [*simp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
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*]: $(\bigcap x{\in}\{\}.\ B\ x) = UNIV$
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*]: $(\bigcup x{\in}insert\ a\ A.\ B\ x) = B\ a \cup UNION\ A\ B$
by *blast*

lemma *UN-Un*: $(\bigcup i \in A \cup B.\ M\ i) = (\bigcup i{\in}A.\ M\ i) \cup (\bigcup i{\in}B.\ M\ i)$

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*[*iff*]:
$(\{\} = (UN\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = \{\})$
$((UN\ x{:}A.\ B\ x) = \{\}) = (\forall x{\in}A.\ B\ x = \{\})$
by *blast+*

lemma *INTER-UNIV-conv*[*iff*]:

$(UNIV = (INT\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = UNIV)$
$((INT\ x{:}A.\ B\ x) = UNIV) = (\forall x{\in}A.\ B\ x = UNIV)$
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*: $(\exists x \in A \cup B.\ P\ x) = ((\exists x{\in}A.\ P\ x) \mid (\exists x{\in}B.\ P\ x))$
by *blast*

lemma *ball-UN*: $(\forall z \in UNION\ A\ B.\ P\ z) = (\forall x{\in}A.\ \forall z \in B\ x.\ P\ z)$
by *blast*

lemma *bex-UN*: $(\exists z \in UNION\ A\ B.\ P\ z) = (\exists x{\in}A.\ \exists z{\in}B\ x.\ P\ z)$
by *blast*

Set difference.

lemma *Diff-eq*: $A - B = A \cap (-B)$
by *blast*

lemma *Diff-eq-empty-iff* [*simp*]: $(A - B = \{\}) = (A \subseteq B)$
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*]: $x \notin A \Longrightarrow A - insert\ x\ B = A - B$
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*: $x \notin A \Longrightarrow (\mathit{insert}\ x\ A) - \{x\} = A$
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 *all-bool-eq*: $(\forall b{::}bool.\ P\ b) = (P\ True\ \&\ P\ False)$
apply *auto*
apply (*tactic* ⟪ *case-tac b 1* ⟫, *auto*)
done

lemma *bool-induct*: $P\ True \Longrightarrow P\ False \Longrightarrow P\ x$
by (*rule conjI* [*THEN all-bool-eq* [*THEN iffD2*], *THEN spec*])

lemma *ex-bool-eq*: $(\exists b{::}bool.\ P\ b) = (P\ True \mid P\ False)$
apply *auto*
apply (*tactic* ⟪ *case-tac b 1* ⟫, *auto*)
done

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*: $(\bigcup b{::}bool.\ A\ b) = (A\ True \cup A\ False)$
apply *auto*
apply (*tactic* ⟪ *case-tac b 1* ⟫, *auto*)
done

lemma *INT-bool-eq*: $(\bigcap b{::}bool.\ A\ b) = (A\ True \cap A\ False)$
apply *auto*
apply (*tactic* ⟪ *case-tac b 1* ⟫, *auto*)
done

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 psubset-def*) *blast*

lemma *all-not-in-conv* [*iff*]: $(\forall x.\ x \notin A) = (A = \{\})$
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*]:
$!!a\ B\ C.\ (UN\ x{:}C.\ insert\ a\ (B\ x)) = (if\ C{=}\{\}\ then\ \{\}\ else\ insert\ a\ (UN\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Un\ B) = ((if\ C{=}\{\}\ then\ \{\}\ else\ (UN\ x{:}C.\ A\ x)\ Un\ B))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ Un\ B\ x) = ((if\ C{=}\{\}\ then\ \{\}\ else\ A\ Un\ (UN\ x{:}C.\ B\ x)))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Int\ B) = ((UN\ x{:}C.\ A\ x)\ Int\ B)$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ Int\ B\ x) = (A\ Int\ (UN\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x - B) = ((UN\ x{:}C.\ A\ x) - B)$
$!!A\ B\ C.\ (UN\ x{:}C.\ A - B\ x) = (A - (INT\ x{:}C.\ B\ x))$
$!!A\ B.\ (UN\ x{:}\ Union\ A.\ B\ x) = (UN\ y{:}A.\ UN\ x{:}y.\ B\ x)$
$!!A\ B\ C.\ (UN\ z{:}\ UNION\ A\ B.\ C\ z) = (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z)$

!!$A\ B\ f.\ (UN\ x{:}f`A.\ B\ x) \quad = (UN\ a{:}A.\ B\ (f\ a))$
by *auto*

lemma *INT-simps* [*simp*]:
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Int\ B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x)\ Int\ B)$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ Int\ B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A\ Int\ (INT\ x{:}C.\ B\ x))$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x\ -\ B) \quad = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x)\ -\ B)$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ -\ B\ x) \quad = (if\ C{=}\{\}\ then\ UNIV\ else\ A\ -\ (UN\ x{:}C.\ B\ x))$
!!$a\ B\ C.\ (INT\ x{:}C.\ insert\ a\ (B\ x)) = insert\ a\ (INT\ x{:}C.\ B\ x)$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Un\ B) \ = ((INT\ x{:}C.\ A\ x)\ Un\ B)$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ Un\ B\ x) \ = (A\ Un\ (INT\ x{:}C.\ B\ x))$
!!$A\ B.\ (INT\ x{:}\ Union\ A.\ B\ x) = (INT\ y{:}A.\ INT\ x{:}y.\ B\ x)$
!!$A\ B\ C.\ (INT\ z{:}\ UNION\ A\ B.\ C\ z) = (INT\ x{:}A.\ INT\ z{:}\ B(x).\ C\ z)$
!!$A\ B\ f.\ (INT\ x{:}f`A.\ B\ x) \quad = (INT\ a{:}A.\ B\ (f\ a))$
by *auto*

lemma *ball-simps* [*simp*]:
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ |\ Q) = ((ALL\ x{:}A.\ P\ x)\ |\ Q)$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ |\ Q\ x) = (P\ |\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ --{>}\ Q\ x) = (P\ --{>}\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ --{>}\ Q) = ((EX\ x{:}A.\ P\ x)\ --{>}\ Q)$
!!$P.\ (ALL\ x{:}\{\}.\ P\ x) = True$
!!$P.\ (ALL\ x{:}UNIV.\ P\ x) = (ALL\ x.\ P\ x)$
!!$a\ B\ P.\ (ALL\ x{:}insert\ a\ B.\ P\ x) = (P\ a\ \&\ (ALL\ x{:}B.\ P\ x))$
!!$A\ P.\ (ALL\ x{:}Union\ A.\ P\ x) = (ALL\ y{:}A.\ ALL\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (ALL\ x{:}\ UNION\ A\ B.\ P\ x) = (ALL\ a{:}A.\ ALL\ x{:}\ B\ a.\ P\ x)$
!!$P\ Q.\ (ALL\ x{:}Collect\ Q.\ P\ x) = (ALL\ x.\ Q\ x\ --{>}\ P\ x)$
!!$A\ P\ f.\ (ALL\ x{:}f`A.\ P\ x) = (ALL\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(ALL\ x{:}A.\ P\ x)) = (EX\ x{:}A.\ \sim P\ x)$
by *auto*

lemma *bex-simps* [*simp*]:
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ x\ \&\ Q) = ((EX\ x{:}A.\ P\ x)\ \&\ Q)$
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x{:}A.\ Q\ x))$
!!$P.\ (EX\ x{:}\{\}.\ P\ x) = False$
!!$P.\ (EX\ x{:}UNIV.\ P\ x) = (EX\ x.\ P\ x)$
!!$a\ B\ P.\ (EX\ x{:}insert\ a\ B.\ P\ x) = (P(a)\ |\ (EX\ x{:}B.\ P\ x))$
!!$A\ P.\ (EX\ x{:}Union\ A.\ P\ x) = (EX\ y{:}A.\ EX\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (EX\ x{:}\ UNION\ A\ B.\ P\ x) = (EX\ a{:}A.\ EX\ x{:}B\ a.\ P\ x)$
!!$P\ Q.\ (EX\ x{:}Collect\ Q.\ P\ x) = (EX\ x.\ Q\ x\ \&\ P\ x)$
!!$A\ P\ f.\ (EX\ x{:}f`A.\ P\ x) = (EX\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(EX\ x{:}A.\ P\ x)) = (ALL\ x{:}A.\ \sim P\ x)$
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*:
$(EX\ x{:}A.\ P\ x\ |\ Q\ x) = ((EX\ x{:}A.\ P\ x)\ |\ (EX\ x{:}A.\ Q\ x))$
by *blast*

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:
$!!a\ B\ C.\ insert\ a\ (UN\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ \{a\}\ else\ (UN\ x{:}C.\ insert\ a\ (B\ x)))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x)\ Un\ B = (if\ C{=}\{\}\ then\ B\ else\ (UN\ x{:}C.\ A\ x\ Un\ B))$
$!!A\ B\ C.\ A\ Un\ (UN\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (UN\ x{:}C.\ A\ Un\ B\ x))$
$!!A\ B\ C.\ ((UN\ x{:}C.\ A\ x)\ Int\ B) = (UN\ x{:}C.\ A\ x\ Int\ B)$
$!!A\ B\ C.\ (A\ Int\ (UN\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A\ Int\ B\ x)$
$!!A\ B\ C.\ ((UN\ x{:}C.\ A\ x) - B) = (UN\ x{:}C.\ A\ x - B)$
$!!A\ B\ C.\ (A - (INT\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A - B\ x)$
$!!A\ B.\ (UN\ y{:}A.\ UN\ x{:}y.\ B\ x) = (UN\ x{:}\ Union\ A.\ B\ x)$
$!!A\ B\ C.\ (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z) = (UN\ z{:}\ UNION\ A\ B.\ C\ z)$
$!!A\ B\ f.\ (UN\ a{:}A.\ B\ (f\ a)) = (UN\ x{:}f`A.\ B\ x)$
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*: $A \subseteq B ==> f`A \subseteq f`B$
by *blast*

lemma *Pow-mono*: $A \subseteq B ==> Pow\ A \subseteq Pow\ B$
by *blast*

lemma *Union-mono*: $A \subseteq B ==> \bigcup A \subseteq \bigcup B$
by *blast*

lemma *Inter-anti-mono*: $B \subseteq A ==> \bigcap A \subseteq \bigcap B$
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*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cap B \subseteq C \cap D$
by *blast*

lemma *Diff-mono*: $A \subseteq C \Longrightarrow D \subseteq B \Longrightarrow A - B \subseteq C - D$
by *blast*

lemma *Compl-anti-mono*: $A \subseteq B \Longrightarrow -B \subseteq -A$
by *blast*

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B \Longrightarrow x \in A \longrightarrow x \in B$
apply (*rule impI*)
apply (*erule subsetD*, *assumption*)
done

lemma *conj-mono*: *P1* $\longrightarrow$ *Q1* $\Longrightarrow$ *P2* $\longrightarrow$ *Q2* $\Longrightarrow$ (*P1* & *P2*) $\longrightarrow$ (*Q1* & *Q2*)
by *iprover*

lemma *disj-mono*: *P1* $\longrightarrow$ *Q1* $\Longrightarrow$ *P2* $\longrightarrow$ *Q2* $\Longrightarrow$ (*P1* | *P2*) $\longrightarrow$ (*Q1* | *Q2*)
by *iprover*

lemma *imp-mono*: *Q1* $\longrightarrow$ *P1* $\Longrightarrow$ *P2* $\longrightarrow$ *Q2* $\Longrightarrow$ (*P1* $\longrightarrow$ *P2*) $\longrightarrow$ (*Q1* $\longrightarrow$ *Q2*)
by *iprover*

lemma *imp-refl*: *P* $\longrightarrow$ *P* **..**

lemma *ex-mono*: $(!!x.\ P\ x \longrightarrow Q\ x) \Longrightarrow (EX\ x.\ P\ x) \longrightarrow (EX\ x.\ Q\ x)$
by *iprover*

lemma *all-mono*: $(!!x.\ P\ x\ \longrightarrow\ Q\ x) \Longrightarrow (ALL\ x.\ P\ x) \longrightarrow (ALL\ x.\ Q\ x)$
by *iprover*

lemma *Collect-mono*: $(!!x.\ P\ x\ \longrightarrow\ Q\ x) \Longrightarrow Collect\ P \subseteq Collect\ Q$
by *blast*

lemma *Int-Collect-mono*:
$A \subseteq B \Longrightarrow (!!x.\ x \in A \Longrightarrow P\ x\ \longrightarrow\ Q\ x) \Longrightarrow A \cap Collect\ P \subseteq B \cap Collect\ Q$
by *blast*

lemmas *basic-monos* =
subset-refl imp-refl disj-mono conj-mono
ex-mono Collect-mono in-mono

lemma *eq-to-mono*: $a = b \Longrightarrow c = d \Longrightarrow b \longrightarrow d \Longrightarrow a \longrightarrow c$
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*

lemma *Least-mono*:
mono $(f::'a::order \Rightarrow 'b::order) \Longrightarrow EX\ x:S.\ ALL\ y:S.\ x <= y$
$\Longrightarrow (LEAST\ y.\ y : f\ `\ S) = f\ (LEAST\ x.\ x : S)$
— 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

5.6 Inverse image of a function

constdefs
vimage :: $('a \Rightarrow 'b) \Rightarrow 'b\ set \Rightarrow 'a\ set$ (**infixr** $-`$ *90*)
$f\ -`\ B == \{x.\ f\ x : B\}$

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* ==> (!!*x*. *f a* = *x* ==> *x*:*B* ==> *P*) ==> *P*
by (*unfold vimage-def*) *blast*

lemma *vimageD*: *a* : *f* −‘ *A* ==> *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*: $A \subseteq B \Longrightarrow f\ -`\ A \subseteq f\ -`\ B$
— monotonicity
by *blast*

5.7 Getting the Contents of a Singleton Set

constdefs
contents :: *'a set* => *'a*
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:$A \Longrightarrow A \subseteq B \Longrightarrow x$:$B$
by (*rule subsetD*)

lemma *set-mp*: $A \subseteq B \Longrightarrow x$:$A \Longrightarrow x$:$B$
by (*rule subsetD*)

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

lemma *ord-eq-le-trans*: $a = b \Longrightarrow b <= c \Longrightarrow a <= c$
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*)

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*: $(a::'a::order) <= b ==> f\ b < (c::'c::order) ==>$
$(!!x\ y.\ x <= y ==> f\ x <= f\ y) ==> f\ a < c$
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-le-less-trans*) **show** *?thesis* **.**
qed

lemma *order-le-less-subst1*: $(a::'a::order) <= f\ b ==> (b::'b::order) < c ==>$
$(!!x\ y.\ x < y ==> f\ x < f\ y) ==> a < f\ c$
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 (*order-le-less-trans*) **show** *?thesis* **.**
qed

lemma *order-less-le-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-less-le-trans*) **show** *?thesis* **.**
qed

lemma *order-less-le-subst1*: $(a::'a::order) < f\ b ==> (b::'b::order) <= c ==>$
$(!!x\ y.\ x <= y ==> f\ x <= f\ y) ==> a < f\ c$
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 (*order-less-le-trans*) **show** *?thesis* **.**
qed

lemma *order-subst1*: $(a::'a::order) <= f\ b ==> (b::'b::order) <= c ==>$
$(!!x\ y.\ x <= y ==> f\ x <= f\ y) ==> a <= f\ c$
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 (*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 *basic-trans-rules* [*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
set-rev-mp
set-mp
order-neq-le-trans
order-le-neq-trans
order-less-trans
order-less-asym'
order-le-less-trans
order-less-le-trans
order-trans
order-antisym
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans
trans

end

6 Typedef: HOL type definitions

theory *Typedef*
imports *Set*
uses (*Tools/typedef-package.ML*)
begin

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!

lemma (**in** *type-definition*) *Rep-inject*:
  (*Rep x* = *Rep y*) = (*x* = *y*)

proof
  assume *Rep x = Rep y*
  hence *Abs (Rep x) = Abs (Rep y)* **by** (*simp only*:)
  also have *Abs (Rep x) = x* **by** (*rule Rep-inverse*)
  also have *Abs (Rep y) = y* **by** (*rule Rep-inverse*)
  finally show *x = y* **.**
next
  assume *x = y*
  thus *Rep x = Rep y* **by** (*simp only*:)
qed

lemma (**in** *type-definition*) *Abs-inject*:
  assumes *x*: *x ∈ A* **and** *y*: *y ∈ A*
  shows *(Abs x = Abs y) = (x = y)*
proof
  assume *Abs x = Abs y*
  hence *Rep (Abs x) = Rep (Abs y)* **by** (*simp only*:)
  also from *x* **have** *Rep (Abs x) = x* **by** (*rule Abs-inverse*)
  also from *y* **have** *Rep (Abs y) = y* **by** (*rule Abs-inverse*)
  finally show *x = y* **.**
next
  assume *x = y*
  thus *Abs x = Abs y* **by** (*simp only*:)
qed

lemma (**in** *type-definition*) *Rep-cases* [*cases set*]:
  assumes *y*: *y ∈ A*
    and *hyp*: *!!x. y = Rep x ==> P*
  shows *P*
proof (*rule hyp*)
  from *y* **have** *Rep (Abs y) = y* **by** (*rule Abs-inverse*)
  thus *y = Rep (Abs y)* **..**
qed

lemma (**in** *type-definition*) *Abs-cases* [*cases type*]:
  assumes *r*: *!!y. x = Abs y ==> y ∈ A ==> P*
  shows *P*
proof (*rule r*)
  have *Abs (Rep x) = x* **by** (*rule Rep-inverse*)
  thus *x = Abs (Rep x)* **..**
  show *Rep x ∈ A* **by** (*rule Rep*)
qed

lemma (**in** *type-definition*) *Rep-induct* [*induct set*]:
  assumes *y*: *y ∈ A*
    and *hyp*: *!!x. P (Rep x)*
  shows *P y*
proof −
  have *P (Rep (Abs y))* **by** (*rule hyp*)

```
  also from y have Rep (Abs y) = y by (rule Abs-inverse)
  finally show P y .
qed

lemma (in type-definition) Abs-induct [induct type]:
  assumes r: !!y. y ∈ A ==> P (Abs y)
  shows P x
proof −
  have Rep x ∈ A by (rule Rep)
  hence P (Abs (Rep x)) by (rule r)
  also have Abs (Rep x) = x by (rule Rep-inverse)
  finally show P x .
qed

use Tools/typedef-package.ML

setup TypedefPackage.setup

end



theory Fun
imports Typedef
begin

instance set :: (type) order
  by (intro-classes,
      (assumption | rule subset-refl subset-trans subset-antisym psubset-eq)+)

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



constdefs
```

override-on :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b)$
$override\text{-}on\ f\ g\ A == \%a.\ if\ a : A\ then\ g\ a\ else\ f\ a$

id :: $'a \Rightarrow 'a$
$id == \%x.\ x$

comp :: $['b \Rightarrow 'c, 'a \Rightarrow 'b, 'a] \Rightarrow 'c$ (**infixl** *o 55*)
$f\ o\ g == \%x.\ f(g(x))$

compatibility

lemmas *o-def = comp-def*

syntax (*xsymbols*)
comp :: $['b \Rightarrow 'c, 'a \Rightarrow 'b, 'a] \Rightarrow 'c$ (**infixl** $\circ$ *55*)
syntax (*HTML* **output**)
comp :: $['b \Rightarrow 'c, 'a \Rightarrow 'b, 'a] \Rightarrow 'c$ (**infixl** $\circ$ *55*)

constdefs
inj-on :: $['a \Rightarrow 'b, 'a\ set] \Rightarrow bool$
$inj\text{-}on\ f\ A == !\ x{:}A.\ !\ y{:}A.\ f(x){=}f(y) \longrightarrow x{=}y$

A common special case: functions injective over the entire domain type.

syntax *inj* :: $('a \Rightarrow 'b) \Rightarrow bool$
translations
$inj\ f == inj\text{-}on\ f\ UNIV$

constdefs
surj :: $('a \Rightarrow 'b) \Rightarrow bool$
$surj\ f == !\ y.\ ?\ x.\ y{=}f(x)$

bij :: $('a \Rightarrow 'b) \Rightarrow bool$
$bij\ f == inj\ f\ \&\ surj\ f$

As a simplification rule, it replaces all function equalities by first-order equalities.

lemma *expand-fun-eq*: $(f = g) = (!\ x.\ f(x){=}g(x))$
apply (*rule iffI*)
apply (*simp* (*no-asm-simp*))
apply (*rule ext*, *simp* (*no-asm-simp*))
done

lemma *apply-inverse*:
$[|\ f(x){=}u;\ !!x.\ P(x) \Longrightarrow g(f(x)) = x;\ P(x)\ |] \Longrightarrow x{=}g(u)$
by *auto*

The Identity Function: *id*

lemma *id-apply* [*simp*]: $id\ x = x$

by (*simp add*: *id-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*)

6.1 The Composition Operator: $f \circ g$

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 *image-eq-UN*: *f*'*A* = (*UN x*:*A*. {*f x*})
by *blast*

lemma *UN-o*: *UNION A* (*g o f*) = *UNION* (*f*'*A*) *g*
by (*unfold comp-def*, *blast*)

6.2 The Injectivity Predicate, *inj*

NB: *inj* now just translates to *inj-on*

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*)

6.3 The Predicate *inj-on*: Injectivity On A Restricted Domain

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

6.4 The Predicate *surj*: Surjectivity

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

6.5 The Predicate *bij*: Bijectivity

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*)

6.6 Facts About the Identity Function

We seem to need both the *id* forms and the $\lambda x.\ x$ forms. The latter can arise by rewriting, while *id* may be used explicitly.

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*)

lemma *vimage-image-eq*: $f\ -`\ (f\ `\ A) = \{y.\ EX\ x{:}A.\ f\ x = f\ y\}$
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 *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 *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 *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-UN*: *(f ‘ (UNION A B)) = (UN x:A.(f ‘ (B x)))*
by *blast*

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

6.7 Function Updating

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*

6.8 *override-on*

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*)

6.9 swap

constdefs
swap :: ['*a*, '*a*, '*a* => '*b*] => ('*a* => '*b*)
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 (*rule-tac P* = *y* = *f b* **in** *case-split-thm*, *blast*)
apply (*rule-tac P* = *y* = *f a* **in** *case-split-thm*, *auto*)
— We don't yet have *case-tac*
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*)

The ML section includes some compatibility bindings and a simproc for function updates, in addition to the usual ML-bindings of theorems.

ML
```
⟪
val id-def = thm id-def;
val inj-on-def = thm inj-on-def;
val surj-def = thm surj-def;
val bij-def = thm bij-def;
val fun-upd-def = thm fun-upd-def;

val o-def = thm comp-def;
val injI = thm inj-onI;
val inj-inverseI = thm inj-on-inverseI;
val set-cs = claset() delrules [equalityI];

val print-translation = [(Pi, dependent-tr' (@Pi, op funcset))];

(* simplifies terms of the form f(...,x:=y,...,x:=z,...) to f(...,x:=z,...) *)
local
  fun gen-fun-upd NONE T - - = NONE
    | gen-fun-upd (SOME f) T x y = SOME (Const (Fun.fun-upd,T) $ f $ x $ y)
  fun dest-fun-T1 (Type (-, T :: Ts)) = T
  fun find-double (t as Const (Fun.fun-upd,T) $ f $ x $ y) =
    let
      fun find (Const (Fun.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

  val current-ss = simpset ()
  fun fun-upd-prover ss =
    rtac eq-reflection 1 THEN rtac ext 1 THEN
    simp-tac (Simplifier.inherit-bounds ss current-ss) 1
in
  val fun-upd2-simproc =
    Simplifier.simproc (Theory.sign-of (the-context ()))
      fun-upd2 [f(v := w, x := y)]
      (fn sg => fn ss => fn t =>
        case find-double t of (T, NONE) => NONE
        | (T, SOME rhs) =>
          SOME (Tactic.prove sg [] [] (Term.equals T $ t $ rhs) (K (fun-upd-prover
```

ss))))
end;
Addsimprocs[*fun-upd2-simproc*];

val expand-fun-eq = *thm expand-fun-eq*;
val apply-inverse = *thm apply-inverse*;
val id-apply = *thm id-apply*;
val o-apply = *thm o-apply*;
val o-assoc = *thm o-assoc*;
val id-o = *thm id-o*;
val o-id = *thm o-id*;
val image-compose = *thm image-compose*;
val image-eq-UN = *thm image-eq-UN*;
val UN-o = *thm UN-o*;
val datatype-injI = *thm datatype-injI*;
val injD = *thm injD*;
val inj-eq = *thm inj-eq*;
val inj-onI = *thm inj-onI*;
val inj-on-inverseI = *thm inj-on-inverseI*;
val inj-onD = *thm inj-onD*;
val inj-on-iff = *thm inj-on-iff*;
val comp-inj-on = *thm comp-inj-on*;
val inj-on-contraD = *thm inj-on-contraD*;
val inj-singleton = *thm inj-singleton*;
val subset-inj-on = *thm subset-inj-on*;
val surjI = *thm surjI*;
val surj-range = *thm surj-range*;
val surjD = *thm surjD*;
val surjE = *thm surjE*;
val comp-surj = *thm comp-surj*;
val bijI = *thm bijI*;
val bij-is-inj = *thm bij-is-inj*;
val bij-is-surj = *thm bij-is-surj*;
val image-ident = *thm image-ident*;
val image-id = *thm image-id*;
val vimage-ident = *thm vimage-ident*;
val vimage-id = *thm vimage-id*;
val vimage-image-eq = *thm vimage-image-eq*;
val image-vimage-subset = *thm image-vimage-subset*;
val image-vimage-eq = *thm image-vimage-eq*;
val surj-image-vimage-eq = *thm surj-image-vimage-eq*;
val inj-vimage-image-eq = *thm inj-vimage-image-eq*;
val vimage-subsetD = *thm vimage-subsetD*;
val vimage-subsetI = *thm vimage-subsetI*;
val vimage-subset-eq = *thm vimage-subset-eq*;
val image-Int-subset = *thm image-Int-subset*;
val image-diff-subset = *thm image-diff-subset*;
val inj-on-image-Int = *thm inj-on-image-Int*;
val inj-on-image-set-diff = *thm inj-on-image-set-diff*;

val image-Int = thm image-Int;
val image-set-diff = thm image-set-diff;
val inj-image-mem-iff = thm inj-image-mem-iff;
val inj-image-subset-iff = thm inj-image-subset-iff;
val inj-image-eq-iff = thm inj-image-eq-iff;
val image-UN = thm image-UN;
val image-INT = thm image-INT;
val bij-image-INT = thm bij-image-INT;
val surj-Compl-image-subset = thm surj-Compl-image-subset;
val inj-image-Compl-subset = thm inj-image-Compl-subset;
val bij-image-Compl-eq = thm bij-image-Compl-eq;
val fun-upd-idem-iff = thm fun-upd-idem-iff;
val fun-upd-idem = thm fun-upd-idem;
val fun-upd-apply = thm fun-upd-apply;
val fun-upd-same = thm fun-upd-same;
val fun-upd-other = thm fun-upd-other;
val fun-upd-upd = thm fun-upd-upd;
val fun-upd-twist = thm fun-upd-twist;
val range-ex1-eq = thm range-ex1-eq;
⟩⟩

end

7 Product-Type: Cartesian products

theory *Product-Type*
imports *Fun*
uses (*Tools/split-rule.ML*)
begin

7.1 Unit

typedef *unit* = {*True*}
proof
  show *True* : *?unit* **by** *blast*
qed

constdefs
  Unity :: *unit* ("'(')")
  () == *Abs-unit True*

lemma *unit-eq*: *u* = ()
  by (*induct u*) (*simp add*: *unit-def Unity-def*)

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

ML-setup ⟨⟨
  val unit-eq-proc =

```
  let val unit-meta-eq = mk-meta-eq (thm unit-eq) in
    Simplifier.simproc (Theory.sign-of (the-context ())) 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];
⟫

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)

lemma unit-induct [induct type: unit]: P () ==> P x
 by simp
```

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]: (%u::unit. f ()) = f
 by (rule ext) simp
```

7.2 Pairs

7.2.1 Type definition

```
constdefs
 Pair-Rep :: ['a, 'b] => ['a, 'b] => bool
 Pair-Rep == (%a b. %x y. x=a & y=b)

global

typedef (Prod)
 ('a, 'b) *    (infixr 20)
   = {f. EX a b. f = Pair-Rep (a::'a) (b::'b)}
proof
 fix a b show Pair-Rep a b : ?Prod
   by blast
qed

syntax (xsymbols)
 *      :: [type, type] => type          ((- ×/ -) [21, 20] 20)
syntax (HTML output)
 *      :: [type, type] => type          ((- ×/ -) [21, 20] 20)

local
```

7.2.2 Abstract constants and syntax

global

consts
```
  fst      :: 'a * 'b => 'a
  snd      :: 'a * 'b => 'b
  split    :: ['a, 'b] => 'c, 'a * 'b] => 'c
  curry    :: ['a * 'b => 'c, 'a, 'b] => 'c
  prod-fun :: ['a => 'b, 'c => 'd, 'a * 'c] => 'b * 'd
  Pair     :: ['a, 'b] => 'a * 'b
  Sigma    :: ['a set, 'a => 'b set] => ('a * 'b) set
```

local

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     (-,/ -)
  @Sigma ::[pttrn, 'a set, 'b set] => ('a * 'b) set ((3SIGMA -:-./ -) 10)
  @Times ::['a set,  'a => 'b set] => ('a * 'b) set (infixr <*> 80)
```

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


  SIGMA x:A. B => Sigma A (%x. B)
  A <*> B      => Sigma A (-K B)
```

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
⟫
```

Deleted x-symbol and html support using Σ (Sigma) because of the danger of confusion with Sum.

syntax (*xsymbols*)

@Times :: [*'a set*, *'a => 'b set*] => (*'a * 'b*) *set* (- × - [*81*, *80*] *80*)

syntax (*HTML* **output**)
@Times :: [*'a set*, *'a => 'b set*] => (*'a * 'b*) *set* (- × - [*81*, *80*] *80*)

print-translation ⟪ [(*Sigma*, *dependent-tr'* (*@Sigma*, *@Times*))] ⟫

7.2.3 Definitions

defs
Pair-def: *Pair a b == Abs-Prod*(*Pair-Rep a b*)
fst-def: *fst p == THE a. EX b. p = (a, b)*
snd-def: *snd p == THE b. EX a. p = (a, b)*
split-def: *split == (%c p. c (fst p) (snd p))*
curry-def: *curry == (%c x y. c (x,y))*
prod-fun-def: *prod-fun f g == split*(*%x y.*(*f*(*x*), *g*(*y*)))
Sigma-def: *Sigma A B == UN x*:*A. UN y*:*B*(*x*). {(*x*, *y*)}

7.2.4 Lemmas and proof tool setup

lemma *ProdI*: *Pair-Rep a b* : *Prod*
by (*unfold Prod-def*) *blast*

lemma *Pair-Rep-inject*: *Pair-Rep a b = Pair-Rep a' b' ==> a = a' & b = b'*
apply (*unfold Pair-Rep-def*)
apply (*drule fun-cong* [*THEN fun-cong*], *blast*)
done

lemma *inj-on-Abs-Prod*: *inj-on Abs-Prod Prod*
apply (*rule inj-on-inverseI*)
apply (*erule Abs-Prod-inverse*)
done

lemma *Pair-inject*:
(*a*, *b*) = (*a'*, *b'*) ==> (*a = a' ==> b = b' ==> R*) ==> *R*
proof −
case *rule-context* [*unfolded Pair-def*]
show *?thesis*
apply (*rule inj-on-Abs-Prod* [*THEN inj-onD*, *THEN Pair-Rep-inject*, *THEN conjE*])
apply (*rule rule-context ProdI*)+
.
qed

lemma *Pair-eq* [*iff*]: ((*a*, *b*) = (*a'*, *b'*)) = (*a = a' & b = b'*)
by (*blast elim*!: *Pair-inject*)

lemma *fst-conv* [*simp*]: *fst* (*a*, *b*) = *a*
by (*unfold fst-def*) *blast*

lemma *snd-conv* [*simp*]: *snd* (*a*, *b*) = *b*
by (*unfold snd-def*) *blast*

lemma *fst-eqD*: *fst* (*x*, *y*) = *a* ==> *x* = *a*
by *simp*

lemma *snd-eqD*: *snd* (*x*, *y*) = *a* ==> *y* = *a*
by *simp*

lemma *PairE-lemma*: *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*: ∗]: (!!*x y*. *p* = (*x*, *y*) ==> *Q*) ==> *Q*
by (*insert PairE-lemma* [*of p*]) *blast*

ML ⟪
local val PairE = *thm PairE in*
fun pair-tac s =
EVERY' [*res-inst-tac* [(*p*, *s*)] *PairE*, *hyp-subst-tac*, *K prune-params-tac*];
end;
⟫

lemma *surjective-pairing*: *p* = (*fst p*, *snd p*)
— Do not add as rewrite rule: invalidates some proofs in IMP
by (*cases p*) *simp*

lemmas *pair-collapse* = *surjective-pairing* [*symmetric*]
declare *pair-collapse* [*simp*]

lemma *surj-pair* [*simp*]: *EX x y*. *z* = (*x*, *y*)
apply (*rule exI*)
apply (*rule exI*)
apply (*rule surjective-pairing*)
done

lemma *split-paired-all*: (!!*x*. *PROP P x*) == (!!*a b*. *PROP P* (*a*, *b*))
proof
fix *a b*
assume !!*x*. *PROP P x*
thus *PROP P* (*a*, *b*) **.**
next
fix *x*
assume !!*a b*. *PROP P* (*a*, *b*)
hence *PROP P* (*fst x*, *snd x*) **.**

thus *PROP P x* **by** *simp*
qed

lemmas *split-tupled-all = split-paired-all unit-all-eq2*

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.

ML-setup ⟪
(∗ *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 (#*prop* (*Thm.rep-thm th*)) *then full-simplify ss th else th*;
end;

claset-ref() := *claset*() *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 *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 *curryI* [*intro*!]: *f* (*a*,*b*) ==> *curry f a b*
by (*simp add*: *curry-def*)

lemma *curryD* [*dest*!]: *curry f a b* ==> *f* (*a*,*b*)
by (*simp add*: *curry-def*)

lemma *curryE*: [| *curry f a b* ; *f* (*a*,*b*) ==> *Q* |] ==> *Q*

by (*simp add*: *curry-def*)

lemma *curry-conv* [*simp*]: *curry f a b* = *f* (*a*,*b*)
by (*simp add*: *curry-def*)

lemma *prod-induct* [*induct type*: ∗]: !!*x*. (!!*a b*. *P* (*a*, *b*)) ==> *P x*
by *fast*

lemma *split-paired-Ex* [*simp*]: (*EX x*. *P x*) = (*EX a b*. *P* (*a*, *b*))
by *fast*

lemma *split-conv* [*simp*]: *split c* (*a*, *b*) = *c a b*
by (*simp add*: *split-def*)

lemmas *split* = *split-conv* — for backwards compatibility

lemmas *splitI* = *split-conv* [*THEN iffD2*, *standard*]
lemmas *splitD* = *split-conv* [*THEN iffD1*, *standard*]

lemma *split-Pair-apply*: *split* (%*x y*. *f* (*x*, *y*)) = *f*
— Subsumes the old *split-Pair* when *f* is the identity function.
apply (*rule ext*)
apply (*tactic* ⟪ *pair-tac x 1* ⟫, *simp*)
done

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-Pair-apply*)

lemma *The-split*: *The* (*split P*) = (*THE xy*. *P* (*fst xy*) (*snd xy*))
by (*simp add*: *split-def*)

lemma *Pair-fst-snd-eq*: !!*s t*. (*s* = *t*) = (*fst s* = *fst t* & *snd s* = *snd t*)
by (*simp only*: *split-tupled-all*, *simp*)

lemma *prod-eqI* [*intro?*]: *fst p* = *fst q* ==> *snd p* = *snd q* ==> *p* = *q*
by (*simp add*: *Pair-fst-snd-eq*)

lemma *split-weak-cong*: *p* = *q* ==> *split c p* = *split c q*
— Prevents simplification of *c*: much faster
by (*erule arg-cong*)

lemma *split-eta*: (%(*x*, *y*). *f* (*x*, *y*)) = *f*
apply (*rule ext*)
apply (*simp only*: *split-tupled-all*)
apply (*rule split-conv*)
done

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-setup ⟪

```
local
  val cond-split-eta = 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 thy ss lhs rhs = mk-meta-eq (Tactic.prove thy [] []
        (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs,rhs)))
      (K (simp-tac (Simplifier.inherit-bounds ss HOL-basic-ss addsimps [cond-split-eta])
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 thy ss (s as Const (split, -) $ Abs (-, -, t) $ arg) =
        (case split-pat beta-term-pat 1 t of
        SOME (i,f) => SOME (metaeq thy ss s (subst arg 0 i f))
        | NONE => NONE)
  |   beta-proc - - - = NONE;
  fun eta-proc thy ss (s as Const (split, -) $ Abs (-, -, t)) =
        (case split-pat eta-term-pat 1 t of
          SOME (-,ft) => SOME (metaeq thy ss s (let val (f $ arg) = ft in f end))
        | NONE => NONE)
  |   eta-proc - - - = NONE;
in
  val split-beta-proc = Simplifier.simproc (Theory.sign-of (the-context ()))
    split-beta [split f z] beta-proc;
  val split-eta-proc = Simplifier.simproc (Theory.sign-of (the-context ()))
```

```
    split-eta [split f] eta-proc;
end;

Addsimprocs [split-beta-proc, split-eta-proc];
⟩⟩
```

lemma *split-beta*: (%(*x*, *y*). *P x y*) *z* = *P* (*fst z*) (*snd z*)
by (*subst surjective-pairing*, *rule split-conv*)

lemma *split-split*: *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*: *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*: [| *z*: *split c p*; !!*x y*. [| *p* = (*x*,*y*); *z*: *c x y* |] ==> *Q* |] ==> *Q*
proof −
case *rule-context* [*unfolded split-def*]
show *?thesis* **by** (*rule rule-context surjective-pairing*)+
qed

declare *mem-splitI2* [*intro*!] *mem-splitI* [*intro*!] *splitI2'* [*intro*!] *splitI2* [*intro*!] *splitI* [*intro*!]
declare *mem-splitE* [*elim*!] *splitE'* [*elim*!] *splitE* [*elim*!]

ML-setup ⟪
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;
(∗ *This prevents applications of splitE for already splitted arguments leading to quite time−consuming computations* (*in particular for nested tuples*) ∗)
claset-ref() := *claset*() *addSbefore* (*split-conv-tac*, *split-conv-tac*);
⟫

lemma *split-eta-SetCompr* [*simp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P* (*x*, *y*)) = *P*
by (*rule ext*, *fast*)

lemma *split-eta-SetCompr2* [*simp*]: (%*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.
apply (*rule ext*, *blast*)
done

lemma *split-comp-eq*:
(%*u*. *f* (*g* (*fst u*)) (*snd u*)) = (*split* (%*x*. *f* (*g x*)))
by (*rule ext*, *auto*)

lemma *The-split-eq* [*simp*]: (*THE* (*x'*,*y'*). *x* = *x'* & *y* = *y'*) = (*x*, *y*)
by *blast*

lemma *injective-fst-snd*: !!*x y*. [|*fst x* = *fst y*; *snd x* = *snd y*|] ==> *x* = *y*
by *auto*

prod-fun — action of the product functor upon functions.

lemma *prod-fun* [*simp*]: *prod-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*)
apply (*rule ext*)
apply (*tactic* ⟪ *pair-tac x 1* ⟫, *simp*)
done

lemma *prod-fun-ident* [*simp*]: *prod-fun* (%*x*. *x*) (%*y*. *y*) = (%*z*. *z*)
apply (*rule ext*)
apply (*tactic* ⟪ *pair-tac z 1* ⟫, *simp*)
done

lemma *prod-fun-imageI* [*intro*]: (*a*, *b*) : *r* ==> (*f a*, *g b*) : *prod-fun f g* ‘ *r*
apply (*rule image-eqI*)
apply (*rule prod-fun* [*symmetric*], *assumption*)
done

lemma *prod-fun-imageE* [*elim*!]:
[| *c*: (*prod-fun f g*)‘*r*; !!*x y*. [| *c*=(*f*(*x*),*g*(*y*)); (*x*,*y*):*r* |] ==> *P*
|] ==> *P*
proof −
case *rule-context*
assume *major*: *c*: (*prod-fun f g*)‘*r*
show *?thesis*
apply (*rule major* [*THEN imageE*])
apply (*rule-tac p* = *x* **in** *PairE*)
apply (*rule rule-context*)
prefer *2*
apply *blast*
apply (*blast intro*: *prod-fun*)
done
qed

constdefs
upd-fst :: $('a \Rightarrow 'c) \Rightarrow 'a * 'b \Rightarrow 'c * 'b$
upd-fst f == *prod-fun f id*

upd-snd :: $('b \Rightarrow 'c) \Rightarrow 'a * 'b \Rightarrow 'a * 'c$
upd-snd f == *prod-fun id f*

lemma *upd-fst-conv* [*simp*]: *upd-fst f* $(x,y) = (f\ x,y)$
by (*simp add*: *upd-fst-def*)

lemma *upd-snd-conv* [*simp*]: *upd-snd f* $(x,y) = (x,f\ y)$
by (*simp add*: *upd-snd-def*)

Disjoint union of a family of sets – Sigma.

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*

Complex rules for Sigma.

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*]:
(*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*]:
(*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 matching, 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 *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

Setup of internal *split-rule*.

constdefs
internal-split :: $('a => 'b => 'c) => 'a * 'b => 'c$
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*

7.3 Code generator setup

types-code
```
  *     ((- */ -))
```
attach (*term-of*) ⟪
```
fun term-of-id-42 f T g U (x, y) = HOLogic.pair-const T U $ f x $ g y;
```
⟫
attach (*test*) ⟪
```
fun gen-id-42 aG bG i = (aG i, bG i);
```
⟫

consts-code
```
  Pair    ((-,/ -))
  fst     (fst)
  snd     (snd)
```

ML ⟪
```
local

fun strip-abs 0 t = ([], t)
  | strip-abs i (Abs (s, T, t)) =
    let
      val s' = Codegen.new-name t s;
      val v = Free (s', T)
    in apfst (cons v) (strip-abs (i-1) (subst-bound (v, t))) end
  | strip-abs i (u as Const (split, -) $ t) = (case strip-abs (i+1) t of
       (v :: v' :: vs, u) => (HOLogic.mk-prod (v, v') :: vs, u)
     | - => ([], u))
  | strip-abs i t = ([], t);

fun let-codegen thy defs gr dep thyname brack t = (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 1 u of
             ([p], u') => apfst (cons (p, t)) (dest-let u')
           | - => ([], l))
        | dest-let t = ([], t);
      fun mk-code (gr, (l, r)) =
        let
          val (gr1, pl) = Codegen.invoke-codegen thy defs dep thyname false (gr, l);
          val (gr2, pr) = Codegen.invoke-codegen thy defs dep thyname false (gr1,
r);
        in (gr2, (pl, pr)) end
```

```
    in case dest-let (t1 $ t2 $ t3) of
        ([], -) => NONE
      | (ps, u) =>
          let
            val (gr1, qs) = foldl-map mk-code (gr, ps);
           val (gr2, pu) = Codegen.invoke-codegen thy defs dep thyname false (gr1,
u);
            val (gr3, pargs) = foldl-map
              (Codegen.invoke-codegen thy defs dep thyname true) (gr2, ts)
          in
            SOME (gr3, Codegen.mk-app brack
              (Pretty.blk (0, [Pretty.str let , Pretty.blk (0, List.concat
                  (separate [Pretty.str ;, Pretty.brk 1] (map (fn (pl, pr) =>
                    [Pretty.block [Pretty.str val , pl, Pretty.str  =,
                      Pretty.brk 1, pr]]) qs))),
                Pretty.brk 1, Pretty.str in , pu,
                Pretty.brk 1, Pretty.str end])) pargs)
          end
    end
  | - => NONE);

fun split-codegen thy defs gr dep thyname brack t = (case strip-comb t of
    (t1 as Const (split, -), t2 :: ts) =>
      (case strip-abs 1 (t1 $ t2) of
        ([p], u) =>
          let
            val (gr1, q) = Codegen.invoke-codegen thy defs dep thyname false (gr,
p);
            val (gr2, pu) = Codegen.invoke-codegen thy defs dep thyname false (gr1,
u);
            val (gr3, pargs) = foldl-map
              (Codegen.invoke-codegen thy defs dep thyname true) (gr2, ts)
          in
            SOME (gr2, Codegen.mk-app brack
              (Pretty.block [Pretty.str (fn , q, Pretty.str  =>,
                Pretty.brk 1, pu, Pretty.str )]) pargs)
          end
      | - => NONE)
  | - => NONE);

in

val prod-codegen-setup =
  [Codegen.add-codegen let-codegen let-codegen,
   Codegen.add-codegen split-codegen split-codegen];

end;
⟩⟩
```

setup *prod-codegen-setup*

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 PairE-lemma = thm PairE-lemma;
val Pair-Rep-inject = thm Pair-Rep-inject;
val Pair-def = thm Pair-def;
val Pair-eq = thm Pair-eq;
val Pair-fst-snd-eq = thm Pair-fst-snd-eq;
val Pair-inject = thm Pair-inject;
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 injective-fst-snd = thm injective-fst-snd;

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-Pair-apply = thm split-Pair-apply;
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;
val unit-induct = thm unit-induct;

⟫

end

8 FixedPoint: Fixed Points and the Knaster-Tarski Theorem

theory *FixedPoint*
imports *Product-Type*
begin

constdefs
  lfp :: [*'a set* ⇒ *'a set*] ⇒ *'a set*
    lfp(*f*) == *Inter*({*u*. *f*(*u*) ⊆ *u*}) — least fixed point

  gfp :: [*'a set*=>*'a set*] => *'a set*
    gfp(*f*) == *Union*({*u*. *u* ⊆ *f*(*u*)})

8.1 Proof of Knaster-Tarski Theorem using *lfp*

lfp f is the least upper bound of the set {*u*. *f u* ⊆ *u*}

lemma *lfp-lowerbound*: *f*(*A*) ⊆ *A* ==> *lfp*(*f*) ⊆ *A*
by (*auto simp add*: *lfp-def*)

lemma *lfp-greatest*: [| !!*u*. *f*(*u*) ⊆ *u* ==> *A*⊆*u* |] ==> *A* ⊆ *lfp*(*f*)
by (*auto simp add*: *lfp-def*)

lemma *lfp-lemma2*: *mono*(*f*) ==> *f*(*lfp*(*f*)) ⊆ *lfp*(*f*)
by (*iprover intro*: *lfp-greatest subset-trans monoD lfp-lowerbound*)

lemma *lfp-lemma3*: *mono*(*f*) ==> *lfp*(*f*) ⊆ *f*(*lfp*(*f*))
by (*iprover intro*: *lfp-lemma2 monoD lfp-lowerbound*)

lemma *lfp-unfold*: *mono*(*f*) ==> *lfp*(*f*) = *f*(*lfp*(*f*))
by (*iprover intro*: *equalityI lfp-lemma2 lfp-lemma3*)

8.2 General induction rules for greatest fixed points

lemma *lfp-induct*:
  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*)
apply (*rule-tac a=a* **in** *Int-lower2* [*THEN subsetD*, *THEN CollectD*])
apply (*rule lfp* [*THEN* [*2*] *lfp-lowerbound* [*THEN subsetD*]])
apply (*rule Int-greatest*)
 apply (*rule subset-trans* [*OF Int-lower1* [*THEN mono* [*THEN monoD*]]

mono [*THEN lfp-lemma2*]])
apply (*blast intro*: *indhyp*)
done

Version of induction for binary relations

lemmas *lfp-induct2* = *lfp-induct* [*of* (*a*,*b*), *split-format* (*complete*)]

lemma *lfp-ordinal-induct*:
assumes *mono*: *mono f*
shows [| !!*S*. *P S* ==> *P*(*f S*); !!*M*. !*S*:*M*. *P S* ==> *P*(*Union M*) |]
==> *P*(*lfp f*)
apply(*subgoal-tac lfp f* = *Union*{*S*. *S* ⊆ *lfp f* & *P S*})
apply (*erule ssubst*, *simp*)
apply(*subgoal-tac Union*{*S*. *S* ⊆ *lfp f* & *P S*} ⊆ *lfp f*)
prefer *2* **apply** *blast*
apply(*rule equalityI*)
prefer *2* **apply** *assumption*
apply(*drule mono* [*THEN monoD*])
apply (*cut-tac mono* [*THEN lfp-unfold*], *simp*)
apply (*rule lfp-lowerbound*, *auto*)
done

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*); *a*:*A*;
!!*x*. [| *x*: *f*(*A Int* {*x*. *P*(*x*)}) |] ==> *P*(*x*)
|] ==> *P*(*a*)
by (*blast intro*: *lfp-induct*)

lemma *lfp-mono*: [| !!*Z*. *f*(*Z*)⊆*g*(*Z*) |] ==> *lfp*(*f*) ⊆ *lfp*(*g*)
by (*rule lfp-lowerbound* [*THEN lfp-greatest*], *blast*)

8.3 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*)

lemma *gfp-least*: [| !!*u*. *u* ⊆ *f*(*u*) ==> *u*⊆*X* |] ==> *gfp*(*f*) ⊆ *X*
by (*auto simp add*: *gfp-def*)

lemma *gfp-lemma2*: *mono*(*f*) ==> *gfp*(*f*) ⊆ *f*(*gfp*(*f*))
by (*iprover intro*: *gfp-least subset-trans monoD gfp-upperbound*)

lemma *gfp-lemma3*: $mono(f) ==> f(gfp(f)) \subseteq gfp(f)$
by (*iprover intro*: *gfp-lemma2 monoD gfp-upperbound*)

lemma *gfp-unfold*: $mono(f) ==> gfp(f) = f(gfp(f))$
by (*iprover intro*: *equalityI gfp-lemma2 gfp-lemma3*)

8.4 Coinduction rules for greatest fixed points

weak version

lemma *weak-coinduct*: $[| a: X;\ X \subseteq f(X) |] ==> a : gfp(f)$
by (*rule gfp-upperbound* [*THEN subsetD*], *auto*)

lemma *weak-coinduct-image*: $!!X.\ [| a : X; g`X \subseteq f\ (g`X) |] ==> g\ a : gfp\ f$
apply (*erule gfp-upperbound* [*THEN subsetD*])
apply (*erule imageI*)
done

lemma *coinduct-lemma*:
$[| X \subseteq f(X\ Un\ gfp(f));\ mono(f) |] ==> X\ Un\ gfp(f) \subseteq f(X\ Un\ gfp(f))$
by (*blast dest*: *gfp-lemma2 mono-Un*)

strong version, thanks to Coen and Frost

lemma *coinduct*: $[| mono(f);\ a: X;\ X \subseteq f(X\ Un\ gfp(f)) |] ==> a : gfp(f)$
by (*blast intro*: *weak-coinduct* [*OF - coinduct-lemma*])

lemma *gfp-fun-UnI2*: $[| mono(f);\ a: gfp(f) |] ==> a: f(X\ Un\ gfp(f))$
by (*blast dest*: *gfp-lemma2 mono-Un*)

8.5 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition $X \subseteq f\ X$ 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*:
$[| X \subseteq f(lfp(\%x.\ f(x)\ Un\ X\ Un\ gfp(f)));\ mono(f) |]$
$==> lfp(\%x.\ f(x)\ Un\ X\ Un\ gfp(f)) \subseteq f(lfp(\%x.\ f(x)\ Un\ X\ Un\ gfp(f)))$
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*, *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*); *a*:*X*; *X* ⊆ *f*(*X Un A*) |] ==> *a*: *A*
by (*auto intro*!: *coinduct*)

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*, *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*)

ML
⟪
val lfp-def = *thm lfp-def*;
val lfp-lowerbound = *thm lfp-lowerbound*;
val lfp-greatest = *thm lfp-greatest*;
val lfp-unfold = *thm lfp-unfold*;
val lfp-induct = *thm lfp-induct*;
val lfp-induct2 = *thm lfp-induct2*;
val lfp-ordinal-induct = *thm lfp-ordinal-induct*;
val def-lfp-unfold = *thm def-lfp-unfold*;
val def-lfp-induct = *thm def-lfp-induct*;
val lfp-mono = *thm lfp-mono*;
val gfp-def = *thm gfp-def*;
val gfp-upperbound = *thm gfp-upperbound*;
val gfp-least = *thm gfp-least*;
val gfp-unfold = *thm gfp-unfold*;
val weak-coinduct = *thm weak-coinduct*;
val weak-coinduct-image = *thm weak-coinduct-image*;
val coinduct = *thm coinduct*;

val gfp-fun-UnI2 = thm gfp-fun-UnI2;
val coinduct3 = thm coinduct3;
val def-gfp-unfold = thm def-gfp-unfold;
val def-coinduct = thm def-coinduct;
val def-Collect-coinduct = thm def-Collect-coinduct;
val def-coinduct3 = thm def-coinduct3;
val gfp-mono = thm gfp-mono;
⟫

end

9 Sum-Type: The Disjoint Sum of Two Types

theory *Sum-Type*
imports *Product-Type*
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

9.1 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*: *inj*(*Inl*)
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*: *inj*(*Inr*)
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*)

9.2 The Disjoint Sum of Sets

lemma *InlI* [*intro*!]: *a* : *A* ==> *Inl*(*a*) : *A* <+> *B*
by (*simp add*: *Plus-def*)

lemma *InrI* [*intro*!]: *b* : *B* ==> *Inr*(*b*) : *A* <+> *B*
by (*simp add*: *Plus-def*)

lemma *PlusE* [*elim*!]:
[| *u*: *A* <+> *B*;
!!*x*. [| *x*:*A*; *u*=*Inl*(*x*) |] ==> *P*;
!!*y*. [| *y*:*B*; *u*=*Inr*(*y*) |] ==> *P*
|] ==> *P*
by (*auto simp add*: *Plus-def*)

Exhaustion rule for sums, a degenerate form of induction

lemma *sumE*:
[| !!*x*::'*a*. *s* = *Inl*(*x*) ==> *P*; !!*y*::'*b*. *s* = *Inr*(*y*) ==> *P*
|] ==> *P*
apply (*rule Abs-Sum-cases* [*of s*])
apply (*auto simp add*: *Sum-def Inl-def Inr-def*)
done

lemma *sum-induct*: [| !!*x*. *P* (*Inl x*); !!*x*. *P* (*Inr x*) |] ==> *P x*

by (*rule sumE* [*of x*], *auto*)

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

9.3 The *Part* Primitive

lemma *Part-eqI* [*intro*]: [| *a : A*; *a=h(b)* |] ==> *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)* |] ==> *P* |] ==> *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 ==> Part A h <= Part B h*
by *blast*

lemmas *basic-monos = basic-monos Part-mono*

lemma *PartD1*: *a : Part A h ==> 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 sum-induct = thm sum-induct;
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

10 Relation: Relations

theory *Relation*
imports *Product-Type*
begin

10.1 Definitions

constdefs
  converse :: $('a * 'b)\ set => ('b * 'a)\ set$ $((\text{-}\hat{}{-}1)\ [1000]\ 999)$
  $r\hat{}{-}1 == \{(y, x).\ (x, y) : r\}$
syntax (*xsymbols*)
  converse :: $('a * 'b)\ set => ('b * 'a)\ set$ $((\text{-}^{-1})\ [1000]\ 999)$

constdefs
  rel-comp :: $[('b * 'c)\ set, ('a * 'b)\ set] => ('a * 'c)\ set$ (**infixr** *O 60*)
  $r\ O\ s == \{(x,z).\ EX\ y.\ (x, y) : s\ \&\ (y, z) : r\}$

Image :: $[('a * 'b)\ set,\ 'a\ set] => 'b\ set$ (**infixl** *" 90*)
r *"* $s == \{y.\ EX\ x{:}s.\ (x,y){:}r\}$

Id :: $('a * 'a)\ set$ — the identity relation
$Id == \{p.\ EX\ x.\ p = (x,x)\}$

diag :: $'a\ set => ('a * 'a)\ set$ — diagonal: identity over a set
$diag\ A == \bigcup x{\in}A.\ \{(x,x)\}$

Domain :: $('a * 'b)\ set => 'a\ set$
$Domain\ r == \{x.\ EX\ y.\ (x,y){:}r\}$

Range :: $('a * 'b)\ set => 'b\ set$
$Range\ r == Domain(r\hat{}{-}1)$

Field :: $('a * 'a)\ set => 'a\ set$
$Field\ r == Domain\ r \cup Range\ r$

refl :: $['a\ set,\ ('a * 'a)\ set] => bool$ — reflexivity over a set
$refl\ A\ r == r \subseteq A \times A\ \&\ (ALL\ x{:}\ A.\ (x,x) : r)$

sym :: $('a * 'a)\ set => bool$ — symmetry predicate
$sym\ r == ALL\ x\ y.\ (x,y){:}\ r\ {-}{-}{>}\ (y,x){:}\ r$

antisym:: $('a * 'a)\ set => bool$ — antisymmetry predicate
$antisym\ r == ALL\ x\ y.\ (x,y){:}r\ {-}{-}{>}\ (y,x){:}r\ {-}{-}{>}\ x{=}y$

trans :: $('a * 'a)\ set => bool$ — transitivity predicate
$trans\ r == (ALL\ x\ y\ z.\ (x,y){:}r\ {-}{-}{>}\ (y,z){:}r\ {-}{-}{>}\ (x,z){:}r)$

single-valued :: $('a * 'b)\ set => bool$
$single\text{-}valued\ r == ALL\ x\ y.\ (x,y){:}r\ {-}{-}{>}\ (ALL\ z.\ (x,z){:}r\ {-}{-}{>}\ y{=}z)$

inv-image :: $('b * 'b)\ set => ('a => 'b) => ('a * 'a)\ set$
$inv\text{-}image\ r\ f == \{(x,\ y).\ (f\ x,\ f\ y) : r\}$

syntax
reflexive :: $('a * 'a)\ set => bool$ — reflexivity over a type
translations
$reflexive == refl\ UNIV$

10.2 The identity relation

lemma *IdI* [*intro*]: $(a,\ a) : Id$
by (*simp add*: *Id-def*)

lemma *IdE* [*elim*!]: $p : Id\ ==>\ (!!x.\ p = (x,\ x)\ ==>\ P)\ ==>\ P$
by (*unfold Id-def*) (*iprover elim*: *CollectE*)

lemma *pair-in-Id-conv* [*iff*]: $((a, b) : Id) = (a = b)$
by (*unfold Id-def*) *blast*

lemma *reflexive-Id*: *reflexive Id*
by (*simp add*: *refl-def*)

lemma *antisym-Id*: *antisym Id*
— A strange result, since *Id* is also symmetric.
by (*simp add*: *antisym-def*)

lemma *trans-Id*: *trans Id*
by (*simp add*: *trans-def*)

10.3 Diagonal: identity over a set

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

lemma *diag-eqI*: $a = b \Longrightarrow a : A \Longrightarrow (a, b) : diag\ A$
by (*simp add*: *diag-def*)

lemma *diagI* [*intro*!]: $a : A \Longrightarrow (a, a) : diag\ A$
by (*rule diag-eqI*) (*rule refl*)

lemma *diagE* [*elim*!]:
$c : diag\ A \Longrightarrow (!!x.\ x : A \Longrightarrow c = (x, x) \Longrightarrow P) \Longrightarrow P$
— The general elimination rule.
by (*unfold diag-def*) (*iprover elim*!: *UN-E singletonE*)

lemma *diag-iff*: $((x, y) : diag\ A) = (x = y\ \&\ x : A)$
by *blast*

lemma *diag-subset-Times*: $diag\ A \subseteq A \times A$
by *blast*

10.4 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 *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*

10.5 Reflexivity

lemma *reflI*: $r \subseteq A \times A \Longrightarrow (!!x.\ x : A \Longrightarrow (x, x) : r) \Longrightarrow refl\ A\ r$
by (*unfold refl-def*) (*iprover intro*!: *ballI*)

lemma *reflD*: $refl\ A\ r \Longrightarrow a : A \Longrightarrow (a, a) : r$
by (*unfold refl-def*) *blast*

10.6 Antisymmetry

lemma *antisymI*:
$(!!x\ y.\ (x, y) : r \Longrightarrow (y, x) : r \Longrightarrow x=y) \Longrightarrow antisym\ r$
by (*unfold antisym-def*) *iprover*

lemma *antisymD*: $antisym\ r \Longrightarrow (a, b) : r \Longrightarrow (b, a) : r \Longrightarrow a = b$
by (*unfold antisym-def*) *iprover*

10.7 Symmetry and Transitivity

lemma *symD*: $sym\ r \Longrightarrow (a, b) : r \Longrightarrow (b, a) : r$
by (*unfold sym-def*, *blast*)

lemma *transI*:
$(!!x\ y\ z.\ (x, y) : r \Longrightarrow (y, z) : r \Longrightarrow (x, z) : r) \Longrightarrow trans\ r$
by (*unfold trans-def*) *iprover*

lemma *transD*: $trans\ r \Longrightarrow (a, b) : r \Longrightarrow (b, c) : r \Longrightarrow (a, c) : r$
by (*unfold trans-def*) *iprover*

10.8 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-Id* [*simp*]: $Id\hat{}{-}1 = Id$
by *blast*

lemma *converse-diag* [*simp*]: $(diag\ A)\hat{}{-}1 = diag\ A$
by *blast*

lemma *refl-converse*: $refl\ A\ r \Longrightarrow refl\ A\ (converse\ r)$
by (*unfold refl-def*) *blast*

lemma *antisym-converse*: $antisym\ (converse\ r) = antisym\ r$
by (*unfold antisym-def*) *blast*

lemma *trans-converse*: $trans\ (converse\ r) = trans\ r$
by (*unfold trans-def*) *blast*

10.9 Domain

lemma *Domain-iff*: $(a : Domain\ r) = (EX\ y.\ (a,\ y) : r)$
by (*unfold Domain-def*) *blast*

lemma *DomainI* [*intro*]: $(a,\ b) : r \Longrightarrow a : Domain\ r$
by (*iprover intro*!: *iffD2* [*OF Domain-iff*])

lemma *DomainE* [*elim*!]:
$a : Domain\ r \Longrightarrow (!!y.\ (a,\ y) : r \Longrightarrow P) \Longrightarrow P$
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-diag* [*simp*]: *Domain* (*diag 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-mono*: *r* ⊆ *s* ==> *Domain r* ⊆ *Domain s*
by *blast*

10.10 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-diag* [*simp*]: *Range* (*diag A*) = *A*
by *auto*

lemma *Range-Un-eq*: $Range(A \cup B) = Range(A) \cup Range(B)$
by *blast*

lemma *Range-Int-subset*: $Range(A \cap B) \subseteq Range(A) \cap Range(B)$
by *blast*

lemma *Range-Diff-subset*: $Range(A) - Range(B) \subseteq Range(A - B)$
by *blast*

lemma *Range-Union*: $Range\ (Union\ S) = (\bigcup A{\in}S.\ Range\ A)$
by *blast*

10.11 Image of a set under a relation

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*]: $(a,\ b) : r \Longrightarrow a : A \Longrightarrow b : r``A$
by (*unfold Image-def*) *blast*

lemma *ImageE* [*elim*!]:
$b : r\ ``\ A \Longrightarrow (!!x.\ (x,\ b) : r \Longrightarrow x : A \Longrightarrow P) \Longrightarrow P$
by (*unfold Image-def*) (*iprover elim*!: *CollectE bexE*)

lemma *rev-ImageI*: $a : A \Longrightarrow (a,\ b) : r \Longrightarrow b : r\ ``\ A$
— 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-diag* [*simp*]: $diag\ A\ ``\ B = A \cap B$
by *blast*

lemma *Image-Int-subset*: $R\ ``\ (A \cap B) \subseteq R\ ``\ A \cap R\ ``\ B$
by *blast*

lemma *Image-Int-eq*:
$single\text{-}valued\ (converse\ R) \Longrightarrow R\ ``\ (A \cap B) = R\ ``\ A \cap R\ ``\ B$
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*: $r \subseteq A \times B \Longrightarrow r``C \subseteq B$
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*: $r' \subseteq r \Longrightarrow A' \subseteq A \Longrightarrow (r' `` A') \subseteq (r `` A)$
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*:
$[|single\text{-}valued\ (r^{-1}); A{\neq}\{\}|] \Longrightarrow r `` INTER\ A\ B = (\bigcap x{\in}A.\ r `` B\ x)$
apply (*rule equalityI*)
apply (*rule Image-INT-subset*)
apply (*simp add*: *single-valued-def*, *blast*)
done

lemma *Image-subset-eq*: $(r``A \subseteq B) = (A \subseteq -((r\hat{}{-}1) `` (-B)))$
by *blast*

10.12 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*)

10.13 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*} `` *A* = {*y*. *EX x*:*A*. *P x y*}
by *auto*

10.14 Inverse image

lemma *trans-inv-image*: *trans r* ==> *trans* (*inv-image r f*)
apply (*unfold trans-def inv-image-def*)
apply (*simp* (*no-asm*))
apply *blast*
done

end

theory *Record*
imports *Product-Type*
uses (*Tools/record-package.ML*)
begin

ML ⟪
val [*h1*, *h2*] = *Goal PROP Goal* (⋀*x*. *PROP P x*) ⟹ (*PROP P x* ⟹ *PROP Q*)
⟹ *PROP Q*;
by (*rtac h2 1*);
by (*rtac* (*gen-all* (*h1 RS Drule.rev-triv-goal*)) *1*);
qed meta-allE;
⟫

lemma *prop-subst*: *s* = *t* ⟹ *PROP P t* ⟹ *PROP P s*
by *simp*

lemma *rec-UNIV-I*: ⋀*x*. *x*∈*UNIV* ≡ *True*
by *simp*

lemma *rec-True-simp*: (*True* ⟹ *PROP P*) ≡ *PROP P*
by *simp*

10.15 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
```

11 Inductive: Support for inductive sets and types

```
theory Inductive
imports FixedPoint Sum-Type Relation Record
uses
  (Tools/inductive-package.ML)
  (Tools/inductive-realizer.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-realizer.ML)
  (Tools/datatype-package.ML)
  (Tools/datatype-codegen.ML)
  (Tools/recfun-codegen.ML)
  (Tools/primrec-package.ML)
begin
```

11.1 Inductive sets

Inversion of injective functions.

constdefs
  myinv :: $('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$
  myinv $(f :: 'a \Rightarrow 'b) \equiv \lambda y.\ THE\ x.\ f\ x = y$

lemma *myinv-f-f*: $inj\ f \Longrightarrow myinv\ f\ (f\ x) = x$
proof −
  assume *inj f*
  hence $(THE\ x'.\ f\ x' = f\ x) = (THE\ x'.\ x' = x)$
    by (*simp only*: *inj-eq*)
  also have $\ldots = x$ **by** (*rule the-eq-trivial*)
  finally show *?thesis* **by** (*unfold myinv-def*)
qed

lemma *f-myinv-f*: $inj\ f \Longrightarrow y \in range\ f \Longrightarrow f\ (myinv\ f\ y) = y$
proof (*unfold myinv-def*)
  assume *inj*: *inj f*
  assume $y \in range\ f$
  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.

use *Tools/inductive-package.ML*
setup *InductivePackage.setup*

theorems *basic-monos* [*mono*] =
  subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-def2
  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-rulify2

11.2 Inductive datatypes and primitive recursion

Package setup.

use *Tools/recfun-codegen.ML*

setup *RecfunCodegen.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-realizer.ML*
use *Tools/datatype-package.ML*
setup *DatatypePackage.setup*

use *Tools/datatype-codegen.ML*
setup *DatatypeCodegen.setup*

use *Tools/inductive-realizer.ML*
setup *InductiveRealizer.setup*

use *Tools/inductive-codegen.ML*
setup *InductiveCodegen.setup*

use *Tools/primrec-package.ML*

end

12 Transitive-Closure: Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
imports *Inductive*
uses (*../Provers/trancl.ML*)
begin

rtrancl is reflexive/transitive closure, *trancl* is transitive closure, *reftcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

consts
  rtrancl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ ((-^*) [*1000*] *999*)

inductive $r\hat{}*$
  intros
    rtrancl-refl [*intro*!, *Pure.intro*!, *simp*]: $(a, a) : r\hat{}*$
    rtrancl-into-rtrancl [*Pure.intro*]: $(a, b) : r\hat{}* \Longrightarrow (b, c) : r \Longrightarrow (a, c) : r\hat{}*$

consts
  trancl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ ((-^+) [*1000*] *999*)

inductive $r\hat{}+$

intros
r-into-trancl [*intro*, *Pure.intro*]: $(a, b) : r \Longrightarrow (a, b) : r\hat{}+$
trancl-into-trancl [*Pure.intro*]: $(a, b) : r\hat{}+ \Longrightarrow (b, c) : r \Longrightarrow (a,c) : r\hat{}+$

syntax
-reflcl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ $((\text{-}\hat{}=)\ [1000]\ 999)$
translations
$r\hat{}= == r \cup Id$

syntax (*xsymbols*)
rtrancl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ $((\text{-}^*)\ [1000]\ 999)$
trancl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ $((\text{-}^+)\ [1000]\ 999)$
-reflcl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ $((\text{-}^=)\ [1000]\ 999)$

syntax (*HTML* **output**)
rtrancl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ $((\text{-}^*)\ [1000]\ 999)$
trancl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ $((\text{-}^+)\ [1000]\ 999)$
-reflcl :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ $((\text{-}^=)\ [1000]\ 999)$

12.1 Reflexive-transitive closure

lemma *r-into-rtrancl* [*intro*]: $!!p.\ p \in r \Longrightarrow p \in r\hat{}*$
— *rtrancl* of r contains r
apply (*simp only*: *split-tupled-all*)
apply (*erule rtrancl-refl* [*THEN rtrancl-into-rtrancl*])
done

lemma *rtrancl-mono*: $r \subseteq s \Longrightarrow r\hat{}* \subseteq s\hat{}*$
— monotonicity of *rtrancl*
apply (*rule subsetI*)
apply (*simp only*: *split-tupled-all*)
apply (*erule rtrancl.induct*)
apply (*rule-tac* [*2*] *rtrancl-into-rtrancl*, *blast*+)
done

theorem *rtrancl-induct* [*consumes 1*, *induct set*: *rtrancl*]:
assumes *a*: $(a, b) : r\hat{}*$
and *cases*: $P\ a\ !!y\ z.\ [|\ (a, y) : r\hat{}*; (y, z) : r; P\ y\ |] \Longrightarrow P\ z$
shows $P\ b$
proof −
from *a* **have** $a = a \longrightarrow P\ b$
by (*induct* %*x y*. $x = a \longrightarrow P\ y$ *a b*) (*iprover intro*: *cases*)+
thus *?thesis* **by** *iprover*
qed

lemmas *rtrancl-induct2* =
rtrancl-induct[*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *trans-rtrancl*: *trans*($r\hat{}*$)
— transitivity of transitive closure!! – by induction
proof (*rule transI*)
fix *x y z*
assume $(x, y) \in r^*$
assume $(y, z) \in r^*$
thus $(x, z) \in r^*$ **by** *induct* (*iprover*!)+
qed

lemmas *rtrancl-trans* = *trans-rtrancl* [*THEN transD*, *standard*]

lemma *rtranclE*:
[| (a::'a,b) : $r\hat{}*$; ($a = b$) ==> P;
!!y.[| (a,y) : $r\hat{}*$; (y,b) : r |] ==> P
|] ==> P
— elimination of *rtrancl* – by induction on a special formula
proof −
assume *major*: (a::'a,b) : $r\hat{}*$
case *rule-context*
show *?thesis*
apply (*subgoal-tac* (a::'a) = b | (*EX* y. (a,y) : $r\hat{}*$ & (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 prems*)+
done
qed

lemma *converse-rtrancl-into-rtrancl*:
(a, b) $\in r \Longrightarrow$ (b, c) $\in r^* \Longrightarrow$ (a, c) $\in r^*$
by (*rule rtrancl-trans*) *iprover*+

More r^* equations and inclusions.

lemma *rtrancl-idemp* [*simp*]: ($r\hat{}*$)$\hat{}*$ = $r\hat{}*$
apply *auto*
apply (*erule rtrancl-induct*)
apply (*rule rtrancl-refl*)
apply (*blast intro*: *rtrancl-trans*)
done

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*: $r \subseteq s\hat{}* ==> r\hat{}* \subseteq s\hat{}*$
by (*drule rtrancl-mono*, *simp*)

lemma *rtrancl-subset*: $R \subseteq S \Longrightarrow S \subseteq R\hat{}* \Longrightarrow S\hat{}* = R\hat{}*$
apply (*drule rtrancl-mono*)
apply (*drule rtrancl-mono*, *simp*)
done

lemma *rtrancl-Un-rtrancl*: $(R\hat{}* \cup S\hat{}*)\hat{}* = (R \cup S)\hat{}*$
by (*blast intro*!: *rtrancl-subset intro*: *r-into-rtrancl rtrancl-mono* [*THEN subsetD*])

lemma *rtrancl-reflcl* [*simp*]: $(R\hat{}=)\hat{}* = R\hat{}*$
by (*blast intro*!: *rtrancl-subset intro*: *r-into-rtrancl*)

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*, *blast*)
apply (*blast intro*!: *r-into-rtrancl*)
done

theorem *rtrancl-converseD*:
assumes *r*: $(x, y) \in (r\hat{}-1)\hat{}*$
shows $(y, x) \in r\hat{}*$
proof −
from *r* **show** *?thesis*
by *induct* (*iprover intro*: *rtrancl-trans dest*!: *converseD*)+
qed

theorem *rtrancl-converseI*:
assumes *r*: $(y, x) \in r\hat{}*$
shows $(x, y) \in (r\hat{}-1)\hat{}*$
proof −
from *r* **show** *?thesis*
by *induct* (*iprover intro*: *rtrancl-trans converseI*)+
qed

lemma *rtrancl-converse*: $(r\hat{}-1)\hat{}* = (r\hat{}*)\hat{}-1$
by (*fast dest*!: *rtrancl-converseD intro*!: *rtrancl-converseI*)

theorem *converse-rtrancl-induct*[*consumes 1*]:
assumes *major*: $(a, b) : r\hat{}*$
and *cases*: $P\ b\ !!y\ z.\ [|\ (y, z) : r;\ (z, b) : r\hat{}*;\ P\ z\ |] \Longrightarrow P\ y$
shows $P\ a$
proof −
from *rtrancl-converseI* [*OF major*]
show *?thesis*
by *induct* (*iprover intro*: *cases dest*!: *converseD rtrancl-converseD*)+
qed

lemmas *converse-rtrancl-induct2* =

converse-rtrancl-induct[*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *converse-rtranclE*:
[| (*x*,*z*):*r*^*;
x=*z* ==> *P*;
!!*y*. [| (*x*,*y*):*r*; (*y*,*z*):*r*^* |] ==> *P*
|] ==> *P*
proof −
assume *major*: (*x*,*z*):*r*^*
case *rule-context*
show *?thesis*
apply (*subgoal-tac x* = *z* | (*EX y*. (*x*,*y*) : *r* & (*y*,*z*) : *r*^*))
apply (*rule-tac* [*2*] *major* [*THEN converse-rtrancl-induct*])
prefer *2* **apply** *iprover*
prefer *2* **apply** *iprover*
apply (*erule asm-rl exE disjE conjE prems*)+
done
qed

ML-setup ⟪
bind-thm (*converse-rtranclE2*, *split-rule*
(*read-instantiate* [(*x*,(*xa*,*xb*)), (*z*,(*za*,*zb*))] (*thm converse-rtranclE*)));
⟫

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*)

12.2 Transitive closure

lemma *trancl-mono*: !!*p*. *p* ∈ *r*^+ ==> *r* ⊆ *s* ==> *p* ∈ *s*^+
apply (*simp only*: *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 *trancl-into-rtrancl*: (*a*, *b*) ∈ *r*^+ ==> (*a*, *b*) ∈ *r*^*
by (*erule trancl.induct*) *iprover*+

lemma *rtrancl-into-trancl1*: **assumes** *r*: (*a*, *b*) ∈ *r*^*
shows !!*c*. (*b*, *c*) ∈ *r* ==> (*a*, *c*) ∈ *r*^+ **using** *r*

```
  by induct iprover+

lemma rtrancl-into-trancl2: [| (a,b) : r;  (b,c) : r^* |]  ==>  (a,c) : r^+
  — intro rule from r and rtrancl
  apply (erule rtranclE, iprover)
  apply (rule rtrancl-trans [THEN rtrancl-into-trancl1])
   apply (assumption | rule r-into-rtrancl)+
  done

lemma trancl-induct [consumes 1, induct set: trancl]:
  assumes a: (a,b) : r^+
  and cases: !!y. (a, y) : r ==> P y
    !!y z. (a,y) : r^+ ==> (y, z) : r ==> P y ==> P z
  shows P b
  — Nice induction rule for trancl
proof -
  from a have a = a --> P b
    by (induct %x y. x = a --> P y a b) (iprover intro: cases)+
  thus ?thesis by iprover
qed

lemma trancl-trans-induct:
  [| (x,y) : r^+;
      !!x y. (x,y) : r ==> P x y;
      !!x y z. [| (x,y) : r^+; P x y; (y,z) : r^+; P y z |] ==> P x z
   |] ==> P x y
  — Another induction rule for trancl, incorporating transitivity
proof -
  assume major: (x,y) : r^+
  case rule-context
  show ?thesis
    by (iprover intro: r-into-trancl major [THEN trancl-induct] prems)
qed

inductive-cases tranclE: (a, b) : r^+

lemma trancl-unfold: r^+ = r Un (r O r^+)
  by (auto intro: trancl-into-trancl elim: tranclE)

lemma trans-trancl: trans(r^+)
  — Transitivity of r^+
proof (rule transI)
  fix x y z
  assume (x, y) ∈ r^+
  assume (y, z) ∈ r^+
  thus (x, z) ∈ r^+ by induct (iprover!)+
qed

lemmas trancl-trans = trans-trancl [THEN transD, standard]
```

lemma *rtrancl-trancl-trancl*: **assumes** *r*: $(x, y) \in r\hat{}*$
shows $!!z.\ (y, z) \in r\hat{}+ \Longrightarrow (x, z) \in r\hat{}+$ **using** *r*
by *induct* (*iprover intro*: *trancl-trans*)+

lemma *trancl-into-trancl2*: $(a, b) \in r \Longrightarrow (b, c) \in r\hat{}+ \Longrightarrow (a, c) \in r\hat{}+$
by (*erule transD* [*OF trans-trancl r-into-trancl*])

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 *trancl-converseI*: $(x, y) \in (r\hat{}+)\hat{}-1 \Longrightarrow (x, y) \in (r\hat{}-1)\hat{}+$
apply (*drule converseD*)
apply (*erule trancl.induct*)
apply (*iprover intro*: *converseI trancl-trans*)+
done

lemma *trancl-converseD*: $(x, y) \in (r\hat{}-1)\hat{}+ \Longrightarrow (x, y) \in (r\hat{}+)\hat{}-1$
apply (*rule converseI*)
apply (*erule trancl.induct*)
apply (*iprover dest*: *converseD intro*: *trancl-trans*)+
done

lemma *trancl-converse*: $(r\hat{}-1)\hat{}+ = (r\hat{}+)\hat{}-1$
by (*fastsimp simp add*: *split-tupled-all*
intro!: *trancl-converseI trancl-converseD*)

lemma *converse-trancl-induct*:
$[|\ (a,b) : r\hat{}+;\ !!y.\ (y,b) : r \Longrightarrow P(y);$
$!!y\ z.[|\ (y,z) : r;\ \ (z,b) : r\hat{}+;\ \ P(z)\ |] \Longrightarrow P(y)\ |]$
$\Longrightarrow P(a)$
proof −
assume *major*: $(a,b) : r\hat{}+$
case *rule-context*
show *?thesis*
apply (*rule major* [*THEN converseI*, *THEN trancl-converseI* [*THEN trancl-induct*]])
apply (*rule prems*)
apply (*erule converseD*)
apply (*blast intro*: *prems dest*!: *trancl-converseD*)

done
qed

lemma *tranclD*: $(x, y) \in R\hat{}+ \Longrightarrow EX\ z.\ (x, z) \in R \wedge (z, y) \in R\hat{}*$
apply (*erule converse-trancl-induct*, *auto*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *irrefl-tranclI*: $r\hat{}-1 \cap r\hat{}* = \{\} \Longrightarrow (x, x) \notin r\hat{}+$
by(*blast elim*: *tranclE dest*: *trancl-into-rtrancl*)

lemma *irrefl-trancl-rD*: $!!X.\ ALL\ x.\ (x, x) \notin r\hat{}+ \Longrightarrow (x, y) \in r \Longrightarrow x \neq y$
by (*blast dest*: *r-into-trancl*)

lemma *trancl-subset-Sigma-aux*:
$(a, b) \in r\hat{}* \Longrightarrow r \subseteq A \times A \Longrightarrow a = b \vee a \in A$
apply (*erule rtrancl-induct*, *auto*)
done

lemma *trancl-subset-Sigma*: $r \subseteq A \times A \Longrightarrow r\hat{}+ \subseteq A \times A$
apply (*rule subsetI*)
apply (*simp only*: *split-tupled-all*)
apply (*erule tranclE*)
apply (*blast dest*!: *trancl-into-rtrancl trancl-subset-Sigma-aux*)+
done

lemma *reftcl-trancl* [*simp*]: $(r\hat{}+)\hat{}= = r\hat{}*$
apply *safe*
apply (*erule trancl-into-rtrancl*)
apply (*blast elim*: *rtranclE dest*: *rtrancl-into-trancl1*)
done

lemma *trancl-reftcl* [*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-reftcl* [*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 reftcl-trancl*]) *simp*

lemma *rtranclD*: $(a, b) \in R\hat{}* \Longrightarrow a = b \vee a \neq b \wedge (a, b) \in R\hat{}+$
by (*force simp add*: *reftcl-trancl* [*symmetric*] *simp del*: *reftcl-trancl*)

lemma *rtrancl-eq-or-trancl*:
(*x*,*y*) ∈ R^* = (*x*=*y* ∨ *x*≠*y* ∧ (*x*,*y*) ∈ R^+)
by (*fast elim*: *trancl-into-rtrancl dest*: *rtranclD*)

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: *Domain* (*R*^*) = *UNIV*
by *blast*

lemma *Range-rtrancl* [*simp*]: *Range* (*R*^*) = *UNIV*
by *blast*

lemma *rtrancl-Un-subset*: (*R*^* ∪ *S*^*) ⊆ (*R Un S*)^*
by (*rule rtrancl-Un-rtrancl* [*THEN subst*]) *fast*

lemma *in-rtrancl-UnI*: *x* ∈ *R*^* ∨ *x* ∈ *S*^* ==> *x* ∈ (*R* ∪ *S*)^*
by (*blast intro*: *subsetD* [*OF rtrancl-Un-subset*])

lemma *trancl-domain* [*simp*]: *Domain* (*r*^+) = *Domain r*
by (*unfold Domain-def*) (*blast dest*: *tranclD*)

lemma *trancl-range* [*simp*]: *Range* (*r*^+) = *Range r*
by (*simp add*: *Range-def trancl-converse* [*symmetric*])

lemma *Not-Domain-rtrancl*:
x ~: *Domain R* ==> ((*x*, *y*) : *R*^*) = (*x* = *y*)
apply *auto*
by (*erule rev-mp*, *erule rtrancl-induct*, *auto*)

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 *trancl-rtrancl-trancl*:
  $(a,\ b) \in r^+ ==> (b,\ c) \in r^* ==> (a,\ c) \in r^+$
 apply (*drule tranclD*)
 apply (*erule exE*, *erule conjE*)
 apply (*drule rtrancl-trans*, *assumption*)
 apply (*drule rtrancl-into-trancl2*, *assumption*, *assumption*)
 done

lemmas *transitive-closure-trans* [*trans*] =
 r-r-into-trancl trancl-trans rtrancl-trans
 trancl-into-trancl trancl-into-trancl2
 rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
 rtrancl-trancl-trancl trancl-rtrancl-trancl

declare *trancl-into-rtrancl* [*elim*]

declare *rtranclE* [*cases set*: *rtrancl*]
declare *tranclE* [*cases set*: *trancl*]

12.3 Setup of transitivity reasoner

use *../Provers/trancl.ML*

ML-setup ⟪

```
structure Trancl_Tac = Trancl_Tac_Fun (
  struct
    val r_into_trancl = thm "r_into_trancl";
    val trancl_trans  = thm "trancl_trans";
    val rtrancl_refl = thm "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 (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 rel;
        in SOME (a,b,rel,r) end
      | dec _ =  NONE
    in dec t end;

  end); (* struct *)
```

simpset-ref() := *simpset* ()
  addSolver (*mk-solver Trancl* (*fn* - => *Trancl-Tac.trancl-tac*))
  addSolver (*mk-solver Rtrancl* (*fn* - => *Trancl-Tac.rtrancl-tac*));

⟩⟩

end

13 Wellfounded-Recursion: Well-founded Recursion

theory *Wellfounded-Recursion*
imports *Transitive-Closure*
begin

consts
  wfrec-rel :: $('a * 'a)$ *set* => $(('a => 'b) => 'a => 'b) => ('a * 'b)$ *set*

inductive *wfrec-rel R F*
intros
  wfrecI: *ALL z.* $(z, x) : R$ −−> $(z, g\ z) :$ *wfrec-rel R F* ==>
    $(x, F\ g\ x) :$ *wfrec-rel R F*

constdefs
  wf :: $('a * 'a)$*set* => *bool*
  wf(r) == $(!P.\ (!x.\ (!y.\ (y,x):r$ −−> $P(y))$ −−> $P(x))$ −−> $(!x.\ P(x)))$

  acyclic :: $('a*'a)$*set* => *bool*
  acyclic r == $!x.\ (x,x) \sim: r\hat{}+$

  cut :: $('a => 'b) => ('a * 'a)$*set* => $'a => 'a => 'b$
  cut f r x == $(\%y.$ *if* $(y,x):r$ *then f y else arbitrary*$)$

  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* => $(('a => 'b) => 'a => 'b) => 'a => 'b$
  wfrec R F == $\%x.$ *THE y.* $(x, y) :$ *wfrec-rel R* $(\%f\ x.\ F\ (cut\ f\ R\ x)\ x)$

axclass *wellorder* ⊆ *linorder*
  wf: *wf* $\{(x,y::'a::ord).\ x<y\}$

lemma *wfUNIVI*:

(!!*P x*. (*ALL x*. (*ALL y*. (*y*,*x*) : *r* −−> *P*(*y*)) −−> *P*(*x*)) ==> *P*(*x*)) ==> *wf*(*r*)
by (*unfold wf-def*, *blast*)

Restriction to domain *A*. If *r* is well-founded over *A* then *wf r*

lemma *wfI*:
[| *r* <= *A* <*> *A*;
!!*x P*. [| *ALL x*. (*ALL y*. (*y*,*x*) : *r* −−> *P y*) −−> *P x*; *x*:*A* |] ==> *P x* |]
==> *wf r*
by (*unfold wf-def*, *blast*)

lemma *wf-induct*:
[| *wf*(*r*);
!!*x*.[| *ALL y*. (*y*,*x*): *r* −−> *P*(*y*) |] ==> *P*(*x*)
|] ==> *P*(*a*)
by (*unfold wf-def*, *blast*)

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *case-names less*, *induct set*: *wf*]

lemma *wf-not-sym* [*rule-format*]: *wf*(*r*) ==> *ALL x*. (*a*,*x*):*r* −−> (*x*,*a*)^~:*r*
by (*erule-tac a*=*a* **in** *wf-induct*, *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*]

transitive closure of a well-founded relation is well-founded!

lemma *wf-trancl*: *wf*(*r*) ==> *wf*(*r*^+)
apply (*subst wf-def*, *clarify*)
apply (*rule allE*, *assumption*)
— Retains the universal formula for later use!
apply (*erule mp*)
apply (*erule-tac a* = *x* **in** *wf-induct*)
apply (*blast elim*: *trancl*E)
done

lemma *wf-converse-trancl*: *wf* (*r*^−1) ==> *wf* ((*r*^+)^−1)
apply (*subst trancl-converse* [*symmetric*])
apply (*erule wf-trancl*)
done

13.0.1 Minimal-element characterization of well-foundedness

lemma *lemma1*: *wf r* ==> *x*:*Q* −−> (*EX z*:*Q*. *ALL y*. (*y*,*z*):*r* −−> *y*^~:*Q*)

apply (*unfold wf-def*)
apply (*drule spec*)
apply (*erule mp* [*THEN spec*], *blast*)
done

lemma *lemma2*: (*ALL Q x. x*:*Q* −−> (*EX z*:*Q. ALL y.* (*y*,*z*):*r* −−> *y*~:*Q*)) ==> *wf r*
apply (*unfold wf-def*, *clarify*)
apply (*drule-tac x* = {*x.* ~ *P x*} **in** *spec*, *blast*)
done

lemma *wf-eq-minimal*: *wf r* = (*ALL Q x. x*:*Q* −−> (*EX z*:*Q. ALL y.* (*y*,*z*):*r* −−> *y*~:*Q*))
by (*blast intro*!: *lemma1 lemma2*)

13.0.2 Other simple well-foundedness results

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

Well-foundedness of the empty relation

lemma *wf-empty* [*iff*]: *wf*({})
by (*simp add*: *wf-def*)

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

13.0.3 Well-Foundedness Results for Unions

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

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*)
apply (*simp only*: *wf-eq-minimal*, *clarify*)
apply (*rename-tac A a*)
apply (*case-tac EX a*:*A*. *EX b*:*A*. (*b*,*a*) : *r*)
prefer *2*
apply *simp*
apply (*drule-tac x*=*A* **in** *spec*)+
apply *blast*
apply (*erule-tac x*={*a*. *a*:*A* & (*EX b*:*A*. (*b*,*a*) : *r*) } **in** *allE*)+
apply (*blast elim*!: *allE*)
done

13.0.4 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

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*)

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

13.1 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.* (*x*, *y*) : *wfrec-rel R F*
apply (*simp add*: *adm-wf-def*)
apply (*erule-tac a*=*x* **in** *wf-induct*)
apply (*rule ex1I*)
apply (*rule-tac g* = %*x. THE y.* (*x*, *y*) : *wfrec-rel R F* **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

* 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

13.2 Code generator setup

consts-code
wfrec (⟨**module**⟩*wfrec?*)
attach ⟪
fun wfrec f x = *f* (*wfrec f*) *x*;
⟫

13.3 Variants for TFL: the Recdef Package

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

13.4 LEAST and wellorderings

See also *wf-linord-ex-has-least* and its consequences in *Wellfounded-Relations.ML*

lemma *wellorder-Least-lemma* [*rule-format*]:
    $P\ (k::'a::wellorder) \longrightarrow P\ (LEAST\ x.\ P(x))\ \&\ (LEAST\ x.\ P(x)) <= k$
apply (*rule-tac* $a = k$ **in** *wf* [*THEN wf-induct*])
apply (*rule impI*)
apply (*rule classical*)
apply (*rule-tac* $s = x$ **in** *Least-equality* [*THEN ssubst*], *auto*)
apply (*auto simp add*: *linorder-not-less* [*symmetric*])
done

lemmas *LeastI* = *wellorder-Least-lemma* [*THEN conjunct1*, *standard*]
lemmas *Least-le* = *wellorder-Least-lemma* [*THEN conjunct2*, *standard*]

— The following 3 lemmas are due to Brian Huffman
lemma *LeastI-ex*: $EX\ x::'a::wellorder.\ P\ x \Longrightarrow P\ (Least\ P)$
apply (*erule exE*)
apply (*erule LeastI*)
done

lemma *LeastI2*:
  $[|\ P\ (a::'a::wellorder);\ !!x.\ P\ x \Longrightarrow Q\ x\ |] \Longrightarrow Q\ (Least\ P)$
by (*blast intro*: *LeastI*)

lemma *LeastI2-ex*:
  $[|\ EX\ a::'a::wellorder.\ P\ a;\ !!x.\ P\ x \Longrightarrow Q\ x\ |] \Longrightarrow Q\ (Least\ P)$
by (*blast intro*: *LeastI-ex*)

lemma *not-less-Least*: $[|\ k < (LEAST\ x.\ P\ x)\ |] \Longrightarrow \sim P\ (k::'a::wellorder)$
apply (*simp* (*no-asm-use*) *add*: *linorder-not-le* [*symmetric*])
apply (*erule contrapos-nn*)
apply (*erule Least-le*)
done

ML
⟪
val wf-def = thm wf-def;
val wfUNIVI = thm wfUNIVI;
val wfI = thm wfI;
val wf-induct = thm wf-induct;
val wf-not-sym = thm wf-not-sym;
val wf-asym = thm wf-asym;
val wf-not-refl = thm wf-not-refl;
val wf-irrefl = thm wf-irrefl;

val wf-trancl = thm wf-trancl;
val wf-converse-trancl = thm wf-converse-trancl;
val wf-eq-minimal = thm wf-eq-minimal;
val wf-subset = thm wf-subset;
val wf-empty = thm wf-empty;
val wf-insert = thm wf-insert;
val wf-UN = thm wf-UN;
val wf-Union = thm wf-Union;
val wf-Un = thm wf-Un;
val wf-prod-fun-image = thm wf-prod-fun-image;
val acyclicI = thm acyclicI;
val wf-acyclic = thm wf-acyclic;
val acyclic-insert = thm acyclic-insert;
val acyclic-converse = thm acyclic-converse;
val acyclic-impl-antisym-rtrancl = thm acyclic-impl-antisym-rtrancl;
val acyclic-subset = thm acyclic-subset;
val cuts-eq = thm cuts-eq;
val cut-apply = thm cut-apply;
val wfrec-unique = thm wfrec-unique;
val wfrec = thm wfrec;
val def-wfrec = thm def-wfrec;
val tfl-wf-induct = thm tfl-wf-induct;
val tfl-cut-apply = thm tfl-cut-apply;
val tfl-wfrec = thm tfl-wfrec;
val LeastI = thm LeastI;
val Least-le = thm Least-le;
val not-less-Least = thm not-less-Least;
⟩

end

14 OrderedGroup: Ordered Groups

theory *OrderedGroup*
imports *Inductive LOrder*
uses *../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.

14.1 Semigroups, Groups

axclass *semigroup-add* $\subseteq$ *plus*
  add-assoc: $(a + b) + c = a + (b + c)$

axclass *ab-semigroup-add* $\subseteq$ *semigroup-add*
  add-commute: $a + b = b + a$

lemma *add-left-commute*: $a + (b + c) = b + (a + (c::'a::ab\text{-}semigroup\text{-}add))$
  by (*rule mk-left-commute* [*of op* +, *OF add-assoc add-commute*])

theorems *add-ac* = *add-assoc add-commute add-left-commute*

axclass *semigroup-mult* $\subseteq$ *times*
  mult-assoc: $(a * b) * c = a * (b * c)$

axclass *ab-semigroup-mult* $\subseteq$ *semigroup-mult*
  mult-commute: $a * b = b * a$

lemma *mult-left-commute*: $a * (b * c) = b * (a * (c::'a::ab\text{-}semigroup\text{-}mult))$
  by (*rule mk-left-commute* [*of op* *, *OF mult-assoc mult-commute*])

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

axclass *comm-monoid-add* $\subseteq$ *zero*, *ab-semigroup-add*
  add-0[*simp*]: $0 + a = a$

axclass *monoid-mult* $\subseteq$ *one*, *semigroup-mult*
  mult-1-left[*simp*]: $1 * a = a$
  mult-1-right[*simp*]: $a * 1 = a$

axclass *comm-monoid-mult* $\subseteq$ *one*, *ab-semigroup-mult*
  mult-1: $1 * a = a$

instance *comm-monoid-mult* $\subseteq$ *monoid-mult*
by (*intro-classes*, *insert mult-1*, *simp-all add*: *mult-commute*, *auto*)

axclass *cancel-semigroup-add* $\subseteq$ *semigroup-add*
  add-left-imp-eq: $a + b = a + c \Longrightarrow b = c$
  add-right-imp-eq: $b + a = c + a \Longrightarrow b = c$

axclass *cancel-ab-semigroup-add* $\subseteq$ *ab-semigroup-add*
  add-imp-eq: $a + b = a + c \Longrightarrow b = c$

instance *cancel-ab-semigroup-add* $\subseteq$ *cancel-semigroup-add*
proof
  {

```
    fix a b c :: 'a
    assume a + b = a + c
    thus b = c by (rule add-imp-eq)
  }
  note f = this
  fix a b c :: 'a
  assume b + a = c + a
  hence a + b = a + c by (simp only: add-commute)
  thus b = c by (rule f)
qed

axclass ab-group-add ⊆ minus, comm-monoid-add
  left-minus[simp]:  − a + a = 0
  diff-minus: a − b = a + (−b)

instance ab-group-add ⊆ cancel-ab-semigroup-add
proof
  fix a b c :: 'a
  assume a + b = a + c
  hence −a + a + b = −a + a + c by (simp only: add-assoc)
  thus b = c by simp
qed

lemma add-0-right [simp]: a + 0 = (a::'a::comm-monoid-add)
proof −
  have a + 0 = 0 + a by (simp only: add-commute)
  also have ... = a by simp
  finally show ?thesis .
qed

lemma add-left-cancel [simp]:
    (a + b = a + c) = (b = (c::'a::cancel-semigroup-add))
by (blast dest: add-left-imp-eq)

lemma add-right-cancel [simp]:
    (b + a = c + a) = (b = (c::'a::cancel-semigroup-add))
  by (blast dest: add-right-imp-eq)

lemma right-minus [simp]: a + −(a::'a::ab-group-add) = 0
proof −
  have a + −a = −a + a by (simp add: add-ac)
  also have ... = 0 by simp
  finally show ?thesis .
qed

lemma right-minus-eq: (a − b = 0) = (a = (b::'a::ab-group-add))
proof
  have a = a − b + b by (simp add: diff-minus add-ac)
  also assume a − b = 0
```

finally show $a = b$ **by** *simp*
next
assume $a = b$
thus $a - b = 0$ **by** (*simp add*: *diff-minus*)
qed

lemma *minus-minus* [*simp*]: $- (- (a::'a::ab\text{-}group\text{-}add)) = a$
proof (*rule add-left-cancel* [*of* $-a$, *THEN iffD1*])
show $(-a + -(-a) = -a + a)$
by *simp*
qed

lemma *equals-zero-I*: $a+b = 0 ==> -a = (b::'a::ab\text{-}group\text{-}add)$
apply (*rule right-minus-eq* [*THEN iffD1*, *symmetric*])
apply (*simp add*: *diff-minus add-commute*)
done

lemma *minus-zero* [*simp*]: $- 0 = (0::'a::ab\text{-}group\text{-}add)$
by (*simp add*: *equals-zero-I*)

lemma *diff-self* [*simp*]: $a - (a::'a::ab\text{-}group\text{-}add) = 0$
by (*simp add*: *diff-minus*)

lemma *diff-0* [*simp*]: $(0::'a::ab\text{-}group\text{-}add) - a = -a$
by (*simp add*: *diff-minus*)

lemma *diff-0-right* [*simp*]: $a - (0::'a::ab\text{-}group\text{-}add) = a$
by (*simp add*: *diff-minus*)

lemma *diff-minus-eq-add* [*simp*]: $a - - b = a + (b::'a::ab\text{-}group\text{-}add)$
by (*simp add*: *diff-minus*)

lemma *neg-equal-iff-equal* [*simp*]: $(-a = -b) = (a = (b::'a::ab\text{-}group\text{-}add))$
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*]: $(-a = 0) = (a = (0::'a::ab\text{-}group\text{-}add))$
by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

lemma *neg-0-equal-iff-equal* [*simp*]: $(0 = -a) = (0 = (a::'a::ab\text{-}group\text{-}add))$
by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*: $(a = - b) = (b = - (a::'a::ab\text{-}group\text{-}add))$
proof −
  have $(- (-a) = - b) = (- a = b)$ **by** (*rule neg-equal-iff-equal*)
  thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

lemma *minus-equation-iff*: $(- a = b) = (- (b::'a::ab\text{-}group\text{-}add) = a)$
proof −
  have $(- a = - (-b)) = (a = -b)$ **by** (*rule neg-equal-iff-equal*)
  thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

lemma *minus-add-distrib* [*simp*]: $- (a + b) = -a + -(b::'a::ab\text{-}group\text{-}add)$
apply (*rule equals-zero-I*)
apply (*simp add*: *add-ac*)
done

lemma *minus-diff-eq* [*simp*]: $- (a - b) = b - (a::'a::ab\text{-}group\text{-}add)$
by (*simp add*: *diff-minus add-commute*)

14.2 (Partially) Ordered Groups

axclass *pordered-ab-semigroup-add* ⊆ *order*, *ab-semigroup-add*
  add-left-mono: $a \leq b \Longrightarrow c + a \leq c + b$

axclass *pordered-cancel-ab-semigroup-add* ⊆ *pordered-ab-semigroup-add*, *cancel-ab-semigroup-add*

instance *pordered-cancel-ab-semigroup-add* ⊆ *pordered-ab-semigroup-add* **..**

axclass *pordered-ab-semigroup-add-imp-le* ⊆ *pordered-cancel-ab-semigroup-add*
  add-le-imp-le-left: $c + a \leq c + b \Longrightarrow a \leq b$

axclass *pordered-ab-group-add* ⊆ *ab-group-add*, *pordered-ab-semigroup-add*

instance *pordered-ab-group-add* ⊆ *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

axclass *ordered-cancel-ab-semigroup-add* ⊆ *pordered-cancel-ab-semigroup-add*, *linorder*

instance *ordered-cancel-ab-semigroup-add* ⊆ *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: $\sim a \leq b$
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 order-antisym*, *simp-all*)
done
with w **show** *False*
by (*simp add*: *linorder-not-le* [*symmetric*])
qed
qed

lemma *add-right-mono*: $a \leq (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add) ==> a + c \leq b + c$
by (*simp add*: *add-commute*[*of* - *c*] *add-left-mono*)

non-strict, in both arguments

lemma *add-mono*:
$[|a \leq b;\ c \leq d|] ==> a + c \leq b + (d::'a::pordered\text{-}ab\text{-}semigroup\text{-}add)$
apply (*erule add-right-mono* [*THEN order-trans*])
apply (*simp add*: *add-commute add-left-mono*)
done

lemma *add-strict-left-mono*:
$a < b ==> c + a < c + (b::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
by (*simp add*: *order-less-le add-left-mono*)

lemma *add-strict-right-mono*:
$a < b ==> a + c < b + (c::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
by (*simp add*: *add-commute* [*of* - *c*] *add-strict-left-mono*)

Strict monotonicity in both arguments

lemma *add-strict-mono*: $[|a<b; c<d|] ==> a + c < b + (d::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
apply (*erule add-strict-right-mono* [*THEN order-less-trans*])
apply (*erule add-strict-left-mono*)
done

lemma *add-less-le-mono*:
$[|\ a<b; c\leq d\ |] ==> a + c < b + (d::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
apply (*erule add-strict-right-mono* [*THEN order-less-le-trans*])
apply (*erule add-left-mono*)
done

lemma *add-le-less-mono*:
$[|\ a\leq b; c<d\ |] ==> a + c < b + (d::'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add)$
apply (*erule add-right-mono* [*THEN order-le-less-trans*])

apply (*erule add-strict-left-mono*)
done

lemma *add-less-imp-less-left*:
  assumes *less*: $c + a < c + b$ **shows** $a < (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le)$
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** *False***by** *simp*
 qed
 ultimately show $a < b$ **by** (*simp add*: *order-le-less*)
qed

lemma *add-less-imp-less-right*:
    $a + c < b + c ==> a < (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le)$
apply (*rule add-less-imp-less-left* [*of c*])
apply (*simp add*: *add-commute*)
done

lemma *add-less-cancel-left* [*simp*]:
  $(c+a < c+b) = (a < (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le))$
by (*blast intro*: *add-less-imp-less-left add-strict-left-mono*)

lemma *add-less-cancel-right* [*simp*]:
  $(a+c < b+c) = (a < (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le))$
by (*blast intro*: *add-less-imp-less-right add-strict-right-mono*)

lemma *add-le-cancel-left* [*simp*]:
  $(c+a \leq c+b) = (a \leq (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le))$
by (*auto*, *drule add-le-imp-le-left*, *simp-all add*: *add-left-mono*)

lemma *add-le-cancel-right* [*simp*]:
  $(a+c \leq b+c) = (a \leq (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le))$
by (*simp add*: *add-commute*[*of a c*] *add-commute*[*of b c*])

lemma *add-le-imp-le-right*:
    $a + c \leq b + c ==> a \leq (b::'a::pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le)$
by *simp*

lemma *add-increasing*:
 fixes $c :: 'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, comm\text{-}monoid\text{-}add\}$

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*)

14.3 Ordering Rules for Unary Minus

lemma *le-imp-neg-le*:
assumes $a \leq (b::'a::\{$*pordered-ab-semigroup-add-imp-le*, *ab-group-add*$\})$ **shows** $-b \leq -a$
proof −
have $-a+a \leq -a+b$
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*]: $(-b \leq -a) = (a \leq (b::'a::$*pordered-ab-group-add*$))$
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*]: $(-a \leq 0) = (0 \leq (a::'a::$*pordered-ab-group-add*$))$
by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-0-le-iff-le* [*simp*]: $(0 \leq -a) = (a \leq (0::'a::$*pordered-ab-group-add*$))$

by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-less-iff-less* [*simp*]: $(-b < -a) = (a < (b::'a::pordered\text{-}ab\text{-}group\text{-}add))$
by (*force simp add*: *order-less-le*)

lemma *neg-less-0-iff-less* [*simp*]: $(-a < 0) = (0 < (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
by (*subst neg-less-iff-less* [*symmetric*], *simp*)

lemma *neg-0-less-iff-less* [*simp*]: $(0 < -a) = (a < (0::'a::pordered\text{-}ab\text{-}group\text{-}add))$
by (*subst neg-less-iff-less* [*symmetric*], *simp*)

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $(a < - b) = (b < - (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
proof −
  have $(- (-a) < - b) = (b < - a)$ **by** (*rule neg-less-iff-less*)
  thus *?thesis* **by** *simp*
qed

lemma *minus-less-iff*: $(- a < b) = (- b < (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
proof −
  have $(- a < - (-b)) = (- b < a)$ **by** (*rule neg-less-iff-less*)
  thus *?thesis* **by** *simp*
qed

lemma *le-minus-iff*: $(a \leq - b) = (b \leq - (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
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*: *order-le-less*)
    apply (*drule mm*)
    apply (*simp-all*)
    apply (*drule mm*[*simplified*], *assumption*)
    done
  then show *?thesis* **by** *simp*
qed

lemma *minus-le-iff*: $(- a \leq b) = (- b \leq (a::'a::pordered\text{-}ab\text{-}group\text{-}add))$
by (*auto simp add*: *order-le-less minus-less-iff*)

lemma *add-diff-eq*: $a + (b - c) = (a + b) - (c::'a::ab\text{-}group\text{-}add)$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-add-eq*: $(a - b) + c = (a + c) - (b::'a::ab\text{-}group\text{-}add)$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-eq-eq*: $(a-b = c) = (a = c + (b::'a::ab\text{-}group\text{-}add))$
by (*auto simp add*: *diff-minus add-assoc*)

lemma *eq-diff-eq*: $(a = c-b) = (a + (b::'a::ab\text{-}group\text{-}add) = c)$

by (*auto simp add*: *diff-minus add-assoc*)

lemma *diff-diff-eq*: $(a - b) - c = a - (b + (c::'a::ab\text{-}group\text{-}add))$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-diff-eq2*: $a - (b - c) = (a + c) - (b::'a::ab\text{-}group\text{-}add)$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-add-cancel*: $a - b + b = (a::'a::ab\text{-}group\text{-}add)$
by (*simp add*: *diff-minus add-ac*)

lemma *add-diff-cancel*: $a + b - b = (a::'a::ab\text{-}group\text{-}add)$
by (*simp add*: *diff-minus add-ac*)

Further subtraction laws

lemma *less-iff-diff-less-0*: $(a < b) = (a - b < (0::'a::pordered\text{-}ab\text{-}group\text{-}add))$
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*: $(a-b < c) = (a < c + (b::'a::pordered\text{-}ab\text{-}group\text{-}add))$
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*: $(a < c-b) = (a + (b::'a::pordered\text{-}ab\text{-}group\text{-}add) < c)$
apply (*subst less-iff-diff-less-0* [*of a+b*])
apply (*subst less-iff-diff-less-0* [*of a*])
apply (*simp add*: *diff-minus add-ac*)
done

lemma *diff-le-eq*: $(a-b \leq c) = (a \leq c + (b::'a::pordered\text{-}ab\text{-}group\text{-}add))$
by (*auto simp add*: *order-le-less diff-less-eq diff-add-cancel add-diff-cancel*)

lemma *le-diff-eq*: $(a \leq c-b) = (a + (b::'a::pordered\text{-}ab\text{-}group\text{-}add) \leq c)$
by (*auto simp add*: *order-le-less less-diff-eq diff-add-cancel add-diff-cancel*)

This list of rewrites simplifies (in)equalities by bringing subtractions to the top and then moving negative terms to the other side. Use with *add-ac*

lemmas *compare-rls* =
    diff-minus [*symmetric*]
    add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
    diff-less-eq less-diff-eq diff-le-eq le-diff-eq
    diff-eq-eq eq-diff-eq

14.4 Support for reasoning about signs

lemma *add-pos-pos*: *0 <*
(x::'a::{comm-monoid-add,pordered-cancel-ab-semigroup-add})
==> 0 < y ==> 0 < x + y
apply (*subgoal-tac 0 + 0 < x + y*)
apply *simp*
apply (*erule add-less-le-mono*)
apply (*erule order-less-imp-le*)
done

lemma *add-pos-nonneg*: *0 <*
(x::'a::{comm-monoid-add,pordered-cancel-ab-semigroup-add})
==> 0 <= y ==> 0 < x + y
apply (*subgoal-tac 0 + 0 < x + y*)
apply *simp*
apply (*erule add-less-le-mono*, *assumption*)
done

lemma *add-nonneg-pos*: *0 <=*
(x::'a::{comm-monoid-add,pordered-cancel-ab-semigroup-add})
==> 0 < y ==> 0 < x + y
apply (*subgoal-tac 0 + 0 < x + y*)
apply *simp*
apply (*erule add-le-less-mono*, *assumption*)
done

lemma *add-nonneg-nonneg*: *0 <=*
(x::'a::{comm-monoid-add,pordered-cancel-ab-semigroup-add})
==> 0 <= y ==> 0 <= x + y
apply (*subgoal-tac 0 + 0 <= x + y*)
apply *simp*
apply (*erule add-mono*, *assumption*)
done

lemma *add-neg-neg*: *(x::'a::{comm-monoid-add,pordered-cancel-ab-semigroup-add})*
< 0 ==> y < 0 ==> x + y < 0
apply (*subgoal-tac x + y < 0 + 0*)
apply *simp*
apply (*erule add-less-le-mono*)
apply (*erule order-less-imp-le*)
done

lemma *add-neg-nonpos*:
(x::'a::{comm-monoid-add,pordered-cancel-ab-semigroup-add}) < 0
==> y <= 0 ==> x + y < 0
apply (*subgoal-tac x + y < 0 + 0*)
apply *simp*
apply (*erule add-less-le-mono*, *assumption*)
done

lemma *add-nonpos-neg*:
(*x*::'*a*::{*comm-monoid-add*,*pordered-cancel-ab-semigroup-add*}) <= *0*
==> *y* < *0* ==> *x* + *y* < *0*
apply (*subgoal-tac x* + *y* < *0* + *0*)
apply *simp*
apply (*erule add-le-less-mono*, *assumption*)
done

lemma *add-nonpos-nonpos*:
(*x*::'*a*::{*comm-monoid-add*,*pordered-cancel-ab-semigroup-add*}) <= *0*
==> *y* <= *0* ==> *x* + *y* <= *0*
apply (*subgoal-tac x* + *y* <= *0* + *0*)
apply *simp*
apply (*erule add-mono*, *assumption*)
done

14.5 Lemmas for the *cancel-numerals* simproc

lemma *eq-iff-diff-eq-0*: (*a* = *b*) = (*a*−*b* = (*0*::'*a*::*ab-group-add*))
by (*simp add*: *compare-rls*)

lemma *le-iff-diff-le-0*: (*a* ≤ *b*) = (*a*−*b* ≤ (*0*::'*a*::*pordered-ab-group-add*))
by (*simp add*: *compare-rls*)

14.6 Lattice Ordered (Abelian) Groups

axclass *lordered-ab-group-meet* < *pordered-ab-group-add*, *meet-semilorder*

axclass *lordered-ab-group-join* < *pordered-ab-group-add*, *join-semilorder*

lemma *add-meet-distrib-left*: *a* + (*meet b c*) = *meet* (*a* + *b*) (*a* + (*c*::'*a*::{*pordered-ab-group-add*, *meet-semilorder*}))
apply (*rule order-antisym*)
apply (*rule meet-imp-le*, *simp-all add*: *meet-join-le*)
apply (*rule add-le-imp-le-left* [*of* −*a*])
apply (*simp only*: *add-assoc*[*symmetric*], *simp*)
apply (*rule meet-imp-le*)
apply (*rule add-le-imp-le-left*[*of a*], *simp only*: *add-assoc*[*symmetric*], *simp add*: *meet-join-le*)+
done

lemma *add-join-distrib-left*: *a* + (*join b c*) = *join* (*a* + *b*) (*a*+ (*c*::'*a*::{*pordered-ab-group-add*, *join-semilorder*}))
apply (*rule order-antisym*)
apply (*rule add-le-imp-le-left* [*of* −*a*])
apply (*simp only*: *add-assoc*[*symmetric*], *simp*)
apply (*rule join-imp-le*)
apply (*rule add-le-imp-le-left* [*of a*], *simp only*: *add-assoc*[*symmetric*], *simp add*: *meet-join-le*)+

apply (*rule join-imp-le*)
apply (*simp-all add*: *meet-join-le*)
done

lemma *is-join-neg-meet*: *is-join* (% (*a*::'*a*::{*pordered-ab-group-add*, *meet-semilorder*})
b. − (*meet* (−*a*) (−*b*)))
apply (*auto simp add*: *is-join-def*)
apply (*rule-tac c*=*meet* (−*a*) (−*b*) **in** *add-le-imp-le-right*, *simp*, *simp add*: *add-meet-distrib-left meet-join-le*)
apply (*rule-tac c*=*meet* (−*a*) (−*b*) **in** *add-le-imp-le-right*, *simp*, *simp add*: *add-meet-distrib-left meet-join-le*)
apply (*subst neg-le-iff-le*[*symmetric*])
apply (*simp add*: *meet-imp-le*)
done

lemma *is-meet-neg-join*: *is-meet* (% (*a*::'*a*::{*pordered-ab-group-add*, *join-semilorder*})
b. − (*join* (−*a*) (−*b*)))
apply (*auto simp add*: *is-meet-def*)
apply (*rule-tac c*=*join* (−*a*) (−*b*) **in** *add-le-imp-le-right*, *simp*, *simp add*: *add-join-distrib-left meet-join-le*)
apply (*rule-tac c*=*join* (−*a*) (−*b*) **in** *add-le-imp-le-right*, *simp*, *simp add*: *add-join-distrib-left meet-join-le*)
apply (*subst neg-le-iff-le*[*symmetric*])
apply (*simp add*: *join-imp-le*)
done

axclass *lordered-ab-group* ⊆ *pordered-ab-group-add*, *lorder*

instance *lordered-ab-group-meet* ⊆ *lordered-ab-group*
proof
show *?j*. *is-join* (*j*::'*a*⇒'*a*⇒('*a*::*lordered-ab-group-meet*)) **by** (*blast intro*: *is-join-neg-meet*)
qed

instance *lordered-ab-group-join* ⊆ *lordered-ab-group*
proof
show *?m*. *is-meet* (*m*::'*a*⇒'*a*⇒('*a*::*lordered-ab-group-join*)) **by** (*blast intro*: *is-meet-neg-join*)
qed

lemma *add-join-distrib-right*: (*join a b*) + (*c*::'*a*::*lordered-ab-group*) = *join* (*a*+*c*) (*b*+*c*)
proof −
have *c* + (*join a b*) = *join* (*c*+*a*) (*c*+*b*) **by** (*simp add*: *add-join-distrib-left*)
thus *?thesis* **by** (*simp add*: *add-commute*)
qed

lemma *add-meet-distrib-right*: (*meet a b*) + (*c*::'*a*::*lordered-ab-group*) = *meet* (*a*+*c*) (*b*+*c*)
proof −

have $c + (meet\ a\ b) = meet\ (c+a)\ (c+b)$ **by** (*simp add*: *add-meet-distrib-left*)
thus *?thesis* **by** (*simp add*: *add-commute*)
qed

lemmas *add-meet-join-distribs* = *add-meet-distrib-right add-meet-distrib-left add-join-distrib-right add-join-distrib-left*

lemma *join-eq-neg-meet*: $join\ a\ (b::'a::lordered\text{-}ab\text{-}group) = -\ meet\ (-a)\ (-b)$
by (*simp add*: *is-join-unique*[*OF is-join-join is-join-neg-meet*])

lemma *meet-eq-neg-join*: $meet\ a\ (b::'a::lordered\text{-}ab\text{-}group) = -\ join\ (-a)\ (-b)$
by (*simp add*: *is-meet-unique*[*OF is-meet-meet is-meet-neg-join*])

lemma *add-eq-meet-join*: $a + b = (join\ a\ b) + (meet\ a\ (b::'a::lordered\text{-}ab\text{-}group))$
proof −
have $0 = -\ meet\ 0\ (a-b) + meet\ (a-b)\ 0$ **by** (*simp add*: *meet-comm*)
hence $0 = join\ 0\ (b-a) + meet\ (a-b)\ 0$ **by** (*simp add*: *meet-eq-neg-join*)
hence $0 = (-a + join\ a\ b) + (meet\ a\ b + (-b))$
apply (*simp add*: *add-join-distrib-left add-meet-distrib-right*)
by (*simp add*: *diff-minus add-commute*)
thus *?thesis*
apply (*simp add*: *compare-rls*)
apply (*subst add-left-cancel*[*symmetric*, *of* $a+b\ join\ a\ b + meet\ a\ b\ -a$])
apply (*simp only*: *add-assoc*, *simp add*: *add-assoc*[*symmetric*])
done
qed

14.7 Positive Part, Negative Part, Absolute Value

constdefs
$pprt :: 'a \Rightarrow ('a::lordered\text{-}ab\text{-}group)$
$pprt\ x == join\ x\ 0$
$nprt :: 'a \Rightarrow ('a::lordered\text{-}ab\text{-}group)$
$nprt\ x == meet\ x\ 0$

lemma *prts*: $a = pprt\ a + nprt\ a$
by (*simp add*: *pprt-def nprt-def add-eq-meet-join*[*symmetric*])

lemma *zero-le-pprt*[*simp*]: $0 \leq pprt\ a$
by (*simp add*: *pprt-def meet-join-le*)

lemma *nprt-le-zero*[*simp*]: $nprt\ a \leq 0$
by (*simp add*: *nprt-def meet-join-le*)

lemma *le-eq-neg*: $(a \leq -b) = (a + b \leq (0::-::lordered\text{-}ab\text{-}group))$ (**is** *?l* = *?r*)
proof −
have *a*: *?l* ⟶ *?r*
apply (*auto*)
apply (*rule add-le-imp-le-right*[*of* - $-b$ -])

```
    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]: 0 <= x ⟹ pprt x = x
  by (simp add: pprt-def le-def-join join-aci)

lemma nprt-eq-id[simp]: x <= 0 ⟹ nprt x = x
  by (simp add: nprt-def le-def-meet meet-aci)

lemma pprt-eq-0[simp]: x <= 0 ⟹ pprt x = 0
  by (simp add: pprt-def le-def-join join-aci)

lemma nprt-eq-0[simp]: 0 <= x ⟹ nprt x = 0
  by (simp add: nprt-def le-def-meet meet-aci)

lemma join-0-imp-0: join a (-a) = 0 ⟹ a = (0::'a::lordered-ab-group)
proof -
  {
    fix a::'a
    assume hyp: join a (-a) = 0
    hence join a (-a) + a = a by (simp)
    hence join (a+a) 0 = a by (simp add: add-join-distrib-right)
    hence join (a+a) 0 <= a by (simp)
    hence 0 <= a by (blast intro: order-trans meet-join-le)
  }
  note p = this
  assume hyp:join a (-a) = 0
  hence hyp2:join (-a) (-(-a)) = 0 by (simp add: join-comm)
  from p[OF hyp] p[OF hyp2] show a = 0 by simp
qed

lemma meet-0-imp-0: meet a (-a) = 0 ⟹ a = (0::'a::lordered-ab-group)
apply (simp add: meet-eq-neg-join)
apply (simp add: join-comm)
apply (erule join-0-imp-0)
done

lemma join-0-eq-0[simp]: (join a (-a) = 0) = (a = (0::'a::lordered-ab-group))
by (auto, erule join-0-imp-0)
```

lemma *meet-0-eq-0*[*simp*]: (*meet a* (−*a*) = *0*) = (*a* = (*0*::'*a*::*lordered-ab-group*))
by (*auto*, *erule meet-0-imp-0*)

lemma *zero-le-double-add-iff-zero-le-single-add*[*simp*]: (*0* ≤ *a* + *a*) = (*0* ≤ (*a*::'*a*::*lordered-ab-group*))
proof
assume *0* <= *a* + *a*
hence *a*:*meet* (*a*+*a*) *0* = *0* **by** (*simp add*: *le-def-meet meet-comm*)
have (*meet a 0*)+(*meet a 0*) = *meet* (*meet* (*a*+*a*) *0*) *a* (**is** *?l*=-) **by** (*simp add*: *add-meet-join-distribs meet-aci*)
hence *?l* = *0* + *meet a 0* **by** (*simp add*: *a*, *simp add*: *meet-comm*)
hence *meet a 0* = *0* **by** (*simp only*: *add-right-cancel*)
then show *0* <= *a* **by** (*simp add*: *le-def-meet meet-comm*)
next
assume *a*: *0* <= *a*
show *0* <= *a* + *a* **by** (*simp add*: *add-mono*[*OF a a*, *simplified*])
qed

lemma *double-add-le-zero-iff-single-add-le-zero*[*simp*]: (*a* + *a* <= *0*) = ((*a*::'*a*::*lordered-ab-group*) <= *0*)
proof −
have (*a* + *a* <= *0*) = (*0* <= −(*a*+*a*)) **by** (*subst le-minus-iff*, *simp*)
moreover have ... = (*a* <= *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*) = ((*a*::'*a*::{*pordered-ab-group-add*,*linorder*}) < *0*) (**is** *?s*)
proof *cases*
assume *a*: *a* < *0*
thus *?s* **by** (*simp add*: *add-strict-mono*[*OF a a*, *simplified*])
next
assume ~(*a* < *0*)
hence *a*:*0* <= *a* **by** (*simp*)
hence *0* <= *a*+*a* **by** (*simp add*: *add-mono*[*OF a a*, *simplified*])
hence ~(*a*+*a* < *0*) **by** *simp*
with *a* **show** *?thesis* **by** *simp*
qed

axclass *lordered-ab-group-abs* ⊆ *lordered-ab-group*
abs-lattice: *abs x* = *join x* (−*x*)

lemma *abs-zero*[*simp*]: *abs 0* = (*0*::'*a*::*lordered-ab-group-abs*)
by (*simp add*: *abs-lattice*)

lemma *abs-eq-0*[*simp*]: (*abs a* = *0*) = (*a* = (*0*::'*a*::*lordered-ab-group-abs*))
by (*simp add*: *abs-lattice*)

lemma *abs-0-eq*[*simp*]: (*0* = *abs a*) = (*a* = (*0*::'*a*::*lordered-ab-group-abs*))

proof −
have (*0* = *abs a*) = (*abs a* = *0*) **by** (*simp only*: *eq-ac*)
thus *?thesis* **by** *simp*
qed

lemma *neg-meet-eq-join*[*simp*]: − *meet a* (*b*::-::*lordered-ab-group*) = *join* (−*a*) (−*b*)
by (*simp add*: *meet-eq-neg-join*)

lemma *neg-join-eq-meet*[*simp*]: − *join a* (*b*::-::*lordered-ab-group*) = *meet* (−*a*) (−*b*)
by (*simp del*: *neg-meet-eq-join add*: *join-eq-neg-meet*)

lemma *join-eq-if*: *join a* (−*a*) = (*if a* < *0 then* −*a else* (*a*::'*a*::{*lordered-ab-group*, *linorder*}))
proof −
note *b* = *add-le-cancel-right*[*of a a* −*a*,*symmetric*,*simplified*]
have *c*: *a* + *a* = *0* ⟹ −*a* = *a* **by** (*rule add-right-imp-eq*[*of* - *a*], *simp*)
show *?thesis* **by** (*auto simp add*: *join-max max-def b linorder-not-less*)
qed

lemma *abs-if-lattice*: |*a*| = (*if a* < *0 then* −*a else* (*a*::'*a*::{*lordered-ab-group-abs*, *linorder*}))
proof −
show *?thesis* **by** (*simp add*: *abs-lattice join-eq-if*)
qed

lemma *abs-ge-zero*[*simp*]: *0* ≤ *abs* (*a*::'*a*::*lordered-ab-group-abs*)
proof −
have *a*:*a* <= *abs a* **and** *b*:−*a* <= *abs a* **by** (*auto simp add*: *abs-lattice meet-join-le*)
show *?thesis* **by** (*rule add-mono*[*OF a b*, *simplified*])
qed

lemma *abs-le-zero-iff* [*simp*]: (*abs a* ≤ (*0*::'*a*::*lordered-ab-group-abs*)) = (*a* = *0*)
proof
assume *abs a* <= *0*
hence *abs a* = *0* **by** (*auto dest*: *order-antisym*)
thus *a* = *0* **by** *simp*
next
assume *a* = *0*
thus *abs a* <= *0* **by** *simp*
qed

lemma *zero-less-abs-iff* [*simp*]: (*0* < *abs a*) = (*a* ≠ (*0*::'*a*::*lordered-ab-group-abs*))
by (*simp add*: *order-less-le*)

lemma *abs-not-less-zero* [*simp*]: ~ *abs a* < (*0*::'*a*::*lordered-ab-group-abs*)
proof −
have *a*:!! *x* (*y*::-::*order*). *x* <= *y* ⟹ ~(*y* < *x*) **by** *auto*

show *?thesis* **by** (*simp add*: *a*)
qed

lemma *abs-ge-self*: $a \leq abs\ (a::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
by (*simp add*: *abs-lattice meet-join-le*)

lemma *abs-ge-minus-self*: $-a \leq abs\ (a::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
by (*simp add*: *abs-lattice meet-join-le*)

lemma *le-imp-join-eq*: $a \leq b \Longrightarrow join\ a\ b = b$
by (*simp add*: *le-def-join*)

lemma *ge-imp-join-eq*: $b \leq a \Longrightarrow join\ a\ b = a$
by (*simp add*: *le-def-join join-aci*)

lemma *le-imp-meet-eq*: $a \leq b \Longrightarrow meet\ a\ b = a$
by (*simp add*: *le-def-meet*)

lemma *ge-imp-meet-eq*: $b \leq a \Longrightarrow meet\ a\ b = b$
by (*simp add*: *le-def-meet meet-aci*)

lemma *abs-prts*: $abs\ (a::-::lordered\text{-}ab\text{-}group\text{-}abs) = pprt\ a - nprt\ a$
apply (*simp add*: *pprt-def nprt-def diff-minus*)
apply (*simp add*: *add-meet-join-distribs join-aci abs-lattice*[*symmetric*])
apply (*subst le-imp-join-eq*, *auto*)
done

lemma *abs-minus-cancel* [*simp*]: $abs\ (-a) = abs(a::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
by (*simp add*: *abs-lattice join-comm*)

lemma *abs-idempotent* [*simp*]: $abs\ (abs\ a) = abs\ (a::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
apply (*simp add*: *abs-lattice*[*of abs a*])
apply (*subst ge-imp-join-eq*)
apply (*rule order-trans*[*of - 0*])
by *auto*

lemma *abs-minus-commute*:
fixes $a :: 'a::lordered\text{-}ab\text{-}group\text{-}abs$
shows $abs\ (a-b) = abs(b-a)$
proof −
have $abs\ (a-b) = abs\ (-\ (a-b))$ **by** (*simp only*: *abs-minus-cancel*)
also have $... = abs(b-a)$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *zero-le-iff-zero-nprt*: $(0 \leq a) = (nprt\ a = 0)$
by (*simp add*: *le-def-meet nprt-def meet-comm*)

lemma *le-zero-iff-zero-pprt*: $(a \leq 0) = (pprt\ a = 0)$

by (*simp add*: *le-def-join pprt-def join-comm*)

lemma *le-zero-iff-pprt-id*: $(0 \leq a) = (pprt\ a = a)$
by (*simp add*: *le-def-join pprt-def join-comm*)

lemma *zero-le-iff-nprt-id*: $(a \leq 0) = (nprt\ a = a)$
by (*simp add*: *le-def-meet nprt-def meet-comm*)

lemma *pprt-mono*[*simp*]: (*a*::-::*lordered-ab-group*) <= *b* ⟹ *pprt a* <= *pprt b*
by (*simp add*: *le-def-join pprt-def join-aci*)

lemma *nprt-mono*[*simp*]: (*a*::-::*lordered-ab-group*) <= *b* ⟹ *nprt a* <= *nprt b*
by (*simp add*: *le-def-meet nprt-def meet-aci*)

lemma *iff2imp*: $(A=B) \Longrightarrow (A \Longrightarrow B)$
by (*simp*)

lemma *abs-of-nonneg* [*simp*]: $0 \leq a \Longrightarrow abs\ a = (a::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
by (*simp add*: *iff2imp*[*OF zero-le-iff-zero-nprt*] *iff2imp*[*OF le-zero-iff-pprt-id*] *abs-prts*)

lemma *abs-of-pos*: $0 < (x::'a::lordered\text{-}ab\text{-}group\text{-}abs) \Longrightarrow abs\ x = x$
by (*rule abs-of-nonneg*, *rule order-less-imp-le*)

lemma *abs-of-nonpos* [*simp*]: $a \leq 0 \Longrightarrow abs\ a = -(a::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
by (*simp add*: *iff2imp*[*OF le-zero-iff-zero-pprt*] *iff2imp*[*OF zero-le-iff-nprt-id*] *abs-prts*)

lemma *abs-of-neg*: $(x::'a::lordered\text{-}ab\text{-}group\text{-}abs) < 0 \Longrightarrow$
abs x = − *x*
by (*rule abs-of-nonpos*, *rule order-less-imp-le*)

lemma *abs-leI*: $[|a \leq b; -a \leq b|] \Longrightarrow abs\ a \leq (b::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
by (*simp add*: *abs-lattice join-imp-le*)

lemma *le-minus-self-iff*: $(a \leq -a) = (a \leq (0::'a::lordered\text{-}ab\text{-}group))$
proof −
from *add-le-cancel-left*[*of* −*a a*+*a* 0] **have** (*a* <= −*a*) = (*a*+*a* <= 0)
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *minus-le-self-iff*: $(-a \leq a) = (0 \leq (a::'a::lordered\text{-}ab\text{-}group))$
proof −
from *add-le-cancel-left*[*of* −*a* 0 *a*+*a*] **have** (−*a* <= *a*) = (0 <= *a*+*a*)
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *abs-le-D1*: $abs\ a \leq b \Longrightarrow a \leq (b::'a::lordered\text{-}ab\text{-}group\text{-}abs)$
by (*insert abs-ge-self*, *blast intro*: *order-trans*)

lemma *abs-le-D2*: *abs a* ≤ *b* ==> −*a* ≤ (*b*::'*a*::*lordered-ab-group-abs*)
by (*insert abs-le-D1* [*of* −*a*], *simp*)

lemma *abs-le-iff*: (*abs a* ≤ *b*) = (*a* ≤ *b* & −*a* ≤ (*b*::'*a*::*lordered-ab-group-abs*))
by (*blast intro*: *abs-leI dest*: *abs-le-D1 abs-le-D2*)

lemma *abs-triangle-ineq*: *abs*(*a*+*b*) ≤ *abs a* + *abs*(*b*::'*a*::*lordered-ab-group-abs*)
proof −
  have *g*:*abs a* + *abs b* = *join* (*a*+*b*) (*join* (−*a*−*b*) (*join* (−*a*+*b*) (*a* + (−*b*))))
(**is** -=*join ?m ?n*)
    apply (*simp add*: *abs-lattice add-meet-join-distribs join-aci*)
    by (*simp only*: *diff-minus*)
  have *a*:*a*+*b* <= *join ?m ?n* **by** (*simp add*: *meet-join-le*)
  have *b*:−*a*−*b* <= *?n* **by** (*simp add*: *meet-join-le*)
  have *c*:*?n* <= *join ?m ?n* **by** (*simp add*: *meet-join-le*)
  from *b c* **have** *d*: −*a*−*b* <= *join ?m ?n* **by** *simp*
  have *e*:−*a*−*b* = −(*a*+*b*) **by** (*simp add*: *diff-minus*)
  from *a d e* **have** *abs*(*a*+*b*) <= *join ?m ?n*
    by (*drule-tac abs-leI*, *auto*)
  with *g*[*symmetric*] **show** *?thesis* **by** *simp*
qed

lemma *abs-triangle-ineq2*: *abs* (*a*::'*a*::*lordered-ab-group-abs*) −
    abs b <= *abs* (*a* − *b*)
  apply (*simp add*: *compare-rls*)
  apply (*subgoal-tac abs a* = *abs* (*a* − *b* + *b*))
  apply (*erule ssubst*)
  apply (*rule abs-triangle-ineq*)
  apply (*rule arg-cong*)**back**
  apply (*simp add*: *compare-rls*)
done

lemma *abs-triangle-ineq3*:
    abs(*abs* (*a*::'*a*::*lordered-ab-group-abs*) − *abs b*) <= *abs* (*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*: *abs* ((*a*::'*a*::*lordered-ab-group-abs*) − *b*) <=
    abs a + *abs 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*:
$|(a::'a::lordered\text{-}ab\text{-}group\text{-}abs) + b - (c+d)| \leq |a-c| + |b-d|$
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*]:
fixes $a::'a::\{lordered\text{-}ab\text{-}group\text{-}abs\}$
shows $abs(abs\ a + abs\ b) = abs\ a + abs\ b$ (**is** *?L* = *?R*)
proof (*rule order-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

Needed for abelian cancellation simprocs:

lemma *add-cancel-21*: $((x::'a::ab\text{-}group\text{-}add) + (y + z) = y + u) = (x + z = u)$
apply (*subst add-left-commute*)
apply (*subst add-left-cancel*)
apply *simp*
done

lemma *add-cancel-end*: $(x + (y + z) = y) = (x = -\ (z::'a::ab\text{-}group\text{-}add))$
apply (*subst add-cancel-21*[*of - - - 0*, *simplified*])
apply (*simp add*: *add-right-cancel*[*symmetric*, *of x −z z*, *simplified*])
done

lemma *less-eqI*: $(x::'a::pordered\text{-}ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (x < y) = (x' < y')$
by (*simp add*: *less-iff-diff-less-0*[*of x y*] *less-iff-diff-less-0*[*of x' y'*])

lemma *le-eqI*: $(x::'a::pordered\text{-}ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (y <= x) = (y' <= x')$
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*: $(x::'a::ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (x = y) = (x' = y')$
by (*simp add*: *eq-iff-diff-eq-0*[*of x y*] *eq-iff-diff-eq-0*[*of x' y'*])

lemma *diff-def*: $(x::'a::ab\text{-}group\text{-}add) - y == x + (-y)$
by (*simp add*: *diff-minus*)

lemma *add-minus-cancel*: $(a::'a::ab\text{-}group\text{-}add) + (-a + b) = b$
by (*simp add*: *add-assoc*[*symmetric*])

lemma *minus-add-cancel*: $-(a::'a::ab\text{-}group\text{-}add) + (a + b) = b$
by (*simp add*: *add-assoc*[*symmetric*])

lemma *le-add-right-mono*:
assumes
$a <= b + (c::'a::pordered\text{-}ab\text{-}group\text{-}add)$
$c <= d$
shows $a <= b + d$
apply (*rule-tac order-trans*[**where** $y = b+c$])
apply (*simp-all add*: *prems*)
done

lemmas *group-eq-simps* =
mult-ac
add-ac
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-eq-eq eq-diff-eq

lemma *estimate-by-abs*:
$a + b <= (c::'a::lordered\text{-}ab\text{-}group\text{-}abs) \Longrightarrow a <= c + abs\ b$
proof −
assume *1*: $a+b <= c$
have *2*: $a <= c+(-b)$
apply (*insert 1*)
apply (*drule-tac add-right-mono*[**where** $c=-b$])
apply (*simp add*: *group-eq-simps*)
done
have *3*: $(-b) <= abs\ b$ **by** (*rule abs-ge-minus-self*)
show *?thesis* **by** (*rule le-add-right-mono*[*OF 2 3*])
qed

Simplification of $x - y < (0::'a)$, etc.

lemmas *diff-less-0-iff-less* = *less-iff-diff-less-0* [*symmetric*]
lemmas *diff-eq-0-iff-eq* = *eq-iff-diff-eq-0* [*symmetric*]
lemmas *diff-le-0-iff-le* = *le-iff-diff-le-0* [*symmetric*]
declare *diff-less-0-iff-less* [*simp*]
declare *diff-eq-0-iff-eq* [*simp*]
declare *diff-le-0-iff-le* [*simp*]

ML ⟪
val add-zero-left = *thmadd-0*;
val add-zero-right = *thmadd-0-right*;

⟩⟩

ML ⟨⟨
val add-assoc = thm add-assoc;
val add-commute = thm add-commute;
val add-left-commute = thm add-left-commute;
val add-ac = thms add-ac;
val mult-assoc = thm mult-assoc;
val mult-commute = thm mult-commute;
val mult-left-commute = thm mult-left-commute;
val mult-ac = thms mult-ac;
val add-0 = thm add-0;
val mult-1-left = thm mult-1-left;
val mult-1-right = thm mult-1-right;
val mult-1 = thm mult-1;
val add-left-imp-eq = thm add-left-imp-eq;
val add-right-imp-eq = thm add-right-imp-eq;
val add-imp-eq = thm add-imp-eq;
val left-minus = thm left-minus;
val diff-minus = thm diff-minus;
val add-0-right = thm add-0-right;
val add-left-cancel = thm add-left-cancel;
val add-right-cancel = thm add-right-cancel;
val right-minus = thm right-minus;
val right-minus-eq = thm right-minus-eq;
val minus-minus = thm minus-minus;
val equals-zero-I = thm equals-zero-I;
val minus-zero = thm minus-zero;
val diff-self = thm diff-self;
val diff-0 = thm diff-0;
val diff-0-right = thm diff-0-right;
val diff-minus-eq-add = thm diff-minus-eq-add;
val neg-equal-iff-equal = thm neg-equal-iff-equal;
val neg-equal-0-iff-equal = thm neg-equal-0-iff-equal;
val neg-0-equal-iff-equal = thm neg-0-equal-iff-equal;
val equation-minus-iff = thm equation-minus-iff;
val minus-equation-iff = thm minus-equation-iff;
val minus-add-distrib = thm minus-add-distrib;
val minus-diff-eq = thm minus-diff-eq;
val add-left-mono = thm add-left-mono;
val add-le-imp-le-left = thm add-le-imp-le-left;
val add-right-mono = thm add-right-mono;
val add-mono = thm add-mono;
val add-strict-left-mono = thm add-strict-left-mono;
val add-strict-right-mono = thm add-strict-right-mono;
val add-strict-mono = thm add-strict-mono;
val add-less-le-mono = thm add-less-le-mono;
val add-le-less-mono = thm add-le-less-mono;
val add-less-imp-less-left = thm add-less-imp-less-left;

val add-less-imp-less-right = thm add-less-imp-less-right;
val add-less-cancel-left = thm add-less-cancel-left;
val add-less-cancel-right = thm add-less-cancel-right;
val add-le-cancel-left = thm add-le-cancel-left;
val add-le-cancel-right = thm add-le-cancel-right;
val add-le-imp-le-right = thm add-le-imp-le-right;
val add-increasing = thm add-increasing;
val le-imp-neg-le = thm le-imp-neg-le;
val neg-le-iff-le = thm neg-le-iff-le;
val neg-le-0-iff-le = thm neg-le-0-iff-le;
val neg-0-le-iff-le = thm neg-0-le-iff-le;
val neg-less-iff-less = thm neg-less-iff-less;
val neg-less-0-iff-less = thm neg-less-0-iff-less;
val neg-0-less-iff-less = thm neg-0-less-iff-less;
val less-minus-iff = thm less-minus-iff;
val minus-less-iff = thm minus-less-iff;
val le-minus-iff = thm le-minus-iff;
val minus-le-iff = thm minus-le-iff;
val add-diff-eq = thm add-diff-eq;
val diff-add-eq = thm diff-add-eq;
val diff-eq-eq = thm diff-eq-eq;
val eq-diff-eq = thm eq-diff-eq;
val diff-diff-eq = thm diff-diff-eq;
val diff-diff-eq2 = thm diff-diff-eq2;
val diff-add-cancel = thm diff-add-cancel;
val add-diff-cancel = thm add-diff-cancel;
val less-iff-diff-less-0 = thm less-iff-diff-less-0;
val diff-less-eq = thm diff-less-eq;
val less-diff-eq = thm less-diff-eq;
val diff-le-eq = thm diff-le-eq;
val le-diff-eq = thm le-diff-eq;
val compare-rls = thms compare-rls;
val eq-iff-diff-eq-0 = thm eq-iff-diff-eq-0;
val le-iff-diff-le-0 = thm le-iff-diff-le-0;
val add-meet-distrib-left = thm add-meet-distrib-left;
val add-join-distrib-left = thm add-join-distrib-left;
val is-join-neg-meet = thm is-join-neg-meet;
val is-meet-neg-join = thm is-meet-neg-join;
val add-join-distrib-right = thm add-join-distrib-right;
val add-meet-distrib-right = thm add-meet-distrib-right;
val add-meet-join-distribs = thms add-meet-join-distribs;
val join-eq-neg-meet = thm join-eq-neg-meet;
val meet-eq-neg-join = thm meet-eq-neg-join;
val add-eq-meet-join = thm add-eq-meet-join;
val prts = thm prts;
val zero-le-pprt = thm zero-le-pprt;
val nprt-le-zero = thm nprt-le-zero;
val le-eq-neg = thm le-eq-neg;
val join-0-imp-0 = thm join-0-imp-0;

val meet-0-imp-0 = thm meet-0-imp-0;
val join-0-eq-0 = thm join-0-eq-0;
val meet-0-eq-0 = thm meet-0-eq-0;
val zero-le-double-add-iff-zero-le-single-add = thm zero-le-double-add-iff-zero-le-single-add;
val double-add-le-zero-iff-single-add-le-zero = thm double-add-le-zero-iff-single-add-le-zero;
val double-add-less-zero-iff-single-less-zero = thm double-add-less-zero-iff-single-less-zero;
val abs-lattice = thm abs-lattice;
val abs-zero = thm abs-zero;
val abs-eq-0 = thm abs-eq-0;
val abs-0-eq = thm abs-0-eq;
val neg-meet-eq-join = thm neg-meet-eq-join;
val neg-join-eq-meet = thm neg-join-eq-meet;
val join-eq-if = thm join-eq-if;
val abs-if-lattice = thm abs-if-lattice;
val abs-ge-zero = thm abs-ge-zero;
val abs-le-zero-iff = thm abs-le-zero-iff;
val zero-less-abs-iff = thm zero-less-abs-iff;
val abs-not-less-zero = thm abs-not-less-zero;
val abs-ge-self = thm abs-ge-self;
val abs-ge-minus-self = thm abs-ge-minus-self;
val le-imp-join-eq = thm le-imp-join-eq;
val ge-imp-join-eq = thm ge-imp-join-eq;
val le-imp-meet-eq = thm le-imp-meet-eq;
val ge-imp-meet-eq = thm ge-imp-meet-eq;
val abs-prts = thm abs-prts;
val abs-minus-cancel = thm abs-minus-cancel;
val abs-idempotent = thm abs-idempotent;
val zero-le-iff-zero-nprt = thm zero-le-iff-zero-nprt;
val le-zero-iff-zero-pprt = thm le-zero-iff-zero-pprt;
val le-zero-iff-pprt-id = thm le-zero-iff-pprt-id;
val zero-le-iff-nprt-id = thm zero-le-iff-nprt-id;
val iff2imp = thm iff2imp;
(∗ *val imp-abs-id = thm imp-abs-id*;
val imp-abs-neg-id = thm imp-abs-neg-id; ∗)
val abs-leI = thm abs-leI;
val le-minus-self-iff = thm le-minus-self-iff;
val minus-le-self-iff = thm minus-le-self-iff;
val abs-le-D1 = thm abs-le-D1;
val abs-le-D2 = thm abs-le-D2;
val abs-le-iff = thm abs-le-iff;
val abs-triangle-ineq = thm abs-triangle-ineq;
val abs-diff-triangle-ineq = thm abs-diff-triangle-ineq;
⟩⟩

end

15 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.

axclass *semiring* $\subseteq$ *ab-semigroup-add*, *semigroup-mult*
  left-distrib: $(a + b) * c = a * c + b * c$
  right-distrib: $a * (b + c) = a * b + a * c$

axclass *semiring-0* $\subseteq$ *semiring*, *comm-monoid-add*

axclass *semiring-0-cancel* $\subseteq$ *semiring-0*, *cancel-ab-semigroup-add*

axclass *comm-semiring* $\subseteq$ *ab-semigroup-add*, *ab-semigroup-mult*
  distrib: $(a + b) * c = a * c + b * c$

instance *comm-semiring* $\subseteq$ *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

axclass *comm-semiring-0* $\subseteq$ *comm-semiring*, *comm-monoid-add*

instance *comm-semiring-0* $\subseteq$ *semiring-0* **..**

axclass *comm-semiring-0-cancel* $\subseteq$ *comm-semiring-0*, *cancel-ab-semigroup-add*

instance *comm-semiring-0-cancel* $\subseteq$ *semiring-0-cancel* **..**

axclass *axclass-0-neq-1* $\subseteq$ *zero*, *one*
  zero-neq-one [*simp*]: $0 \neq 1$

axclass *semiring-1* $\subseteq$ *axclass-0-neq-1*, *semiring-0*, *monoid-mult*

axclass *comm-semiring-1* $\subseteq$ *axclass-0-neq-1*, *comm-semiring-0*, *comm-monoid-mult*

instance *comm-semiring-1* $\subseteq$ *semiring-1* **..**

axclass *axclass-no-zero-divisors* $\subseteq$ *zero*, *times*
no-zero-divisors: $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$

axclass *semiring-1-cancel* $\subseteq$ *semiring-1*, *cancel-ab-semigroup-add*

instance *semiring-1-cancel* $\subseteq$ *semiring-0-cancel* **..**

axclass *comm-semiring-1-cancel* $\subseteq$ *comm-semiring-1*, *cancel-ab-semigroup-add*

instance *comm-semiring-1-cancel* $\subseteq$ *semiring-1-cancel* **..**

instance *comm-semiring-1-cancel* $\subseteq$ *comm-semiring-0-cancel* **..**

axclass *ring* $\subseteq$ *semiring*, *ab-group-add*

instance *ring* $\subseteq$ *semiring-0-cancel* **..**

axclass *comm-ring* $\subseteq$ *comm-semiring-0*, *ab-group-add*

instance *comm-ring* $\subseteq$ *ring* **..**

instance *comm-ring* $\subseteq$ *comm-semiring-0-cancel* **..**

axclass *ring-1* $\subseteq$ *ring*, *semiring-1*

instance *ring-1* $\subseteq$ *semiring-1-cancel* **..**

axclass *comm-ring-1* $\subseteq$ *comm-ring*, *comm-semiring-1*

instance *comm-ring-1* $\subseteq$ *ring-1* **..**

instance *comm-ring-1* $\subseteq$ *comm-semiring-1-cancel* **..**

axclass *idom* $\subseteq$ *comm-ring-1*, *axclass-no-zero-divisors*

axclass *field* $\subseteq$ *comm-ring-1*, *inverse*
left-inverse [*simp*]: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
divide-inverse: $a\ /\ b = a * inverse\ b$

lemma *mult-zero-left* [*simp*]: $0 * a = (0::'a::semiring\text{-}0\text{-}cancel)$
proof $-$

have *0∗a + 0∗a = 0∗a + 0*
by (*simp add*: *left-distrib* [*symmetric*])
thus *?thesis*
by (*simp only*: *add-left-cancel*)
qed

lemma *mult-zero-right* [*simp*]: *a ∗ 0 = (0::'a::semiring-0-cancel)*
proof −
have *a∗0 + a∗0 = a∗0 + 0*
by (*simp add*: *right-distrib* [*symmetric*])
thus *?thesis*
by (*simp only*: *add-left-cancel*)
qed

lemma *field-mult-eq-0-iff* [*simp*]: *(a∗b = (0::'a::field)) = (a = 0 | b = 0)*
proof *cases*
assume *a=0* **thus** *?thesis* **by** *simp*
next
assume *anz* [*simp*]: *a≠0*
{ assume *a ∗ b = 0*
hence *inverse a ∗ (a ∗ b) = 0* **by** *simp*
hence *b = 0* **by** (*simp* (*no-asm-use*) *add*: *mult-assoc* [*symmetric*])**}**
thus *?thesis* **by** *force*
qed

instance *field ⊆ idom*
by (*intro-classes*, *simp*)

axclass *division-by-zero ⊆ zero*, *inverse*
inverse-zero [*simp*]: *inverse 0 = 0*

15.1 Distribution rules

theorems *ring-distrib = right-distrib left-distrib*

For the *combine-numerals* simproc

lemma *combine-common-factor*:
a∗e + (b∗e + c) = (a+b)∗e + (c::'a::semiring)
by (*simp add*: *left-distrib add-ac*)

lemma *minus-mult-left*: *− (a ∗ b) = (−a) ∗ (b::'a::ring)*
apply (*rule equals-zero-I*)
apply (*simp add*: *left-distrib* [*symmetric*])
done

lemma *minus-mult-right*: *− (a ∗ b) = a ∗ −(b::'a::ring)*
apply (*rule equals-zero-I*)
apply (*simp add*: *right-distrib* [*symmetric*])
done

lemma *minus-mult-minus* [*simp*]: $(-\ a) * (-\ b) = a * (b::'a::ring)$
by (*simp add*: *minus-mult-left* [*symmetric*] *minus-mult-right* [*symmetric*])

lemma *minus-mult-commute*: $(-\ a) * b = a * (-\ b::'a::ring)$
by (*simp add*: *minus-mult-left* [*symmetric*] *minus-mult-right* [*symmetric*])

lemma *right-diff-distrib*: $a * (b - c) = a * b - a * (c::'a::ring)$
by (*simp add*: *right-distrib diff-minus*
minus-mult-left [*symmetric*] *minus-mult-right* [*symmetric*])

lemma *left-diff-distrib*: $(a - b) * c = a * c - b * (c::'a::ring)$
by (*simp add*: *left-distrib diff-minus*
minus-mult-left [*symmetric*] *minus-mult-right* [*symmetric*])

axclass *pordered-semiring* $\subseteq$ *semiring-0*, *pordered-ab-semigroup-add*
mult-left-mono: $a <= b \Longrightarrow 0 <= c \Longrightarrow c * a <= c * b$
mult-right-mono: $a <= b \Longrightarrow 0 <= c \Longrightarrow a * c <= b * c$

axclass *pordered-cancel-semiring* $\subseteq$ *pordered-semiring*, *cancel-ab-semigroup-add*

instance *pordered-cancel-semiring* $\subseteq$ *semiring-0-cancel* **..**

axclass *ordered-semiring-strict* $\subseteq$ *semiring-0*, *ordered-cancel-ab-semigroup-add*
mult-strict-left-mono: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
mult-strict-right-mono: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$

instance *ordered-semiring-strict* $\subseteq$ *semiring-0-cancel* **..**

instance *ordered-semiring-strict* $\subseteq$ *pordered-cancel-semiring*
apply *intro-classes*
apply (*case-tac* $a < b\ \&\ 0 < c$)
apply (*auto simp add*: *mult-strict-left-mono order-less-le*)
apply (*auto simp add*: *mult-strict-left-mono order-le-less*)
apply (*simp add*: *mult-strict-right-mono*)
done

axclass *pordered-comm-semiring* $\subseteq$ *comm-semiring-0*, *pordered-ab-semigroup-add*
mult-mono: $a <= b \Longrightarrow 0 <= c \Longrightarrow c * a <= c * b$

axclass *pordered-cancel-comm-semiring* $\subseteq$ *pordered-comm-semiring*, *cancel-ab-semigroup-add*

instance *pordered-cancel-comm-semiring* $\subseteq$ *pordered-comm-semiring* **..**

axclass *ordered-comm-semiring-strict* $\subseteq$ *comm-semiring-0*, *ordered-cancel-ab-semigroup-add*
mult-strict-mono: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$

instance *pordered-comm-semiring* $\subseteq$ *pordered-semiring*
by (*intro-classes*, *insert mult-mono*, *simp-all add*: *mult-commute*, *blast*+)

instance *pordered-cancel-comm-semiring* ⊆ *pordered-cancel-semiring* **..**

instance *ordered-comm-semiring-strict* ⊆ *ordered-semiring-strict*
by (*intro-classes*, *insert mult-strict-mono*, *simp-all add*: *mult-commute*, *blast*+)

instance *ordered-comm-semiring-strict* ⊆ *pordered-cancel-comm-semiring*
apply (*intro-classes*)
apply (*case-tac* $a < b$ & $0 < c$)
apply (*auto simp add*: *mult-strict-left-mono order-less-le*)
apply (*auto simp add*: *mult-strict-left-mono order-le-less*)
done

axclass *pordered-ring* ⊆ *ring*, *pordered-semiring*

instance *pordered-ring* ⊆ *pordered-ab-group-add* **..**

instance *pordered-ring* ⊆ *pordered-cancel-semiring* **..**

axclass *lordered-ring* ⊆ *pordered-ring*, *lordered-ab-group-abs*

instance *lordered-ring* ⊆ *lordered-ab-group-meet* **..**

instance *lordered-ring* ⊆ *lordered-ab-group-join* **..**

axclass *axclass-abs-if* ⊆ *minus*, *ord*, *zero*
abs-if: *abs a* = (*if* (*a* < *0*) *then* (−*a*) *else a*)

axclass *ordered-ring-strict* ⊆ *ring*, *ordered-semiring-strict*, *axclass-abs-if*

instance *ordered-ring-strict* ⊆ *lordered-ab-group* **..**

instance *ordered-ring-strict* ⊆ *lordered-ring*
by (*intro-classes*, *simp add*: *abs-if join-eq-if*)

axclass *pordered-comm-ring* ⊆ *comm-ring*, *pordered-comm-semiring*

axclass *ordered-semidom* ⊆ *comm-semiring-1-cancel*, *ordered-comm-semiring-strict*

zero-less-one [*simp*]: *0* < *1*

axclass *ordered-idom* ⊆ *comm-ring-1*, *ordered-comm-semiring-strict*, *axclass-abs-if*

instance *ordered-idom* ⊆ *ordered-ring-strict* **..**

axclass *ordered-field* ⊆ *field*, *ordered-idom*

lemmas *linorder-neqE-ordered-idom* =

linorder-neqE[**where** *'a = ?'b*::*ordered-idom*]

lemma *eq-add-iff1*:
(*a*∗*e* + *c* = *b*∗*e* + *d*) = ((*a*−*b*)∗*e* + *c* = (*d*::*'a*::*ring*))
apply (*simp add*: *diff-minus left-distrib*)
apply (*simp add*: *diff-minus left-distrib add-ac*)
apply (*simp add*: *compare-rls minus-mult-left* [*symmetric*])
done

lemma *eq-add-iff2*:
(*a*∗*e* + *c* = *b*∗*e* + *d*) = (*c* = (*b*−*a*)∗*e* + (*d*::*'a*::*ring*))
apply (*simp add*: *diff-minus left-distrib add-ac*)
apply (*simp add*: *compare-rls minus-mult-left* [*symmetric*])
done

lemma *less-add-iff1*:
(*a*∗*e* + *c* < *b*∗*e* + *d*) = ((*a*−*b*)∗*e* + *c* < (*d*::*'a*::*pordered-ring*))
apply (*simp add*: *diff-minus left-distrib add-ac*)
apply (*simp add*: *compare-rls minus-mult-left* [*symmetric*])
done

lemma *less-add-iff2*:
(*a*∗*e* + *c* < *b*∗*e* + *d*) = (*c* < (*b*−*a*)∗*e* + (*d*::*'a*::*pordered-ring*))
apply (*simp add*: *diff-minus left-distrib add-ac*)
apply (*simp add*: *compare-rls minus-mult-left* [*symmetric*])
done

lemma *le-add-iff1*:
(*a*∗*e* + *c* ≤ *b*∗*e* + *d*) = ((*a*−*b*)∗*e* + *c* ≤ (*d*::*'a*::*pordered-ring*))
apply (*simp add*: *diff-minus left-distrib add-ac*)
apply (*simp add*: *compare-rls minus-mult-left* [*symmetric*])
done

lemma *le-add-iff2*:
(*a*∗*e* + *c* ≤ *b*∗*e* + *d*) = (*c* ≤ (*b*−*a*)∗*e* + (*d*::*'a*::*pordered-ring*))
apply (*simp add*: *diff-minus left-distrib add-ac*)
apply (*simp add*: *compare-rls minus-mult-left* [*symmetric*])
done

15.2 Ordering Rules for Multiplication

lemma *mult-left-le-imp-le*:
[|*c*∗*a* ≤ *c*∗*b*; *0* < *c*|] ==> *a* ≤ (*b*::*'a*::*ordered-semiring-strict*)
by (*force simp add*: *mult-strict-left-mono linorder-not-less* [*symmetric*])

lemma *mult-right-le-imp-le*:
[|*a*∗*c* ≤ *b*∗*c*; *0* < *c*|] ==> *a* ≤ (*b*::*'a*::*ordered-semiring-strict*)
by (*force simp add*: *mult-strict-right-mono linorder-not-less* [*symmetric*])

lemma *mult-left-less-imp-less*:
[|$c*a < c*b$; $0 \leq c$|] ==> $a < (b::'a::ordered\text{-}semiring\text{-}strict)$
by (*force simp add*: *mult-left-mono linorder-not-le* [*symmetric*])

lemma *mult-right-less-imp-less*:
[|$a*c < b*c$; $0 \leq c$|] ==> $a < (b::'a::ordered\text{-}semiring\text{-}strict)$
by (*force simp add*: *mult-right-mono linorder-not-le* [*symmetric*])

lemma *mult-strict-left-mono-neg*:
[|$b < a$; $c < 0$|] ==> $c * a < c * (b::'a::ordered\text{-}ring\text{-}strict)$
apply (*drule mult-strict-left-mono* [*of - - −c*])
apply (*simp-all add*: *minus-mult-left* [*symmetric*])
done

lemma *mult-left-mono-neg*:
[|$b \leq a$; $c \leq 0$|] ==> $c * a \leq c * (b::'a::pordered\text{-}ring)$
apply (*drule mult-left-mono* [*of - - −c*])
apply (*simp-all add*: *minus-mult-left* [*symmetric*])
done

lemma *mult-strict-right-mono-neg*:
[|$b < a$; $c < 0$|] ==> $a * c < b * (c::'a::ordered\text{-}ring\text{-}strict)$
apply (*drule mult-strict-right-mono* [*of - - −c*])
apply (*simp-all add*: *minus-mult-right* [*symmetric*])
done

lemma *mult-right-mono-neg*:
[|$b \leq a$; $c \leq 0$|] ==> $a * c \leq (b::'a::pordered\text{-}ring) * c$
apply (*drule mult-right-mono* [*of - - −c*])
apply (*simp*)
apply (*simp-all add*: *minus-mult-right* [*symmetric*])
done

15.3 Products of Signs

lemma *mult-pos-pos*: [| $(0::'a::ordered\text{-}semiring\text{-}strict) < a$; $0 < b$ |] ==> $0 < a*b$
by (*drule mult-strict-left-mono* [*of 0 b*], *auto*)

lemma *mult-nonneg-nonneg*: [| $(0::'a::pordered\text{-}cancel\text{-}semiring) \leq a$; $0 \leq b$ |] ==> $0 \leq a*b$
by (*drule mult-left-mono* [*of 0 b*], *auto*)

lemma *mult-pos-neg*: [| $(0::'a::ordered\text{-}semiring\text{-}strict) < a$; $b < 0$ |] ==> $a*b < 0$
by (*drule mult-strict-left-mono* [*of b 0*], *auto*)

lemma *mult-nonneg-nonpos*: [| $(0::'a::pordered\text{-}cancel\text{-}semiring) \leq a$; $b \leq 0$ |] ==> $a*b \leq 0$

by (*drule mult-left-mono* [*of b 0*], *auto*)

lemma *mult-pos-neg2*: [| (*0*::'*a*::*ordered-semiring-strict*) < *a*; *b* < *0* |] ==> *b*∗*a* < *0*
by (*drule mult-strict-right-mono*[*of b 0*], *auto*)

lemma *mult-nonneg-nonpos2*: [| (*0*::'*a*::*pordered-cancel-semiring*) ≤ *a*; *b* ≤ *0* |] ==> *b*∗*a* ≤ *0*
by (*drule mult-right-mono*[*of b 0*], *auto*)

lemma *mult-neg-neg*: [| *a* < (*0*::'*a*::*ordered-ring-strict*); *b* < *0* |] ==> *0* < *a*∗*b*
by (*drule mult-strict-right-mono-neg*, *auto*)

lemma *mult-nonpos-nonpos*: [| *a* ≤ (*0*::'*a*::*pordered-ring*); *b* ≤ *0* |] ==> *0* ≤ *a*∗*b*
by (*drule mult-right-mono-neg*[*of a 0 b*], *auto*)

lemma *zero-less-mult-pos*:
[| *0* < *a*∗*b*; *0* < *a*|] ==> *0* < (*b*::'*a*::*ordered-semiring-strict*)
apply (*case-tac b*≤*0*)
apply (*auto simp add*: *order-le-less linorder-not-less*)
apply (*drule-tac mult-pos-neg* [*of a b*])
apply (*auto dest*: *order-less-not-sym*)
done

lemma *zero-less-mult-pos2*:
[| *0* < *b*∗*a*; *0* < *a*|] ==> *0* < (*b*::'*a*::*ordered-semiring-strict*)
apply (*case-tac b*≤*0*)
apply (*auto simp add*: *order-le-less linorder-not-less*)
apply (*drule-tac mult-pos-neg2* [*of a b*])
apply (*auto dest*: *order-less-not-sym*)
done

lemma *zero-less-mult-iff*:
((*0*::'*a*::*ordered-ring-strict*) < *a*∗*b*) = (*0* < *a* & *0* < *b* | *a* < *0* & *b* < *0*)
apply (*auto simp add*: *order-le-less linorder-not-less mult-pos-pos*
mult-neg-neg)
apply (*blast dest*: *zero-less-mult-pos*)
apply (*blast dest*: *zero-less-mult-pos2*)
done

A field has no "zero divisors", and this theorem holds without the assumption of an ordering. See *field-mult-eq-0-iff* below.

lemma *mult-eq-0-iff* [*simp*]: (*a*∗*b* = (*0*::'*a*::*ordered-ring-strict*)) = (*a* = *0* | *b* = *0*)
apply (*case-tac a* < *0*)
apply (*auto simp add*: *linorder-not-less order-le-less linorder-neq-iff*)
apply (*force dest*: *mult-strict-right-mono-neg mult-strict-right-mono*)+
done

lemma *zero-le-mult-iff*:
  $((0::'a::ordered\text{-}ring\text{-}strict) \leq a*b) = (0 \leq a \ \& \ 0 \leq b \mid a \leq 0 \ \& \ b \leq 0)$
by (*auto simp add*: *eq-commute* [*of 0*] *order-le-less linorder-not-less*
    zero-less-mult-iff)

lemma *mult-less-0-iff*:
  $(a*b < (0::'a::ordered\text{-}ring\text{-}strict)) = (0 < a \ \& \ b < 0 \mid a < 0 \ \& \ 0 < b)$
apply (*insert zero-less-mult-iff* [*of* $-a$ *b*])
apply (*force simp add*: *minus-mult-left*[*symmetric*])
done

lemma *mult-le-0-iff*:
  $(a*b \leq (0::'a::ordered\text{-}ring\text{-}strict)) = (0 \leq a \ \& \ b \leq 0 \mid a \leq 0 \ \& \ 0 \leq b)$
apply (*insert zero-le-mult-iff* [*of* $-a$ *b*])
apply (*force simp add*: *minus-mult-left*[*symmetric*])
done

lemma *split-mult-pos-le*: $(0 \leq a \ \& \ 0 \leq b) \mid (a \leq 0 \ \& \ b \leq 0) \Longrightarrow 0 \leq a * (b::-::pordered\text{-}ring)$
by (*auto simp add*: *mult-nonneg-nonneg mult-nonpos-nonpos*)

lemma *split-mult-neg-le*: $(0 \leq a \ \& \ b \leq 0) \mid (a \leq 0 \ \& \ 0 \leq b) \Longrightarrow a * b \leq (0::-::pordered\text{-}cancel\text{-}semiring)$
by (*auto simp add*: *mult-nonneg-nonpos mult-nonneg-nonpos2*)

lemma *zero-le-square*: $(0::'a::ordered\text{-}ring\text{-}strict) \leq a*a$
by (*simp add*: *zero-le-mult-iff linorder-linear*)

Proving axiom *zero-less-one* makes all *ordered-semidom* theorems available to members of *ordered-idom*

instance *ordered-idom* $\subseteq$ *ordered-semidom*
proof
  have $(0::'a) \leq 1*1$ **by** (*rule zero-le-square*)
  thus $(0::'a) < 1$ **by** (*simp add*: *order-le-less*)
qed

instance *ordered-ring-strict* $\subseteq$ *axclass-no-zero-divisors*
by (*intro-classes*, *simp*)

instance *ordered-idom* $\subseteq$ *idom* **..**

All three types of comparision involving 0 and 1 are covered.

lemmas *one-neq-zero* = *zero-neq-one* [*THEN not-sym*]
declare *one-neq-zero* [*simp*]

lemma *zero-le-one* [*simp*]: $(0::'a::ordered\text{-}semidom) \leq 1$
  by (*rule zero-less-one* [*THEN order-less-imp-le*])

lemma *not-one-le-zero* [*simp*]: $\sim (1::'a::ordered\text{-}semidom) \leq 0$

by (*simp add*: *linorder-not-le*)

lemma *not-one-less-zero* [*simp*]: ~ (*1*::'*a*::*ordered-semidom*) < *0*
by (*simp add*: *linorder-not-less*)

15.4 More Monotonicity

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
    [|*a*<*b*; *c*<*d*; *0*<*b*; *0*≤*c*|] ==> *a* * *c* < *b* * (*d*::'*a*::*ordered-semiring-strict*)
apply (*case-tac c=0*)
 apply (*simp add*: *mult-pos-pos*)
apply (*erule mult-strict-right-mono* [*THEN order-less-trans*])
 apply (*force simp add*: *order-le-less*)
apply (*erule mult-strict-left-mono*, *assumption*)
done

This weaker variant has more natural premises

lemma *mult-strict-mono'*:
    [| *a*<*b*; *c*<*d*; *0* ≤ *a*; *0* ≤ *c*|] ==> *a* * *c* < *b* * (*d*::'*a*::*ordered-semiring-strict*)
apply (*rule mult-strict-mono*)
apply (*blast intro*: *order-le-less-trans*)+
done

lemma *mult-mono*:
    [|*a* ≤ *b*; *c* ≤ *d*; *0* ≤ *b*; *0* ≤ *c*|]
     ==> *a* * *c* ≤ *b* * (*d*::'*a*::*pordered-semiring*)
apply (*erule mult-right-mono* [*THEN order-trans*], *assumption*)
apply (*erule mult-left-mono*, *assumption*)
done

lemma *less-1-mult*: [| *1* < *m*; *1* < *n* |] ==> *1* < *m**(*n*::'*a*::*ordered-semidom*)
apply (*insert mult-strict-mono* [*of 1 m 1 n*])
apply (*simp add*: *order-less-trans* [*OF zero-less-one*])
done

lemma *mult-less-le-imp-less*: (*a*::'*a*::*ordered-semiring-strict*) < *b* ==>
    c <= *d* ==> *0* <= *a* ==> *0* < *c* ==> *a* * *c* < *b* * *d*
  apply (*subgoal-tac a* * *c* < *b* * *c*)
  apply (*erule order-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*: (*a*::'*a*::*ordered-semiring-strict*) <= *b* ==>
    c < *d* ==> *0* < *a* ==> *0* <= *c* ==> *a* * *c* < *b* * *d*
  apply (*subgoal-tac a* * *c* <= *b* * *c*)

apply (*erule order-le-less-trans*)
apply (*erule mult-strict-left-mono*)
apply *simp*
apply (*erule mult-right-mono*)
apply *simp*
done

15.5 Cancellation Laws for Relationships With a Common Factor

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*:
$(a*c < b*c) = ((0 < c \ \&\ a < b) \mid (c < 0 \ \&\ b < (a::'a::ordered\text{-}ring\text{-}strict)))$
apply (*case-tac* $c = 0$)
apply (*auto simp add*: *linorder-neq-iff mult-strict-right-mono*
mult-strict-right-mono-neg)
apply (*auto simp add*: *linorder-not-less*
linorder-not-le [*symmetric*, *of* $a*c$]
linorder-not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *order-less-imp-le mult-right-mono*
mult-right-mono-neg)
done

lemma *mult-less-cancel-left-disj*:
$(c*a < c*b) = ((0 < c \ \&\ a < b) \mid (c < 0 \ \&\ b < (a::'a::ordered\text{-}ring\text{-}strict)))$
apply (*case-tac* $c = 0$)
apply (*auto simp add*: *linorder-neq-iff mult-strict-left-mono*
mult-strict-left-mono-neg)
apply (*auto simp add*: *linorder-not-less*
linorder-not-le [*symmetric*, *of* $c*a$]
linorder-not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *order-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*:
fixes $c :: 'a :: ordered\text{-}ring\text{-}strict$
shows $(a*c < b*c) = ((0 \leq c \longrightarrow a < b) \ \&\ (c \leq 0 \longrightarrow b < a))$
by (*insert mult-less-cancel-right-disj* [*of* a c b], *auto*)

lemma *mult-less-cancel-left*:
fixes $c :: 'a :: ordered\text{-}ring\text{-}strict$
shows $(c*a < c*b) = ((0 \leq c \longrightarrow a < b) \,\&\, (c \leq 0 \longrightarrow b < a))$
by (*insert mult-less-cancel-left-disj* [*of c a b*], *auto*)

lemma *mult-le-cancel-right*:
$(a*c \leq b*c) = ((0<c \longrightarrow a \leq b) \,\&\, (c<0 \longrightarrow b \leq (a::'a::ordered\text{-}ring\text{-}strict)))$
by (*simp add*: *linorder-not-less* [*symmetric*] *mult-less-cancel-right-disj*)

lemma *mult-le-cancel-left*:
$(c*a \leq c*b) = ((0<c \longrightarrow a \leq b) \,\&\, (c<0 \longrightarrow b \leq (a::'a::ordered\text{-}ring\text{-}strict)))$
by (*simp add*: *linorder-not-less* [*symmetric*] *mult-less-cancel-left-disj*)

lemma *mult-less-imp-less-left*:
assumes *less*: $c*a < c*b$ **and** *nonneg*: $0 \leq c$
shows $a < (b::'a::ordered\text{-}semiring\text{-}strict)$
proof (*rule ccontr*)
assume $\sim a < b$
hence $b \leq a$ **by** (*simp add*: *linorder-not-less*)
hence $c*b \leq c*a$ **by** (*rule mult-left-mono*)
with *this* **and** *less* **show** *False*
by (*simp add*: *linorder-not-less* [*symmetric*])
qed

lemma *mult-less-imp-less-right*:
assumes *less*: $a*c < b*c$ **and** *nonneg*: $0 <= c$
shows $a < (b::'a::ordered\text{-}semiring\text{-}strict)$
proof (*rule ccontr*)
assume $\sim a < b$
hence $b \leq a$ **by** (*simp add*: *linorder-not-less*)
hence $b*c \leq a*c$ **by** (*rule mult-right-mono*)
with *this* **and** *less* **show** *False*
by (*simp add*: *linorder-not-less* [*symmetric*])
qed

Cancellation of equalities with a common factor

lemma *mult-cancel-right* [*simp*]:
$(a*c = b*c) = (c = (0::'a::ordered\text{-}ring\text{-}strict) \mid a=b)$
apply (*cut-tac linorder-less-linear* [*of 0 c*])
apply (*force dest*: *mult-strict-right-mono-neg mult-strict-right-mono*
simp add: *linorder-neq-iff*)
done

These cancellation theorems require an ordering. Versions are proved below that work for fields without an ordering.

lemma *mult-cancel-left* [*simp*]:
$(c*a = c*b) = (c = (0::'a::ordered\text{-}ring\text{-}strict) \mid a=b)$
apply (*cut-tac linorder-less-linear* [*of 0 c*])
apply (*force dest*: *mult-strict-left-mono-neg mult-strict-left-mono*

simp add: *linorder-neq-iff*)
done

15.5.1 Special Cancellation Simprules for Multiplication

These also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(c \leq b*c) = ((0<c \longrightarrow 1\leq b) \ \&\ (c<0 \longrightarrow b \leq 1))$
by (*insert mult-le-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-le-cancel-right2*:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(a*c \leq c) = ((0<c \longrightarrow a\leq 1) \ \&\ (c<0 \longrightarrow 1 \leq a))$
by (*insert mult-le-cancel-right* [*of a c 1*], *simp*)

lemma *mult-le-cancel-left1*:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(c \leq c*b) = ((0<c \longrightarrow 1\leq b) \ \&\ (c<0 \longrightarrow b \leq 1))$
by (*insert mult-le-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-le-cancel-left2*:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(c*a \leq c) = ((0<c \longrightarrow a\leq 1) \ \&\ (c<0 \longrightarrow 1 \leq a))$
by (*insert mult-le-cancel-left* [*of c a 1*], *simp*)

lemma *mult-less-cancel-right1*:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(c < b*c) = ((0 \leq c \longrightarrow 1<b) \ \&\ (c \leq 0 \longrightarrow b < 1))$
by (*insert mult-less-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-less-cancel-right2*:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(a*c < c) = ((0 \leq c \longrightarrow a<1) \ \&\ (c \leq 0 \longrightarrow 1 < a))$
by (*insert mult-less-cancel-right* [*of a c 1*], *simp*)

lemma *mult-less-cancel-left1*:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(c < c*b) = ((0 \leq c \longrightarrow 1<b) \ \&\ (c \leq 0 \longrightarrow b < 1))$
by (*insert mult-less-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-less-cancel-left2*:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(c*a < c) = ((0 \leq c \longrightarrow a<1) \ \&\ (c \leq 0 \longrightarrow 1 < a))$
by (*insert mult-less-cancel-left* [*of c a 1*], *simp*)

lemma *mult-cancel-right1* [*simp*]:
fixes $c :: 'a :: ordered\text{-}idom$
shows $(c = b*c) = (c = 0 \mid b=1)$

by (*insert mult-cancel-right* [*of 1 c b*], *force*)

lemma *mult-cancel-right2* [*simp*]:
fixes *c* :: *'a* :: *ordered-idom*
  shows (*a***c* = *c*) = (*c* = *0* | *a*=*1*)
by (*insert mult-cancel-right* [*of a c 1*], *simp*)

lemma *mult-cancel-left1* [*simp*]:
fixes *c* :: *'a* :: *ordered-idom*
  shows (*c* = *c***b*) = (*c* = *0* | *b*=*1*)
by (*insert mult-cancel-left* [*of c 1 b*], *force*)

lemma *mult-cancel-left2* [*simp*]:
fixes *c* :: *'a* :: *ordered-idom*
  shows (*c***a* = *c*) = (*c* = *0* | *a*=*1*)
by (*insert mult-cancel-left* [*of c a 1*], *simp*)

Simprules for comparisons where common factors can be cancelled.

lemmas *mult-compare-simps* =
    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

This list of rewrites decides ring equalities by ordered rewriting.

lemmas *ring-eq-simps* =

  left-distrib right-distrib left-diff-distrib right-diff-distrib
  group-eq-simps

15.6 Fields

lemma *right-inverse* [*simp*]:
    assumes *not0*: *a* ≠ *0* **shows** *a* * *inverse* (*a*::*'a*::*field*) = *1*
proof −
  have *a* * *inverse a* = *inverse a* * *a* **by** (*simp add*: *mult-ac*)
  also have ... = *1* **using** *not0* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *right-inverse-eq*: *b* ≠ *0* ==> (*a* / *b* = *1*) = (*a* = (*b*::*'a*::*field*))
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::'a::field) = 1/a
by (simp add: divide-inverse)

lemma divide-self: a ≠ 0 ==> a / (a::'a::field) = 1
  by (simp add: divide-inverse)

lemma divide-zero [simp]: a / 0 = (0::'a::{field,division-by-zero})
by (simp add: divide-inverse)

lemma divide-self-if [simp]:
     a / (a::'a::{field,division-by-zero}) = (if a=0 then 0 else 1)
  by (simp add: divide-self)

lemma divide-zero-left [simp]: 0/a = (0::'a::field)
by (simp add: divide-inverse)

lemma inverse-eq-divide: inverse (a::'a::field) = 1/a
by (simp add: divide-inverse)

lemma add-divide-distrib: (a+b)/(c::'a::field) = a/c + b/c
by (simp add: divide-inverse left-distrib)
```

Compared with *mult-eq-0-iff*, this version removes the requirement of an ordering.

```
lemma field-mult-eq-0-iff [simp]: (a*b = (0::'a::field)) = (a = 0 | b = 0)
proof cases
  assume a=0 thus ?thesis by simp
next
  assume anz [simp]: a≠0
  { assume a * b = 0
    hence inverse a * (a * b) = 0 by simp
    hence b = 0  by (simp (no-asm-use) add: mult-assoc [symmetric])}
  thus ?thesis by force
qed
```

Cancellation of equalities with a common factor

```
lemma field-mult-cancel-right-lemma:
     assumes cnz: c ≠ (0::'a::field)
        and eq:  a*c = b*c
       shows a=b
```

proof −
have $(a * c) * inverse\ c = (b * c) * inverse\ c$
by (*simp add*: *eq*)
thus *a=b*
by (*simp add*: *mult-assoc cnz*)
qed

lemma *field-mult-cancel-right* [*simp*]:
$(a*c = b*c) = (c = (0::'a::field) \mid a=b)$
proof *cases*
assume *c=0* **thus** *?thesis* **by** *simp*
next
assume *c≠0*
thus *?thesis* **by** (*force dest*: *field-mult-cancel-right-lemma*)
qed

lemma *field-mult-cancel-left* [*simp*]:
$(c*a = c*b) = (c = (0::'a::field) \mid a=b)$
by (*simp add*: *mult-commute* [*of c*] *field-mult-cancel-right*)

lemma *nonzero-imp-inverse-nonzero*: $a \neq 0 ==> inverse\ a \neq (0::'a::field)$
proof
assume *ianz*: *inverse a = 0*
assume *a ≠ 0*
hence *1 = a * inverse a* **by** *simp*
also have *... = 0* **by** (*simp add*: *ianz*)
finally have *1 = (0::'a::field)* **.**
thus *False* **by** (*simp add*: *eq-commute*)
qed

15.7 Basic Properties of *inverse*

lemma *inverse-zero-imp-zero*: $inverse\ a = 0 ==> a = (0::'a::field)$
apply (*rule ccontr*)
apply (*blast dest*: *nonzero-imp-inverse-nonzero*)
done

lemma *inverse-nonzero-imp-nonzero*:
$inverse\ a = 0 ==> a = (0::'a::field)$
apply (*rule ccontr*)
apply (*blast dest*: *nonzero-imp-inverse-nonzero*)
done

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
$(inverse\ a = 0) = (a = (0::'a::\{field,division\text{-}by\text{-}zero\}))$
by (*force dest*: *inverse-nonzero-imp-nonzero*)

lemma *nonzero-inverse-minus-eq*:
assumes [*simp*]: *a≠0* **shows** $inverse(-a) = -inverse(a::'a::field)$

proof −
have $-a * inverse\ (-\ a) = -a * -\ inverse\ a$
by *simp*
thus *?thesis*
by (*simp only*: *field-mult-cancel-left*, *simp*)
qed

lemma *inverse-minus-eq* [*simp*]:
$inverse(-a) = -inverse(a::'a::\{field,division\text{-}by\text{-}zero\})$
proof *cases*
assume *a=0* **thus** *?thesis* **by** (*simp add*: *inverse-zero*)
next
assume $a\neq 0$
thus *?thesis* **by** (*simp add*: *nonzero-inverse-minus-eq*)
qed

lemma *nonzero-inverse-eq-imp-eq*:
assumes *inveq*: *inverse a* = *inverse b*
and *anz*: $a \neq 0$
and *bnz*: $b \neq 0$
shows $a = (b::'a::field)$
proof −
have $a * inverse\ b = a * inverse\ a$
by (*simp add*: *inveq*)
hence $(a * inverse\ b) * b = (a * inverse\ a) * b$
by *simp*
thus $a = b$
by (*simp add*: *mult-assoc anz bnz*)
qed

lemma *inverse-eq-imp-eq*:
inverse a = *inverse b* ==> $a = (b::'a::\{field,division\text{-}by\text{-}zero\})$
apply (*case-tac 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*]:
$(inverse\ a = inverse\ b) = (a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
by (*force dest*!: *inverse-eq-imp-eq*)

lemma *nonzero-inverse-inverse-eq*:
assumes [*simp*]: $a \neq 0$ **shows** $inverse(inverse\ (a::'a::field)) = a$
proof −
have $(inverse\ (inverse\ a) * inverse\ a) * a = a$
by (*simp add*: *nonzero-imp-inverse-nonzero*)
thus *?thesis*
by (*simp add*: *mult-assoc*)

```
  qed

lemma inverse-inverse-eq [simp]:
     inverse(inverse (a::'a::{field,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

lemma inverse-1 [simp]: inverse 1 = (1::'a::field)
  proof −
  have inverse 1 * 1 = (1::'a::field)
    by (rule left-inverse [OF zero-neq-one [symmetric]])
  thus ?thesis  by simp
  qed

lemma inverse-unique:
  assumes ab: a*b = 1
  shows inverse a = (b::'a::field)
proof −
  have a ≠ 0 using ab by auto
  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-mult-distrib:
      assumes anz: a ≠ 0
         and bnz: b ≠ 0
      shows inverse(a*b) = inverse(b) * inverse(a::'a::field)
  proof −
  have inverse(a*b) * (a * b) * inverse(b) = inverse(b)
    by (simp add: field-mult-eq-0-iff anz bnz)
  hence inverse(a*b) * a = inverse(b)
    by (simp add: mult-assoc bnz)
  hence inverse(a*b) * a * inverse(a) = inverse(b) * inverse(a)
    by simp
  thus ?thesis
    by (simp add: mult-assoc anz)
  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
```

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::'a::field)
apply (simp add: left-distrib mult-assoc)
apply (simp add: mult-commute [of inverse a])
apply (simp add: mult-assoc [symmetric] add-commute)
done

lemma inverse-divide [simp]:
      inverse (a/b) = b / (a::'a::{field,division-by-zero})
  by (simp add: divide-inverse mult-commute)
```

15.8 Calculations with fractions

```
lemma nonzero-mult-divide-cancel-left:
  assumes [simp]: b≠0 and [simp]: c≠0
    shows (c*a)/(c*b) = a/(b::'a::field)
proof -
  have (c*a)/(c*b) = c * a * (inverse b * inverse c)
    by (simp add: field-mult-eq-0-iff 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 mult-divide-cancel-left:
     c≠0 ==> (c*a) / (c*b) = a / (b::'a::{field,division-by-zero})
apply (case-tac b = 0)
apply (simp-all add: nonzero-mult-divide-cancel-left)
done

lemma nonzero-mult-divide-cancel-right:
     [|b≠0; c≠0|] ==> (a*c) / (b*c) = a/(b::'a::field)
by (simp add: mult-commute [of - c] nonzero-mult-divide-cancel-left)

lemma mult-divide-cancel-right:
     c≠0 ==> (a*c) / (b*c) = a / (b::'a::{field,division-by-zero})
apply (case-tac b = 0)
apply (simp-all add: nonzero-mult-divide-cancel-right)
```

done

lemma *mult-divide-cancel-eq-if*:
(*c***a*) / (*c***b*) =
(*if c=0 then 0 else a* / (*b*::'*a*::{*field*,*division-by-zero*}))
by (*simp add*: *mult-divide-cancel-left*)

lemma *divide-1* [*simp*]: *a*/*1* = (*a*::'*a*::*field*)
by (*simp add*: *divide-inverse*)

lemma *times-divide-eq-right*: *a* * (*b*/*c*) = (*a***b*) / (*c*::'*a*::*field*)
by (*simp add*: *divide-inverse mult-assoc*)

lemma *times-divide-eq-left*: (*b*/*c*) * *a* = (*b***a*) / (*c*::'*a*::*field*)
by (*simp add*: *divide-inverse mult-ac*)

lemma *divide-divide-eq-right* [*simp*]:
a / (*b*/*c*) = (*a***c*) / (*b*::'*a*::{*field*,*division-by-zero*})
by (*simp add*: *divide-inverse mult-ac*)

lemma *divide-divide-eq-left* [*simp*]:
(*a* / *b*) / (*c*::'*a*::{*field*,*division-by-zero*}) = *a* / (*b***c*)
by (*simp add*: *divide-inverse mult-assoc*)

lemma *add-frac-eq*: (*y*::'*a*::*field*) ~= *0* ==> *z* ~= *0* ==>
x / *y* + *w* / *z* = (*x* * *z* + *w* * *y*) / (*y* * *z*)
apply (*subgoal-tac x* / *y* = (*x* * *z*) / (*y* * *z*))
apply (*erule ssubst*)
apply (*subgoal-tac w* / *z* = (*w* * *y*) / (*y* * *z*))
apply (*erule ssubst*)
apply (*rule add-divide-distrib* [*THEN sym*])
apply (*subst mult-commute*)
apply (*erule nonzero-mult-divide-cancel-left* [*THEN sym*])
apply *assumption*
apply (*erule nonzero-mult-divide-cancel-right* [*THEN sym*])
apply *assumption*
done

15.8.1 Special Cancellation Simprules for Division

lemma *mult-divide-cancel-left-if* [*simp*]:
fixes *c* :: '*a* :: {*field*,*division-by-zero*}
shows (*c***a*) / (*c***b*) = (*if c=0 then 0 else a*/*b*)
by (*simp add*: *mult-divide-cancel-left*)

lemma *mult-divide-cancel-right-if* [*simp*]:
fixes *c* :: '*a* :: {*field*,*division-by-zero*}
shows (*a***c*) / (*b***c*) = (*if c=0 then 0 else a*/*b*)

by (*simp add*: *mult-divide-cancel-right*)

lemma *mult-divide-cancel-left-if1* [*simp*]:
fixes $c :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $c \,/\, (c*b) = (if\ c=0\ then\ 0\ else\ 1/b)$
apply (*insert mult-divide-cancel-left-if* [*of c 1 b*])
apply (*simp del*: *mult-divide-cancel-left-if*)
done

lemma *mult-divide-cancel-left-if2* [*simp*]:
fixes $c :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $(c*a) \,/\, c = (if\ c=0\ then\ 0\ else\ a)$
apply (*insert mult-divide-cancel-left-if* [*of c a 1*])
apply (*simp del*: *mult-divide-cancel-left-if*)
done

lemma *mult-divide-cancel-right-if1* [*simp*]:
fixes $c :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $c \,/\, (b*c) = (if\ c=0\ then\ 0\ else\ 1/b)$
apply (*insert mult-divide-cancel-right-if* [*of 1 c b*])
apply (*simp del*: *mult-divide-cancel-right-if*)
done

lemma *mult-divide-cancel-right-if2* [*simp*]:
fixes $c :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $(a*c) \,/\, c = (if\ c=0\ then\ 0\ else\ a)$
apply (*insert mult-divide-cancel-right-if* [*of a c 1*])
apply (*simp del*: *mult-divide-cancel-right-if*)
done

Two lemmas for cancelling the denominator

lemma *times-divide-self-right* [*simp*]:
fixes $a :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $a * (b/a) = (if\ a=0\ then\ 0\ else\ b)$
by (*simp add*: *times-divide-eq-right*)

lemma *times-divide-self-left* [*simp*]:
fixes $a :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $(b/a) * a = (if\ a=0\ then\ 0\ else\ b)$
by (*simp add*: *times-divide-eq-left*)

15.9 Division and Unary Minus

lemma *nonzero-minus-divide-left*: $b \neq 0 \Longrightarrow -(a/b) = (-a) \,/\, (b::'a::field)$
by (*simp add*: *divide-inverse minus-mult-left*)

lemma *nonzero-minus-divide-right*: $b \neq 0 \Longrightarrow -(a/b) = a \,/\, -(b::'a::field)$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq minus-mult-right*)

lemma *nonzero-minus-divide-divide*: $b \neq 0 \Longrightarrow (-a)/(-b) = a \;/\; (b::'a::field)$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *minus-divide-left*: $-(a/b) = (-a) \;/\; (b::'a::field)$
by (*simp add*: *divide-inverse minus-mult-left* [*symmetric*])

lemma *minus-divide-right*: $-(a/b) = a \;/\; -(b::'a::\{field,division\text{-}by\text{-}zero\})$
by (*simp add*: *divide-inverse minus-mult-right* [*symmetric*])

The effect is to extract signs from divisions

lemmas *divide-minus-left* = *minus-divide-left* [*symmetric*]
lemmas *divide-minus-right* = *minus-divide-right* [*symmetric*]
declare *divide-minus-left* [*simp*] *divide-minus-right* [*simp*]

Also, extract signs from products

lemmas *mult-minus-left* = *minus-mult-left* [*symmetric*]
lemmas *mult-minus-right* = *minus-mult-right* [*symmetric*]
declare *mult-minus-left* [*simp*] *mult-minus-right* [*simp*]

lemma *minus-divide-divide* [*simp*]:
$(-a)/(-b) = a \;/\; (b::'a::\{field,division\text{-}by\text{-}zero\})$
apply (*case-tac b=0*, *simp*)
apply (*simp add*: *nonzero-minus-divide-divide*)
done

lemma *diff-divide-distrib*: $(a-b)/(c::'a::field) = a/c - b/c$
by (*simp add*: *diff-minus add-divide-distrib*)

lemma *diff-frac-eq*: $(y::'a::field) \neq 0 \Longrightarrow z \neq 0 \Longrightarrow$
$x \;/\; y - w \;/\; z = (x * z - w * y) \;/\; (y * z)$
apply (*subst diff-def*)+
apply (*subst minus-divide-left*)
apply (*subst add-frac-eq*)
apply *simp-all*
done

15.10 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 [*simp*]: $a \neq 0$
shows $0 < (a::'a::ordered\text{-}field)$
proof −
have $0 < inverse\ (inverse\ a)$
by (*rule positive-imp-inverse-positive*)
thus $0 < a$
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 (*case-tac* $a = 0$, *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 [*simp*]: $a \neq 0$
shows $a < (0::'a::ordered\text{-}field)$
proof −
have $inverse\ (inverse\ a) < 0$
by (*rule negative-imp-inverse-negative*)
thus $a < 0$
by (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-negative-iff-negative* [*simp*]:
(*inverse a* < *0*) = (*a* < (*0*::'*a*::{*ordered-field*,*division-by-zero*}))
apply (*case-tac a* = *0*, *simp*)
apply (*blast intro*: *inverse-negative-imp-negative negative-imp-inverse-negative*)
done

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
(*0* ≤ *inverse a*) = (*0* ≤ (*a*::'*a*::{*ordered-field*,*division-by-zero*}))
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
(*inverse a* ≤ *0*) = (*a* ≤ (*0*::'*a*::{*ordered-field*,*division-by-zero*}))
by (*simp add*: *linorder-not-less* [*symmetric*])

15.11 Anti-Monotonicity of *inverse*

lemma *less-imp-inverse-less*:
assumes *less*: *a* < *b*
and *apos*: *0* < *a*
shows *inverse b* < *inverse* (*a*::'*a*::*ordered-field*)
proof (*rule ccontr*)
assume ~ *inverse b* < *inverse a*
hence *inverse a* ≤ *inverse b*
by (*simp add*: *linorder-not-less*)
hence ~ (*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-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*]:
[|*0* < *a*; *0* < *b*|]
==> (*inverse a* < *inverse b*) = (*b* < (*a*::'*a*::*ordered-field*))
by (*blast intro*: *less-imp-inverse-less dest*: *inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
[|*a* ≤ *b*; *0* < *a*|] ==> *inverse b* ≤ *inverse* (*a*::'*a*::*ordered-field*)
by (*force simp add*: *order-le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le* [*simp*]:
[|*0* < *a*; *0* < *b*|]
==> (*inverse a* ≤ *inverse b*) = (*b* ≤ (*a*::'*a*::*ordered-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*]:
[|*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*]:
[|*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*)

15.12 Inverses and the Number One

lemma *one-less-inverse-iff*:
$(1 < inverse\ x) = (0 < x\ \&\ x < (1::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$**proof**
cases
assume $0 < x$
with *inverse-less-iff-less* [*OF zero-less-one*, *of x*]
show *?thesis* **by** *simp*
next
assume *notless*: $\sim (0 < x)$
have $\sim (1 < inverse\ x)$
proof
assume $1 < inverse\ x$
also with *notless* **have** $... \leq 0$ **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*]:
$(inverse\ x = 1) = (x = (1::'a::\{field,division\text{-}by\text{-}zero\}))$
by (*insert inverse-eq-iff-eq* [*of x 1*], *simp*)

lemma *one-le-inverse-iff*:
$(1 \leq inverse\ x) = (0 < x\ \&\ x \leq (1::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
by (*force simp add*: *order-le-less one-less-inverse-iff zero-less-one*
eq-commute [*of 1*])

lemma *inverse-less-1-iff*:
$(inverse\ x < 1) = (x \leq 0 \mid 1 < (x::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
by (*simp add*: *linorder-not-le* [*symmetric*] *one-le-inverse-iff*)

lemma *inverse-le-1-iff*:
$(inverse\ x \leq 1) = (x \leq 0 \mid 1 \leq (x::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
by (*simp add*: *linorder-not-less* [*symmetric*] *one-less-inverse-iff*)

15.13 Simplification of Inequalities Involving Literal Divisors

lemma *pos-le-divide-eq*: $0 < (c::'a::ordered\text{-}field) ==> (a \leq b/c) = (a*c \leq b)$
proof −
assume *less*: $0<c$
hence $(a \leq b/c) = (a*c \leq (b/c)*c)$
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-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-le-divide-eq*: $c < (0::'a::ordered\text{-}field) ==> (a \leq b/c) = (b \leq a*c)$
proof −
assume *less*: $c<0$
hence $(a \leq b/c) = ((b/c)*c \leq a*c)$
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-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 (*case-tac* $c=0$, *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 (*case-tac* $c=0$, *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-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-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-field,division-by-zero}))
apply (case-tac 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-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-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-field,division-by-zero}))
apply (case-tac c=0, simp)
apply (force simp add: pos-divide-less-eq neg-divide-less-eq linorder-neq-iff)
done

lemma nonzero-eq-divide-eq: c≠0 ==> ((a::'a::field) = b/c) = (a*c = b)
proof -
  assume [simp]: c≠0
  have (a = b/c) = (a*c = (b/c)*c)
    by (simp add: field-mult-cancel-right)
  also have ... = (a*c = b)
    by (simp add: divide-inverse mult-assoc)
  finally show ?thesis .
qed

lemma eq-divide-eq:
  ((a::'a::{field,division-by-zero}) = b/c) = (if c≠0 then a*c = b else a=0)
by (simp add: nonzero-eq-divide-eq)

lemma nonzero-divide-eq-eq: c≠0 ==> (b/c = (a::'a::field)) = (b = a*c)
proof -
  assume [simp]: c≠0
  have (b/c = a) = ((b/c)*c = a*c)
    by (simp add: field-mult-cancel-right)
  also have ... = (b = a*c)
    by (simp add: divide-inverse mult-assoc)
  finally show ?thesis .
qed

lemma divide-eq-eq:
  (b/c = (a::'a::{field,division-by-zero})) = (if c≠0 then b = a*c else a=0)
by (force simp add: nonzero-divide-eq-eq)

lemma divide-eq-imp: (c::'a::{division-by-zero,field}) ~= 0 ==>
    b = a * c ==> b / c = a
  by (subst divide-eq-eq, simp)

lemma eq-divide-imp: (c::'a::{division-by-zero,field}) ~= 0 ==>
    a * c = b ==> a = b / c
  by (subst eq-divide-eq, simp)

lemma frac-eq-eq: (y::'a::field) ~= 0 ==> z ~= 0 ==>
```

$(x\ /\ y = w\ /\ z) = (x * z = w * y)$
apply (*subst nonzero-eq-divide-eq*)
apply *assumption*
apply (*subst times-divide-eq-left*)
apply (*erule nonzero-divide-eq-eq*)
done

15.14 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*]:
$(a/b = 0) = (a=0 \mid b=(0::'a::\{field,division\text{-}by\text{-}zero\}))$
by (*simp add*: *divide-inverse field-mult-eq-0-iff*)

lemma *divide-pos-pos*: $0 < (x::'a::ordered\text{-}field) ==> 0 < y ==> 0 < x\ /\ y$
apply (*subst pos-less-divide-eq*)
apply *assumption*
apply *simp*
done

lemma *divide-nonneg-pos*: $0 <= (x::'a::ordered\text{-}field) ==> 0 < y ==> 0 <= x\ /\ y$
apply (*subst pos-le-divide-eq*)
apply *assumption*
apply *simp*
done

lemma *divide-neg-pos*: $(x::'a::ordered\text{-}field) < 0 ==> 0 < y ==> x\ /\ y < 0$
apply (*subst pos-divide-less-eq*)

apply *assumption*
apply *simp*
done

lemma *divide-nonpos-pos*: (*x*::'*a*::*ordered-field*) <= *0* ==>
0 < *y* ==> *x* / *y* <= *0*
apply (*subst pos-divide-le-eq*)
apply *assumption*
apply *simp*
done

lemma *divide-pos-neg*: *0* < (*x*::'*a*::*ordered-field*) ==> *y* < *0* ==> *x* / *y* < *0*
apply (*subst neg-divide-less-eq*)
apply *assumption*
apply *simp*
done

lemma *divide-nonneg-neg*: *0* <= (*x*::'*a*::*ordered-field*) ==>
y < *0* ==> *x* / *y* <= *0*
apply (*subst neg-divide-le-eq*)
apply *assumption*
apply *simp*
done

lemma *divide-neg-neg*: (*x*::'*a*::*ordered-field*) < *0* ==> *y* < *0* ==> *0* < *x* / *y*
apply (*subst neg-less-divide-eq*)
apply *assumption*
apply *simp*
done

lemma *divide-nonpos-neg*: (*x*::'*a*::*ordered-field*) <= *0* ==> *y* < *0* ==>
0 <= *x* / *y*
apply (*subst neg-le-divide-eq*)
apply *assumption*
apply *simp*
done

15.15 Cancellation Laws for Division

lemma *divide-cancel-right* [*simp*]:
(*a*/*c* = *b*/*c*) = (*c* = *0* | *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*case-tac c=0*, *simp*)
apply (*simp add*: *divide-inverse field-mult-cancel-right*)
done

lemma *divide-cancel-left* [*simp*]:
(*c*/*a* = *c*/*b*) = (*c* = *0* | *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*case-tac c=0*, *simp*)
apply (*simp add*: *divide-inverse field-mult-cancel-left*)

done

15.16 Division and the Number One

Simplify expressions equated with 1

lemma *divide-eq-1-iff* [*simp*]:
(*a*/*b* = *1*) = (*b* ≠ *0* & *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*case-tac b=0*, *simp*)
apply (*simp add*: *right-inverse-eq*)
done

lemma *one-eq-divide-iff* [*simp*]:
(*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*]:
((*0*::'*a*::{*ordered-field*,*division-by-zero*}) = *1*/*a*) = (*a* = *0*)
apply (*case-tac a=0*, *simp*)
apply (*auto simp add*: *nonzero-eq-divide-eq*)
done

lemma *one-divide-eq-0-iff* [*simp*]:
(*1*/*a* = (*0*::'*a*::{*ordered-field*,*division-by-zero*})) = (*a* = *0*)
apply (*case-tac 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*]
lemmas *divide-less-0-1-iff* = *divide-less-0-iff* [*of 1*]
lemmas *zero-le-divide-1-iff* = *zero-le-divide-iff* [*of 1*]
lemmas *divide-le-0-1-iff* = *divide-le-0-iff* [*of 1*]

declare *zero-less-divide-1-iff* [*simp*]
declare *divide-less-0-1-iff* [*simp*]
declare *zero-le-divide-1-iff* [*simp*]
declare *divide-le-0-1-iff* [*simp*]

15.17 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 (*force simp add*: *zero-less-mult-iff divide-inverse mult-strict-left-mono*
order-less-imp-not-eq order-less-imp-not-eq2
less-imp-inverse-less less-imp-inverse-less-neg)

lemma *divide-left-mono*:
[|*b* ≤ *a*; *0* ≤ *c*; *0* < *a*∗*b*|] ==> *c* / *a* ≤ *c* / (*b*::'*a*::*ordered-field*)
apply (*subgoal-tac a* ≠ *0* & *b* ≠ *0*)
prefer *2*
apply (*force simp add*: *zero-less-mult-iff order-less-imp-not-eq*)
apply (*case-tac c=0*, *simp add*: *divide-inverse*)
apply (*force simp add*: *divide-strict-left-mono order-le-less*)
done

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*)
apply (*subgoal-tac a* ≠ *0* & *b* ≠ *0*)
prefer *2*
apply (*force simp add*: *zero-less-mult-iff order-less-imp-not-eq*)
apply (*drule divide-strict-left-mono* [*of* - - −*c*])
apply (*simp-all add*: *mult-commute nonzero-minus-divide-left* [*symmetric*])
done

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1*:
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*:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(b \ / \ a \leq 1) = ((0 < a \ \& \ b \leq a) \mid (a < 0 \ \& \ a \leq b) \mid a=0)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1*:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(1 < b \ / \ a) = ((0 < a \ \& \ a < b) \mid (a < 0 \ \& \ b < a))$
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1*:
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*)

15.18 Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*]:
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*]:
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*]:
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*]:
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*]:
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*]:
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*]:

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 *eq-divide-eq-1* [*simp*]:
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*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(b \ / \ a = 1) = ((a \neq 0 \ \& \ a = b))$
by (*auto simp add*: *divide-eq-eq*)

15.19 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 (*subst pos-le-divide-eq*, *assumption*+)

lemma *mult-imp-div-pos-less*: *0* < (*y*::'*a*::*ordered-field*) ==> *x* < *z* * *y* ==>
x / *y* < *z*
by (*subst pos-divide-less-eq*, *assumption*+)

lemma *mult-imp-less-div-pos*: *0* < (*y*::'*a*::*ordered-field*) ==> *z* * *y* < *x* ==>
z < *x* / *y*
by (*subst pos-less-divide-eq*, *assumption*+)

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

lemmas times-divide-eq = times-divide-eq-right times-divide-eq-left
```

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]
```

15.20 Ordered Fields are Dense

```
lemma less-add-one: a < (a+1::'a::ordered-semidom)
proof -
  have a+0 < (a+1::'a::ordered-semidom)
    by (blast intro: zero-less-one add-strict-left-mono)
  thus ?thesis by simp
qed

lemma zero-less-two: 0 < (1+1::'a::ordered-semidom)
  by (blast intro: order-less-trans zero-less-one less-add-one)

lemma less-half-sum: a < b ==> a < (a+b) / (1+1::'a::ordered-field)
by (simp add: zero-less-two pos-less-divide-eq right-distrib)

lemma gt-half-sum: a < b ==> (a+b)/(1+1::'a::ordered-field) < b
by (simp add: zero-less-two pos-divide-less-eq right-distrib)

lemma dense: a < b ==> ∃ r::'a::ordered-field. a < r & r < b
```

by (*blast intro*!: *less-half-sum gt-half-sum*)

15.21 Absolute Value

lemma *abs-one* [*simp*]: *abs 1* = (*1*::'*a*::*ordered-idom*)
by (*simp add*: *abs-if zero-less-one* [*THEN order-less-not-sym*])

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*] *ring-eq-simps*)
{
fix *u v* :: '*a*
have *bh*: ⟦*u* = *a*; *v* = *b*⟧ ⟹
u * *v* = *pprt a* * *pprt b* + *pprt a* * *nprt b* +
nprt a * *pprt b* + *nprt a* * *nprt b*
apply (*subst prts*[*of u*], *subst prts*[*of v*])
apply (*simp add*: *left-distrib right-distrib add-ac*)
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

lemma *abs-eq-mult*:

```
  assumes (0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0)
  shows abs (a*b) = abs a * abs (b::'a::lordered-ring)
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:
        ring-eq-simps
        iff2imp[OF zero-le-iff-zero-nprt] iff2imp[OF le-zero-iff-zero-pprt]
        iff2imp[OF le-zero-iff-pprt-id] iff2imp[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: ring-eq-simps)
      apply(drule (1) mult-nonneg-nonneg[of a b],simp)
      apply(drule (1) mult-nonpos-nonpos[of a b],simp)
      done
  qed
qed

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 (*case-tac 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 (*case-tac 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

lemma *eq-minus-self-iff*: (*a* = −*a*) = (*a* = (*0*::'*a*::*ordered-idom*))
by (*force simp add*: *order-eq-iff le-minus-self-iff minus-le-self-iff*)

lemma *less-minus-self-iff*: (*a* < −*a*) = (*a* < (*0*::'*a*::*ordered-idom*))
by (*simp add*: *order-less-le le-minus-self-iff eq-minus-self-iff*)

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 minus-le-self-iff eq-minus-self-iff*)
apply (*simp add*: *le-minus-self-iff linorder-neq-iff*)
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

15.22 Miscellaneous

lemma *linprog-dual-estimate*:
assumes
$A * x \leq (b::'a::lordered\text{-}ring)$
$0 \leq y$
$abs\ (A - A') \leq \delta A$
$b \leq b'$
$abs\ (c - c') \leq \delta c$
$abs\ x \leq r$
shows
$c * x \leq y * b' + (y * \delta A + abs\ (y * A' - c') + \delta c) * r$
proof −
from *prems* **have** *1*: $y * b <= y * b'$ **by** (*simp add*: *mult-left-mono*)
from *prems* **have** *2*: $y * (A * x) <= y * b$ **by** (*simp add*: *mult-left-mono*)
have *3*: $y * (A * x) = c * x + (y * (A - A') + (y * A' - c') + (c'-c)) * x$ **by** (*simp add*: *ring-eq-simps*)
from *1 2 3* **have** *4*: $c * x + (y * (A - A') + (y * A' - c') + (c'-c)) * x <= y * b'$ **by** *simp*
have *5*: $c * x <= y * b' + abs((y * (A - A') + (y * A' - c') + (c'-c)) * x)$
by (*simp only*: *4 estimate-by-abs*)
have *6*: $abs((y * (A - A') + (y * A' - c') + (c'-c)) * x) <= abs\ (y * (A - A') + (y * A' - c') + (c'-c)) * abs\ x$
by (*simp add*: *abs-le-mult*)
have *7*: $(abs\ (y * (A - A') + (y * A' - c') + (c'-c))) * abs\ x <= (abs\ (y * (A-A') + (y*A'-c')) + abs(c'-c)) * abs\ x$
by(*rule abs-triangle-ineq* [*THEN mult-right-mono*]) *simp*
have *8*: $(abs\ (y * (A-A') + (y*A'-c')) + abs(c'-c)) * abs\ x <= (abs\ (y * (A-A')) + abs\ (y*A'-c') + abs(c'-c)) * abs\ x$
by (*simp add*: *abs-triangle-ineq mult-right-mono*)
have *9*: $(abs\ (y * (A-A')) + abs\ (y*A'-c') + abs(c'-c)) * abs\ x <= (abs\ y * abs\ (A-A') + abs\ (y*A'-c') + abs\ (c'-c)) * abs\ x$
by (*simp add*: *abs-le-mult mult-right-mono*)
have *10*: $c'-c = -(c-c')$ **by** (*simp add*: *ring-eq-simps*)
have *11*: $abs\ (c'-c) = abs\ (c-c')$
by (*subst 10*, *subst abs-minus-cancel*, *simp*)
have *12*: $(abs\ y * abs\ (A-A') + abs\ (y*A'-c') + abs\ (c'-c)) * abs\ x <= (abs\ y * abs\ (A-A') + abs\ (y*A'-c') + \delta c) * abs\ x$
by (*simp add*: *11 prems mult-right-mono*)
have *13*: $(abs\ y * abs\ (A-A') + abs\ (y*A'-c') + \delta c) * abs\ x <= (abs\ y * \delta A + abs\ (y*A'-c') + \delta c) * abs\ x$
by (*simp add*: *prems mult-right-mono mult-left-mono*)
have *r*: $(abs\ y * \delta A + abs\ (y*A'-c') + \delta c) * abs\ x <= (abs\ y * \delta A + abs\ (y*A'-c') + \delta c) * r$

apply (*rule mult-left-mono*)
apply (*simp add*: *prems*)
apply (*rule-tac add-mono*[*of 0*::'*a* - *0*, *simplified*])+
apply (*rule mult-left-mono*[*of 0* δA, *simplified*])
apply (*simp-all*)
apply (*rule order-trans*[**where** *y*=*abs* ($A-A'$)], *simp-all add*: *prems*)
apply (*rule order-trans*[**where** *y*=*abs* ($c-c'$)], *simp-all add*: *prems*)
done
from *6 7 8 9 12 13 r* **have** *14*: *abs*(($y * (A - A') + (y * A' - c') + (c'-c)) * x$) <=(*abs y* * δA + *abs* ($y*A'-c'$) + δc) * *r*
by (*simp*)
show *?thesis*
apply (*rule-tac le-add-right-mono*[*of* - - *abs*(($y * (A - A') + (y * A' - c') + (c'-c)) * x$)])
apply (*simp-all only*: *5 14*[*simplified abs-of-nonneg*[*of y*, *simplified prems*]])
done
qed

lemma *le-ge-imp-abs-diff-1*:
assumes
A1 <= (*A*::'*a*::*lordered-ring*)
A <= *A2*
shows *abs* ($A-A1$) <= $A2-A1$
proof −
have *0* <= *A* − *A1*
proof −
have *1*: *A* − *A1* = *A* + (− *A1*) **by** *simp*
show *?thesis* **by** (*simp only*: *1 add-right-mono*[*of A1 A* $-A1$, *simplified*, *simplified prems*])
qed
then have *abs* ($A-A1$) = $A-A1$ **by** (*rule abs-of-nonneg*)
with *prems* **show** *abs* ($A-A1$) <= ($A2-A1$) **by** *simp*
qed

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: ring-eq-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-le-dual-prts:
  assumes
  A * x ≤ (b::'a::lordered-ring)
  0 ≤ y
  A1 ≤ A
  A ≤ A2
  c1 ≤ c
  c ≤ c2
  r1 ≤ x
  x ≤ r2
  shows
  c * x ≤ y * b + (let s1 = c1 - y * A2; s2 = c2 - y * A1 in pprt s2 * pprt r2
+ pprt s1 * nprt r2 + nprt s2 * pprt r1 + nprt s1 * nprt r1)
  (is - <= - + ?C)
```

proof −
from *prems* **have** $y * (A * x) <= y * b$ **by** (*simp add*: *mult-left-mono*)
moreover have $y * (A * x) = c * x + (y * A - c) * x$ **by** (*simp add*: *ring-eq-simps*)
ultimately have $c * x + (y * A - c) * x <= y * b$ **by** *simp*
then have $c * x <= y * b - (y * A - c) * x$ **by** (*simp add*: *le-diff-eq*)
then have *cx*: $c * x <= y * b + (c - y * A) * x$ **by** (*simp add*: *ring-eq-simps*)
have *s2*: $c - y * A <= c2 - y * A1$
by (*simp add*: *diff-def prems add-mono mult-left-mono*)
have *s1*: $c1 - y * A2 <= c - y * A$
by (*simp add*: *diff-def prems add-mono mult-left-mono*)
have *prts*: $(c - y * A) * x <= ?C$
apply (*simp add*: *Let-def*)
apply (*rule mult-le-prts*)
apply (*simp-all add*: *prems s1 s2*)
done
then have $y * b + (c - y * A) * x <= y * b + ?C$
by *simp*
with *cx* **show** *?thesis*
by(*simp only*:)
qed

ML ⟪
val left-distrib = thm left-distrib;
val right-distrib = thm right-distrib;
val mult-commute = thm mult-commute;
val distrib = thm distrib;
val zero-neq-one = thm zero-neq-one;
val no-zero-divisors = thm no-zero-divisors;
val left-inverse = thm left-inverse;
val divide-inverse = thm divide-inverse;
val mult-zero-left = thm mult-zero-left;
val mult-zero-right = thm mult-zero-right;
val field-mult-eq-0-iff = thm field-mult-eq-0-iff;
val inverse-zero = thm inverse-zero;
val ring-distrib = thms ring-distrib;
val combine-common-factor = thm combine-common-factor;
val minus-mult-left = thm minus-mult-left;
val minus-mult-right = thm minus-mult-right;
val minus-mult-minus = thm minus-mult-minus;
val minus-mult-commute = thm minus-mult-commute;
val right-diff-distrib = thm right-diff-distrib;
val left-diff-distrib = thm left-diff-distrib;
val mult-left-mono = thm mult-left-mono;
val mult-right-mono = thm mult-right-mono;
val mult-strict-left-mono = thm mult-strict-left-mono;
val mult-strict-right-mono = thm mult-strict-right-mono;
val mult-mono = thm mult-mono;
val mult-strict-mono = thm mult-strict-mono;

val abs-if = thm abs-if;
val zero-less-one = thm zero-less-one;
val eq-add-iff1 = thm eq-add-iff1;
val eq-add-iff2 = thm eq-add-iff2;
val less-add-iff1 = thm less-add-iff1;
val less-add-iff2 = thm less-add-iff2;
val le-add-iff1 = thm le-add-iff1;
val le-add-iff2 = thm le-add-iff2;
val mult-left-le-imp-le = thm mult-left-le-imp-le;
val mult-right-le-imp-le = thm mult-right-le-imp-le;
val mult-left-less-imp-less = thm mult-left-less-imp-less;
val mult-right-less-imp-less = thm mult-right-less-imp-less;
val mult-strict-left-mono-neg = thm mult-strict-left-mono-neg;
val mult-left-mono-neg = thm mult-left-mono-neg;
val mult-strict-right-mono-neg = thm mult-strict-right-mono-neg;
val mult-right-mono-neg = thm mult-right-mono-neg;
(*
val mult-pos = thm mult-pos;
val mult-pos-le = thm mult-pos-le;
val mult-pos-neg = thm mult-pos-neg;
val mult-pos-neg-le = thm mult-pos-neg-le;
val mult-pos-neg2 = thm mult-pos-neg2;
val mult-pos-neg2-le = thm mult-pos-neg2-le;
val mult-neg = thm mult-neg;
val mult-neg-le = thm mult-neg-le;
*)
val zero-less-mult-pos = thm zero-less-mult-pos;
val zero-less-mult-pos2 = thm zero-less-mult-pos2;
val zero-less-mult-iff = thm zero-less-mult-iff;
val mult-eq-0-iff = thm mult-eq-0-iff;
val zero-le-mult-iff = thm zero-le-mult-iff;
val mult-less-0-iff = thm mult-less-0-iff;
val mult-le-0-iff = thm mult-le-0-iff;
val split-mult-pos-le = thm split-mult-pos-le;
val split-mult-neg-le = thm split-mult-neg-le;
val zero-le-square = thm zero-le-square;
val zero-le-one = thm zero-le-one;
val not-one-le-zero = thm not-one-le-zero;
val not-one-less-zero = thm not-one-less-zero;
val mult-left-mono-neg = thm mult-left-mono-neg;
val mult-right-mono-neg = thm mult-right-mono-neg;
val mult-strict-mono = thm mult-strict-mono;
val mult-strict-mono' = thm mult-strict-mono';
val mult-mono = thm mult-mono;
val less-1-mult = thm less-1-mult;
val mult-less-cancel-right-disj = thm mult-less-cancel-right-disj;
val mult-less-cancel-left-disj = thm mult-less-cancel-left-disj;
val mult-less-cancel-right = thm mult-less-cancel-right;
val mult-less-cancel-left = thm mult-less-cancel-left;

val mult-le-cancel-right = thm mult-le-cancel-right;
val mult-le-cancel-left = thm mult-le-cancel-left;
val mult-less-imp-less-left = thm mult-less-imp-less-left;
val mult-less-imp-less-right = thm mult-less-imp-less-right;
val mult-cancel-right = thm mult-cancel-right;
val mult-cancel-left = thm mult-cancel-left;
val ring-eq-simps = thms ring-eq-simps;
val right-inverse = thm right-inverse;
val right-inverse-eq = thm right-inverse-eq;
val nonzero-inverse-eq-divide = thm nonzero-inverse-eq-divide;
val divide-self = thm divide-self;
val divide-zero = thm divide-zero;
val divide-zero-left = thm divide-zero-left;
val inverse-eq-divide = thm inverse-eq-divide;
val add-divide-distrib = thm add-divide-distrib;
val field-mult-eq-0-iff = thm field-mult-eq-0-iff;
val field-mult-cancel-right-lemma = thm field-mult-cancel-right-lemma;
val field-mult-cancel-right = thm field-mult-cancel-right;
val field-mult-cancel-left = thm field-mult-cancel-left;
val nonzero-imp-inverse-nonzero = thm nonzero-imp-inverse-nonzero;
val inverse-zero-imp-zero = thm inverse-zero-imp-zero;
val inverse-nonzero-imp-nonzero = thm inverse-nonzero-imp-nonzero;
val inverse-nonzero-iff-nonzero = thm inverse-nonzero-iff-nonzero;
val nonzero-inverse-minus-eq = thm nonzero-inverse-minus-eq;
val inverse-minus-eq = thm inverse-minus-eq;
val nonzero-inverse-eq-imp-eq = thm nonzero-inverse-eq-imp-eq;
val inverse-eq-imp-eq = thm inverse-eq-imp-eq;
val inverse-eq-iff-eq = thm inverse-eq-iff-eq;
val nonzero-inverse-inverse-eq = thm nonzero-inverse-inverse-eq;
val inverse-inverse-eq = thm inverse-inverse-eq;
val inverse-1 = thm inverse-1;
val nonzero-inverse-mult-distrib = thm nonzero-inverse-mult-distrib;
val inverse-mult-distrib = thm inverse-mult-distrib;
val inverse-add = thm inverse-add;
val inverse-divide = thm inverse-divide;
val nonzero-mult-divide-cancel-left = thm nonzero-mult-divide-cancel-left;
val mult-divide-cancel-left = thm mult-divide-cancel-left;
val nonzero-mult-divide-cancel-right = thm nonzero-mult-divide-cancel-right;
val mult-divide-cancel-right = thm mult-divide-cancel-right;
val mult-divide-cancel-eq-if = thm mult-divide-cancel-eq-if;
val divide-1 = thm divide-1;
val times-divide-eq-right = thm times-divide-eq-right;
val times-divide-eq-left = thm times-divide-eq-left;
val divide-divide-eq-right = thm divide-divide-eq-right;
val divide-divide-eq-left = thm divide-divide-eq-left;
val nonzero-minus-divide-left = thm nonzero-minus-divide-left;
val nonzero-minus-divide-right = thm nonzero-minus-divide-right;
val nonzero-minus-divide-divide = thm nonzero-minus-divide-divide;
val minus-divide-left = thm minus-divide-left;

val minus-divide-right = thm minus-divide-right;
val minus-divide-divide = thm minus-divide-divide;
val diff-divide-distrib = thm diff-divide-distrib;
val positive-imp-inverse-positive = thm positive-imp-inverse-positive;
val negative-imp-inverse-negative = thm negative-imp-inverse-negative;
val inverse-le-imp-le = thm inverse-le-imp-le;
val inverse-positive-imp-positive = thm inverse-positive-imp-positive;
val inverse-positive-iff-positive = thm inverse-positive-iff-positive;
val inverse-negative-imp-negative = thm inverse-negative-imp-negative;
val inverse-negative-iff-negative = thm inverse-negative-iff-negative;
val inverse-nonnegative-iff-nonnegative = thm inverse-nonnegative-iff-nonnegative;
val inverse-nonpositive-iff-nonpositive = thm inverse-nonpositive-iff-nonpositive;
val less-imp-inverse-less = thm less-imp-inverse-less;
val inverse-less-imp-less = thm inverse-less-imp-less;
val inverse-less-iff-less = thm inverse-less-iff-less;
val le-imp-inverse-le = thm le-imp-inverse-le;
val inverse-le-iff-le = thm inverse-le-iff-le;
val inverse-le-imp-le-neg = thm inverse-le-imp-le-neg;
val less-imp-inverse-less-neg = thm less-imp-inverse-less-neg;
val inverse-less-imp-less-neg = thm inverse-less-imp-less-neg;
val inverse-less-iff-less-neg = thm inverse-less-iff-less-neg;
val le-imp-inverse-le-neg = thm le-imp-inverse-le-neg;
val inverse-le-iff-le-neg = thm inverse-le-iff-le-neg;
val one-less-inverse-iff = thm one-less-inverse-iff;
val inverse-eq-1-iff = thm inverse-eq-1-iff;
val one-le-inverse-iff = thm one-le-inverse-iff;
val inverse-less-1-iff = thm inverse-less-1-iff;
val inverse-le-1-iff = thm inverse-le-1-iff;
val zero-less-divide-iff = thm zero-less-divide-iff;
val divide-less-0-iff = thm divide-less-0-iff;
val zero-le-divide-iff = thm zero-le-divide-iff;
val divide-le-0-iff = thm divide-le-0-iff;
val divide-eq-0-iff = thm divide-eq-0-iff;
val pos-le-divide-eq = thm pos-le-divide-eq;
val neg-le-divide-eq = thm neg-le-divide-eq;
val le-divide-eq = thm le-divide-eq;
val pos-divide-le-eq = thm pos-divide-le-eq;
val neg-divide-le-eq = thm neg-divide-le-eq;
val divide-le-eq = thm divide-le-eq;
val pos-less-divide-eq = thm pos-less-divide-eq;
val neg-less-divide-eq = thm neg-less-divide-eq;
val less-divide-eq = thm less-divide-eq;
val pos-divide-less-eq = thm pos-divide-less-eq;
val neg-divide-less-eq = thm neg-divide-less-eq;
val divide-less-eq = thm divide-less-eq;
val nonzero-eq-divide-eq = thm nonzero-eq-divide-eq;
val eq-divide-eq = thm eq-divide-eq;
val nonzero-divide-eq-eq = thm nonzero-divide-eq-eq;
val divide-eq-eq = thm divide-eq-eq;

val divide-cancel-right = thm divide-cancel-right;
val divide-cancel-left = thm divide-cancel-left;
val divide-eq-1-iff = thm divide-eq-1-iff;
val one-eq-divide-iff = thm one-eq-divide-iff;
val zero-eq-1-divide-iff = thm zero-eq-1-divide-iff;
val one-divide-eq-0-iff = thm one-divide-eq-0-iff;
val divide-strict-right-mono = thm divide-strict-right-mono;
val divide-right-mono = thm divide-right-mono;
val divide-strict-left-mono = thm divide-strict-left-mono;
val divide-left-mono = thm divide-left-mono;
val divide-strict-left-mono-neg = thm divide-strict-left-mono-neg;
val divide-strict-right-mono-neg = thm divide-strict-right-mono-neg;
val less-add-one = thm less-add-one;
val zero-less-two = thm zero-less-two;
val less-half-sum = thm less-half-sum;
val gt-half-sum = thm gt-half-sum;
val dense = thm dense;
val abs-one = thm abs-one;
val abs-le-mult = thm abs-le-mult;
val abs-eq-mult = thm abs-eq-mult;
val abs-mult = thm abs-mult;
val abs-mult-self = thm abs-mult-self;
val nonzero-abs-inverse = thm nonzero-abs-inverse;
val abs-inverse = thm abs-inverse;
val nonzero-abs-divide = thm nonzero-abs-divide;
val abs-divide = thm abs-divide;
val abs-mult-less = thm abs-mult-less;
val eq-minus-self-iff = thm eq-minus-self-iff;
val less-minus-self-iff = thm less-minus-self-iff;
val abs-less-iff = thm abs-less-iff;
⟩⟩

end

16 Nat: Natural numbers

theory *Nat*
imports *Wellfounded-Recursion Ring-and-Field*
begin

16.1 Type *ind*

typedecl *ind*

consts
Zero-Rep :: *ind*
Suc-Rep :: *ind* => *ind*

axioms
— the axiom of infinity in 2 parts
inj-Suc-Rep: *inj Suc-Rep*
Suc-Rep-not-Zero-Rep: *Suc-Rep x* $\neq$ *Zero-Rep*
finalconsts
Zero-Rep
Suc-Rep

16.2 Type nat

Type definition

consts
Nat :: *ind set*

inductive *Nat*
intros
Zero-RepI: *Zero-Rep* : *Nat*
Suc-RepI: *i* : *Nat* ==> *Suc-Rep i* : *Nat*

global

typedef (**open** *Nat*)
nat = *Nat* **by** (*rule exI*, *rule Nat.Zero-RepI*)

instance *nat* :: {*ord*, *zero*, *one*} **..**

Abstract constants and syntax

consts
Suc :: *nat* => *nat*
pred-nat :: (*nat* * *nat*) *set*

local

defs
Zero-nat-def: *0* == *Abs-Nat Zero-Rep*
Suc-def: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat n*)))
One-nat-def [*simp*]: *1* == *Suc 0*

— nat operations
pred-nat-def: *pred-nat* == {(*m*, *n*). *n* = *Suc m*}

less-def: *m* < *n* == (*m*, *n*) : *trancl pred-nat*

le-def: *m* $\leq$ (*n*::*nat*) == ~ (*n* < *m*)

Induction

theorem *nat-induct*: *P 0* ==> (!!*n*. *P n* ==> *P* (*Suc n*)) ==> *P n*
apply (*unfold Zero-nat-def Suc-def*)

apply (*rule Rep-Nat-inverse* [*THEN subst*]) — types force good instantiation
apply (*erule Rep-Nat* [*THEN Nat.induct*])
apply (*iprover elim*: *Abs-Nat-inverse* [*THEN subst*])
done

Distinctness of constructors

lemma *Suc-not-Zero* [*iff*]: *Suc m* ≠ *0*
by (*simp add*: *Zero-nat-def Suc-def Abs-Nat-inject Rep-Nat Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep*)

lemma *Zero-not-Suc* [*iff*]: *0* ≠ *Suc m*
by (*rule not-sym*, *rule Suc-not-Zero not-sym*)

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*)

Injectiveness of *Suc*

lemma *inj-Suc*[*simp*]: *inj-on Suc N*
by (*simp add*: *Suc-def inj-on-def Abs-Nat-inject Rep-Nat Suc-RepI inj-Suc-Rep* [*THEN inj-eq*] *Rep-Nat-inject*)

lemma *Suc-inject*: *Suc x* = *Suc y* ==> *x* = *y*
by (*rule inj-Suc* [*THEN injD*])

lemma *Suc-Suc-eq* [*iff*]: (*Suc m* = *Suc n*) = (*m* = *n*)
by (*rule inj-Suc* [*THEN inj-eq*])

lemma *nat-not-singleton*: (∀ *x*. *x* = (*0*::*nat*)) = *False*
by *auto*

nat is a datatype

rep-datatype *nat*
distinct *Suc-not-Zero Zero-not-Suc*
inject *Suc-Suc-eq*
induction *nat-induct*

lemma *n-not-Suc-n*: *n* ≠ *Suc n*
by (*induct n*) *simp-all*

lemma *Suc-n-not-n*: *Suc t* ≠ *t*
by (*rule not-sym*, *rule n-not-Suc-n*)

A special form of induction for reasoning about *m* < *n* and *m* − *n*

theorem *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

16.3 Basic properties of "less than"

lemma *wf-pred-nat*: *wf pred-nat*
apply (*unfold wf-def pred-nat-def*, *clarify*)
apply (*induct-tac x*, *blast*+)
done

lemma *wf-less*: *wf* $\{(x, y::nat).\ x < y\}$
apply (*unfold less-def*)
apply (*rule wf-pred-nat* [*THEN wf-trancl*, *THEN wf-subset*], *blast*)
done

lemma *less-eq*: $((m, n) : pred\text{-}nat\hat{}+) = (m < n)$
apply (*unfold less-def*)
apply (*rule refl*)
done

16.3.1 Introduction properties

lemma *less-trans*: $i < j \Longrightarrow j < k \Longrightarrow i < (k::nat)$
apply (*unfold less-def*)
apply (*rule trans-trancl* [*THEN transD*], *assumption*+)
done

lemma *lessI* [*iff*]: $n < Suc\ n$
apply (*unfold less-def pred-nat-def*)
apply (*simp add*: *r-into-trancl*)
done

lemma *less-SucI*: $i < j \Longrightarrow i < Suc\ j$
apply (*rule less-trans*, *assumption*)
apply (*rule lessI*)
done

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
apply (*induct n*)
apply (*rule lessI*)
apply (*erule less-trans*)
apply (*rule lessI*)
done

16.3.2 Elimination properties

lemma *less-not-sym*: $n < m \Longrightarrow \sim m < (n::nat)$
apply (*unfold less-def*)
apply (*blast intro*: *wf-pred-nat wf-trancl* [*THEN wf-asym*])
done

lemma *less-asym*:
assumes *h1*: $(n::nat) < m$ **and** *h2*: $\sim P \Longrightarrow m < n$ **shows** P
apply (*rule contrapos-np*)
apply (*rule less-not-sym*)
apply (*rule h1*)
apply (*erule h2*)
done

lemma *less-not-refl*: $\sim n < (n::nat)$
apply (*unfold less-def*)
apply (*rule wf-pred-nat* [*THEN wf-trancl*, *THEN wf-not-refl*])
done

lemma *less-irrefl* [*elim*!]: $(n::nat) < n \Longrightarrow R$
by (*rule notE*, *rule less-not-refl*)

lemma *less-not-refl2*: $n < m \Longrightarrow m \neq (n::nat)$ **by** *blast*

lemma *less-not-refl3*: $(s::nat) < t \Longrightarrow s \neq t$
by (*rule not-sym*, *rule less-not-refl2*)

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
apply (*rule major* [*unfolded less-def pred-nat-def*, *THEN tranclE*], *simp-all*)
apply (*erule p1*)
apply (*rule p2*)
apply (*simp add*: *less-def pred-nat-def*, *assumption*)
done

lemma *not-less0* [*iff*]: $\sim n < (0::nat)$
by (*blast elim*: *lessE*)

lemma *less-zeroE*: $(n::nat) < 0 \Longrightarrow R$
by (*rule notE*, *rule not-less0*)

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 *less-Suc-eq*: $(m < Suc\ n) = (m < n \mid m = n)$
by (*blast elim*!: *less-SucE intro*: *less-trans*)

lemma *less-one* [*iff*]: $(n < (1::nat)) = (n = 0)$
by (*simp add*: *less-Suc-eq*)

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
by (*simp add*: *less-Suc-eq*)

lemma *Suc-mono*: $m < n \Longrightarrow Suc\ m < Suc\ n$
by (*induct n*) (*fast elim*: *less-trans lessE*)+

"Less than" is a linear ordering

lemma *less-linear*: $m < n \mid m = n \mid n < (m::nat)$
apply (*induct m*)
apply (*induct n*)
apply (*rule refl* [*THEN disjI1*, *THEN disjI2*])
apply (*rule zero-less-Suc* [*THEN disjI1*])
apply (*blast intro*: *Suc-mono less-SucI elim*: *lessE*)
done

"Less than" is antisymmetric, sort of

lemma *less-antisym*: $[\![\neg\ n < m;\ n < Suc\ m\ ]\!] \Longrightarrow m = n$
apply(*simp only*:*less-Suc-eq*)
apply *blast*
done

lemma *nat-neq-iff*: $((m::nat) \neq n) = (m < n \mid n < m)$
using *less-linear* **by** *blast*

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

16.3.3 Inductive (?) properties

lemma *Suc-lessI*: $m < n \Longrightarrow Suc\ m \neq n \Longrightarrow Suc\ m < n$
apply (*simp add*: *nat-neq-iff*)
apply (*blast elim*!: *less-irrefl less-SucE elim*: *less-asym*)
done

lemma *Suc-lessD*: $Suc\ m < n \Longrightarrow m < n$

apply (*induct n*)
apply (*fast intro*!: *lessI* [*THEN less-SucI*] *elim*: *less-trans lessE*)+
done

lemma *Suc-lessE*: **assumes** *major*: *Suc i* < *k*
and *minor*: !!*j*. *i* < *j* ==> *k* = *Suc j* ==> *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* ==> *m* < *n*
by (*blast elim*: *lessE dest*: *Suc-lessD*)

lemma *Suc-less-eq* [*iff*, *code*]: (*Suc m* < *Suc n*) = (*m* < *n*)
apply (*rule iffI*)
apply (*erule Suc-less-SucD*)
apply (*erule Suc-mono*)
done

lemma *less-trans-Suc*:
assumes *le*: *i* < *j* **shows** *j* < *k* ==> *Suc i* < *k*
apply (*induct k*, *simp-all*)
apply (*insert le*)
apply (*simp add*: *less-Suc-eq*)
apply (*blast dest*: *Suc-lessD*)
done

lemma [*code*]: ((*n*::*nat*) < *0*) = *False* **by** *simp*
lemma [*code*]: (*0* < *Suc n*) = *True* **by** *simp*

Can be used with *less-Suc-eq* to get *n* = *m* ∨ *n* < *m*

lemma *not-less-eq*: (~ *m* < *n*) = (*n* < *Suc m*)
by (*rule-tac m* = *m* **and** *n* = *n* **in** *diff-induct*, *simp-all*)

Complete induction, aka course-of-values induction

lemma *nat-less-induct*:
assumes *prem*: !!*n*. ∀ *m*::*nat*. *m* < *n* −−> *P m* ==> *P n* **shows** *P n*
apply (*rule-tac a*=*n* **in** *wf-induct*)
apply (*rule wf-pred-nat* [*THEN wf-trancl*])
apply (*rule prem*)
apply (*unfold less-def*, *assumption*)
done

lemmas *less-induct* = *nat-less-induct* [*rule-format*, *case-names less*]

16.4 Properties of "less than or equal"

Was *le-eq-less-Suc*, but this orientation is more useful

lemma *less-Suc-eq-le*: $(m < Suc\ n) = (m \leq n)$
by (*unfold le-def*, *rule not-less-eq* [*symmetric*])

lemma *le-imp-less-Suc*: $m \leq n \Longrightarrow m < Suc\ n$
by (*rule less-Suc-eq-le* [*THEN iffD2*])

lemma *le0* [*iff*]: $(0::nat) \leq n$
by (*unfold le-def*, *rule not-less0*)

lemma *Suc-n-not-le-n*: $\sim Suc\ n \leq n$
by (*simp add*: *le-def*)

lemma *le-0-eq* [*iff*]: $((i::nat) \leq 0) = (i = 0)$
by (*induct i*) (*simp-all add*: *le-def*)

lemma *le-Suc-eq*: $(m \leq Suc\ n) = (m \leq n \mid m = Suc\ n)$
by (*simp del*: *less-Suc-eq-le 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*: $m < n \Longrightarrow Suc(m) \leq n$
apply (*simp add*: *le-def less-Suc-eq*)
apply (*blast elim!*: *less-irrefl less-asym*)
done — formerly called lessD

lemma *Suc-leD*: $Suc(m) \leq n \Longrightarrow m \leq n$
by (*simp add*: *le-def less-Suc-eq*)

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: $Suc\ m \leq n \Longrightarrow m < n$
apply (*simp add*: *le-def less-Suc-eq*)
using *less-linear*
apply *blast*
done

lemma *Suc-le-eq*: $(Suc\ m \leq n) = (m < n)$
by (*blast intro*: *Suc-leI Suc-le-lessD*)

lemma *le-SucI*: $m \leq n \Longrightarrow m \leq Suc\ n$
by (*unfold le-def*) (*blast dest*: *Suc-lessD*)

lemma *less-imp-le*: $m < n \Longrightarrow m \leq (n::nat)$
by (*unfold le-def*) (*blast elim*: *less-asym*)

For instance, $(Suc\ m < Suc\ n) = (Suc\ m \leq n) = (m < n)$

lemmas *le-simps* = *less-imp-le less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *le-imp-less-or-eq*: $m \leq n \Longrightarrow m < n \mid m = (n::nat)$
apply (*unfold le-def*)
using *less-linear*
apply (*blast elim*: *less-irrefl less-asym*)
done

lemma *less-or-eq-imp-le*: $m < n \mid m = n \Longrightarrow m \leq (n::nat)$
apply (*unfold le-def*)
using *less-linear*
apply (*blast elim*!: *less-irrefl elim*: *less-asym*)
done

lemma *le-eq-less-or-eq*: $(m \leq (n::nat)) = (m < n \mid m=n)$
by (*iprover intro*: *less-or-eq-imp-le le-imp-less-or-eq*)

Useful with *Blast*.

lemma *eq-imp-le*: $(m::nat) = n \Longrightarrow m \leq n$
by (*rule less-or-eq-imp-le*, *rule disjI2*)

lemma *le-refl*: $n \leq (n::nat)$
by (*simp add*: *le-eq-less-or-eq*)

lemma *le-less-trans*: $[\!| i \leq j; j < k |\!] \Longrightarrow i < (k::nat)$
by (*blast dest*!: *le-imp-less-or-eq intro*: *less-trans*)

lemma *less-le-trans*: $[\!| i < j; j \leq k |\!] \Longrightarrow i < (k::nat)$
by (*blast dest*!: *le-imp-less-or-eq intro*: *less-trans*)

lemma *le-trans*: $[\!| i \leq j; j \leq k |\!] \Longrightarrow i \leq (k::nat)$
by (*blast dest*!: *le-imp-less-or-eq intro*: *less-or-eq-imp-le less-trans*)

lemma *le-anti-sym*: $[\!| m \leq n; n \leq m |\!] \Longrightarrow m = (n::nat)$
by (*blast dest*!: *le-imp-less-or-eq elim*!: *less-irrefl elim*: *less-asym*)

lemma *Suc-le-mono* [*iff*]: $(Suc\ n \leq Suc\ m) = (n \leq m)$
by (*simp add*: *le-simps*)

Axiom *order-less-le* of class *order*:

lemma *nat-less-le*: $((m::nat) < n) = (m \leq n \ \&\ m \neq n)$
by (*simp add*: *le-def nat-neq-iff*) (*blast elim*!: *less-asym*)

lemma *le-neq-implies-less*: $(m::nat) \leq n \Longrightarrow m \neq n \Longrightarrow m < n$
by (*rule iffD2*, *rule nat-less-le*, *rule conjI*)

Axiom *linorder-linear* of class *linorder*:

lemma *nat-le-linear*: $(m::nat) \leq n \mid n \leq m$
apply (*simp add*: *le-eq-less-or-eq*)
using *less-linear*
apply *blast*

done

Type @typ nat is a wellfounded linear order

instance *nat* :: {*order*, *linorder*, *wellorder*}
by *intro-classes*
(*assumption* |
rule le-refl le-trans le-anti-sym nat-less-le nat-le-linear wf-less)+

lemmas *linorder-neqE-nat* = *linorder-neqE*[**where** *'a* = *nat*]

lemma *not-less-less-Suc-eq*: ~ *n* < *m* ==> (*n* < *Suc m*) = (*n* = *m*)
by (*blast elim*!: *less-SucE*)

Rewrite $n < Suc\ m$ to $n = m$ if $\neg\ n < m$ or $m \leq n$ hold. Not suitable as default simprules because they often lead to looping

lemma *le-less-Suc-eq*: *m* ≤ *n* ==> (*n* < *Suc m*) = (*n* = *m*)
by (*rule not-less-less-Suc-eq*, *rule leD*)

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

Re-orientation of the equations $0 = x$ and $1 = x$. No longer added as simprules (they loop) but via *reorient-simproc* in Bin

Polymorphic, not just for *nat*

lemma *zero-reorient*: (*0* = *x*) = (*x* = *0*)
by *auto*

lemma *one-reorient*: (*1* = *x*) = (*x* = *1*)
by *auto*

16.5 Arithmetic operators

axclass *power* < *type*

consts
power :: (*'a*::*power*) => *nat* => *'a* (**infixr** ^ *80*)

arithmetic operators + − and ∗

instance *nat* :: {*plus*, *minus*, *times*, *power*} **..**

size of a datatype value; overloaded

consts *size* :: *'a* => *nat*

primrec
add-0: *0* + *n* = *n*
add-Suc: *Suc m* + *n* = *Suc* (*m* + *n*)

primrec

diff-0: $m - 0 = m$
diff-Suc: $m - Suc\ n = (case\ m - n\ of\ 0 => 0 \mid Suc\ k => k)$

primrec
mult-0: $0 * n = 0$
mult-Suc: $Suc\ m * n = n + (m * n)$

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 (*case-tac n*) *simp-all*

lemma *gr-implies-not0*: $!!n{::}nat.\ m<n ==> n \neq 0$
by (*case-tac n*) *simp-all*

lemma *neq0-conv* [*iff*]: $!!n{::}nat.\ (n \neq 0) = (0 < n)$
by (*case-tac n*) *simp-all*

This theorem is useful with *blast*

lemma *gr0I*: $((n{::}nat) = 0 ==> False) ==> 0 < n$
by (*rule iffD1*, *rule neq0-conv*, *iprover*)

lemma *gr0-conv-Suc*: $(0 < n) = (\exists m.\ n = Suc\ m)$
by (*fast intro*: *not0-implies-Suc*)

lemma *not-gr0* [*iff*]: $!!n{::}nat.\ (\sim (0 < n)) = (n = 0)$
apply (*rule iffI*)
apply (*rule ccontr*, *simp-all*)
done

lemma *Suc-le-D*: $(Suc\ n \leq m') ==> (?\ m.\ m' = Suc\ m)$
by (*induct m'*) *simp-all*

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $(m < Suc\ n) = (m = 0 \mid (\exists j.\ m = Suc\ j\ \&\ j < n))$
by (*case-tac m*) *simp-all*

lemma *nat-induct2*: $[|P\ 0;\ P\ (Suc\ 0);\ !!k.\ P\ k ==> P\ (Suc\ (Suc\ k))|] ==> P\ n$
apply (*rule nat-less-induct*)
apply (*case-tac n*)
apply (*case-tac* [*2*] *nat*)
apply (*blast intro*: *less-trans*)+
done

16.6 *LEAST* theorems for type *nat*

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*)
by (*erule* (*1*) *Least-Suc* [*THEN ssubst*], *simp*)

16.7 *min* and *max*

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*: *min-of-mono*)

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*: *max-of-mono*)

16.8 Basic rewrite rules for the arithmetic operators

Difference

lemma *diff-0-eq-0* [*simp*, *code*]: *0* − *n* = (*0*::*nat*)
by (*induct n*) *simp-all*

lemma *diff-Suc-Suc* [*simp*, *code*]: *Suc*(*m*) − *Suc*(*n*) = *m* − *n*
by (*induct n*) *simp-all*

Could be (and is, below) generalized in various ways However, none of the generalizations are currently in the simpset, and I dread to think what happens if I put them in

lemma *Suc-pred* [*simp*]: *0 < n ==> Suc (n − Suc 0) = n*
by (*simp split add*: *nat.split*)

declare *diff-Suc* [*simp del*, *code del*]

16.9 Addition

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*

lemma [*code*]: *Suc m + n = m + Suc n* **by** *simp*

Associative law for addition

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

Commutative law for addition

lemma *nat-add-commute*: *m + n = n + (m*::*nat)*
by (*induct m*) *simp-all*

lemma *nat-add-left-commute*: *x + (y + z) = y + ((x + z)*::*nat)*
apply (*rule mk-left-commute* [*of op +*])
apply (*rule nat-add-assoc*)
apply (*rule nat-add-commute*)
done

lemma *nat-add-left-cancel* [*simp*]: *(k + m = k + n) = (m = (n*::*nat))*
by (*induct k*) *simp-all*

lemma *nat-add-right-cancel* [*simp*]: *(m + k = n + k) = (m=(n*::*nat))*
by (*induct k*) *simp-all*

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*

Reasoning about *m + 0 = 0*, etc.

lemma *add-is-0* [*iff*]: *!!m*::*nat. (m + n = 0) = (m = 0 & n = 0)*
by (*case-tac m*) *simp-all*

lemma *add-is-1*: *(m+n= Suc 0) = (m= Suc 0 & n=0 | m=0 & n= Suc 0)*
by (*case-tac m*) *simp-all*

lemma *one-is-add*: $(Suc\ 0 = m + n) = (m = Suc\ 0\ \&\ n = 0 \mid m = 0\ \&\ n = Suc\ 0)$
by (*rule trans*, *rule eq-commute*, *rule add-is-1*)

lemma *add-gr-0* [*iff*]: $!!m::nat.\ (0 < m + n) = (0 < m \mid 0 < n)$
by (*simp del*: *neq0-conv add*: *neq0-conv* [*symmetric*])

lemma *add-eq-self-zero*: $!!m::nat.\ m + n = m ==> n = 0$
apply (*drule add-0-right* [*THEN ssubst*])
apply (*simp add*: *nat-add-assoc del*: *add-0-right*)
done

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

16.10 Multiplication

right annihilation in product

lemma *mult-0-right* [*simp*]: $(m::nat) * 0 = 0$
by (*induct m*) *simp-all*

right successor law for multiplication

lemma *mult-Suc-right* [*simp*]: $m * Suc\ n = m + (m * n)$
by (*induct m*) (*simp-all add*: *nat-add-left-commute*)

Commutative law for multiplication

lemma *nat-mult-commute*: $m * n = n * (m::nat)$
by (*induct m*) *simp-all*

addition distributes over multiplication

lemma *add-mult-distrib*: $(m + n) * k = (m * k) + ((n * k)::nat)$
by (*induct m*) (*simp-all add*: *nat-add-assoc nat-add-left-commute*)

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + ((k * n)::nat)$
by (*induct m*) (*simp-all add*: *nat-add-assoc*)

Associative law for multiplication

lemma *nat-mult-assoc*: $(m * n) * k = m * ((n * k)::nat)$
by (*induct m*) (*simp-all add*: *add-mult-distrib*)

The naturals form a *comm-semiring-1-cancel*

instance *nat* :: *comm-semiring-1-cancel*

proof
fix *i j k* :: *nat*
show $(i + j) + k = i + (j + k)$ **by** (*rule nat-add-assoc*)
show $i + j = j + i$ **by** (*rule nat-add-commute*)
show $0 + i = i$ **by** *simp*
show $(i * j) * k = i * (j * k)$ **by** (*rule nat-mult-assoc*)
show $i * j = j * i$ **by** (*rule nat-mult-commute*)
show $1 * i = i$ **by** *simp*
show $(i + j) * k = i * k + j * k$ **by** (*simp add*: *add-mult-distrib*)
show $0 \neq (1::nat)$ **by** *simp*
assume $k+i = k+j$ **thus** $i=j$ **by** *simp*
qed

lemma *mult-is-0* [*simp*]: $((m::nat) * n = 0) = (m=0 \mid n=0)$
apply (*induct m*)
apply (*induct-tac* [*2*] *n*, *simp-all*)
done

16.11 Monotonicity of Addition

strict, in 1st argument

lemma *add-less-mono1*: $i < j \Longrightarrow i + k < j + (k::nat)$
by (*induct k*) *simp-all*

strict, in both arguments

lemma *add-less-mono*: $[|i < j; k < l|] \Longrightarrow i + k < j + (l::nat)$
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*: $m < n \Longrightarrow (\exists k.\ n = Suc\ (m + k))$
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*: $(i::nat) < j \Longrightarrow 0 < k \Longrightarrow k * i < k * j$
apply (*erule-tac m1 = 0* **in** *less-imp-Suc-add* [*THEN exE*], *simp*)
apply (*induct-tac x*)
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 $i \leq j \Longrightarrow k + i \leq k + j$ **by** *simp*
show $i < j \Longrightarrow 0 < k \Longrightarrow k * i < k * j$ **by** (*simp add*: *mult-less-mono2*)
qed

lemma *nat-mult-1*: $(1::nat) * n = n$
by *simp*

lemma *nat-mult-1-right*: $n * (1::nat) = n$
by *simp*

16.12 Additional theorems about "less than"

A [clumsy] way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma *less-mono-imp-le-mono*:
assumes *lt-mono*: $!!i\ j::nat.\ i < j \Longrightarrow f\ i < f\ j$
and *le*: $i \leq j$ **shows** $f\ i \leq ((f\ j)::nat)$ **using** *le*
apply (*simp add*: *order-le-less*)
apply (*blast intro*!: *lt-mono*)
done

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j \Longrightarrow i + k \leq j + (k::nat)$
by (*rule add-right-mono*)

non-strict, in both arguments

lemma *add-le-mono*: $[|\ i \leq j;\ \ k \leq l\ |] \Longrightarrow i + k \leq j + (l::nat)$
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 (*erule add-lessD1* [*THEN less-irrefl*])
done

lemma *not-add-less2* [*iff*]: $\sim (j + i < (i::nat))$
by (*simp add*: *add-commute not-add-less1*)

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*: $!!k::nat.\ k < l \Longrightarrow m + l = k + n \Longrightarrow m < n$
by (*force simp del*: *add-Suc-right*
simp add: *less-iff-Suc-add add-Suc-right* [*symmetric*] *add-ac*)

16.13 Difference

lemma *diff-self-eq-0* [*simp*]: $(m::nat) - m = 0$
by (*induct m*) *simp-all*

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add-diff-inverse*: $\sim\ m < n \Longrightarrow n + (m - n) = (m::nat)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *le-add-diff-inverse* [*simp*]: $n \leq m \Longrightarrow n + (m - n) = (m::nat)$
by (*simp add*: *add-diff-inverse linorder-not-less*)

lemma *le-add-diff-inverse2* [*simp*]: $n \leq m \Longrightarrow (m - n) + n = (m::nat)$
by (*simp add*: *le-add-diff-inverse add-commute*)

16.14 More results about difference

lemma *Suc-diff-le*: $n \leq m \Longrightarrow Suc\ m - n = Suc\ (m - n)$
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 *less-imp-diff-less*: $(j::nat) < k \Longrightarrow j - n < k$
by (*rule le-less-trans*, *rule diff-le-self*)

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-Suc-less* [*simp*]: $0<n \Longrightarrow n - Suc\ i < n$
apply (*case-tac n*, *safe*)
apply (*simp add*: *le-simps*)
done

This and the next few suggested by Florian Kammueller

lemma *diff-commute*: $(i::nat) - j - k = i - k - j$
by (*simp add*: *diff-diff-left add-commute*)

lemma *diff-add-assoc*: $k \leq (j::nat) \Longrightarrow (i + j) - k = i + (j - k)$
by (*induct j k rule*: *diff-induct*) *simp-all*

lemma *diff-add-assoc2*: $k \leq (j::nat) \Longrightarrow (j + i) - k = (j - k) + i$
by (*simp add*: *add-commute diff-add-assoc*)

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-assoc*)

lemma *le-imp-diff-is-add*: $i \leq (j::nat) \Longrightarrow (j - i = k) = (j = k + i)$
apply *safe*
apply (*simp-all add*: *diff-add-inverse2*)
done

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*: $i < j \Longrightarrow \exists k::nat.\ 0 < k \ \&\ i + k = j$
apply (*rule-tac x* = $j - i$ **in** *exI*)
apply (*simp* (*no-asm-simp*) *add*: *add-diff-inverse less-not-sym*)
done

lemma *zero-induct-lemma*: $P\ k \Longrightarrow (!!n.\ P\ (Suc\ n) \Longrightarrow P\ n) \Longrightarrow P\ (k - i)$
apply (*induct k i rule*: *diff-induct*)
apply (*simp-all* (*no-asm*))
apply *iprover*
done

lemma *zero-induct*: $P\ k \Longrightarrow (!!n.\ P\ (Suc\ n) \Longrightarrow P\ n) \Longrightarrow P\ 0$
apply (*rule diff-self-eq-0* [*THEN subst*])
apply (*rule zero-induct-lemma*, *iprover*+)
done

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*

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

lemmas *nat-distrib* =

add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

16.15 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 (*case-tac* [*2*] *n*, *simp-all*)
done

lemma *one-le-mult-iff* [*simp*]: $(Suc\ 0 \leq m * n) = (1 \leq m \ \& \ 1 \leq n)$
apply (*induct m*)
apply (*case-tac* [*2*] *n*, *simp-all*)
done

lemma *mult-eq-1-iff* [*simp*]: $(m * n = Suc\ 0) = (m = 1 \ \& \ n = 1)$
apply (*induct m*, *simp*)
apply (*induct n*, *simp*, *fastsimp*)
done

lemma *one-eq-mult-iff* [*simp*]: $(Suc\ 0 = m * n) = (m = 1 \ \& \ n = 1)$
apply (*rule trans*)
apply (*rule-tac* [*2*] *mult-eq-1-iff*, *fastsimp*)
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 *mult-cancel2* [*simp*]: $(m * k = n * k) = (m = n \mid (k = (0::nat)))$
apply (*cut-tac less-linear*, *safe*, *auto*)
apply (*drule mult-less-mono1*, *assumption*, *simp*)+
done

lemma *mult-cancel1* [*simp*]: $(k * m = k * n) = (m = n \mid (k = (0::nat)))$
by (*simp add*: *mult-commute* [*of k*])

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 *Suc-mult-cancel1*: $(Suc\ k * m = Suc\ k * n) = (m = n)$
by (*subst mult-cancel1*) *simp*

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 (*fastsimp elim*!: *less-SucE*)
apply (*fastsimp dest*: *mult-less-mono2*)
done

end

17 NatArith: Further Arithmetic Facts Concerning the Natural Numbers

theory *NatArith*
imports *Nat*
uses *arith-data.ML*
begin

setup *arith-setup*

The following proofs may rely on the arithmetic proof procedures.

lemma *le-iff-add*: $(m::nat) \leq n = (\exists k.\ n = m + k)$

by (*auto simp*: *le-eq-less-or-eq dest*: *less-imp-Suc-add*)

lemma *pred-nat-trancl-eq-le*: $((m, n) : pred\text{-}nat\hat{}*) = (m \leq n)$
by (*simp add*: *less-eq reflcl-trancl* [*symmetric*]
del: *reflcl-trancl*, *arith*)

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* $a<b$ *rule*: *case-split*)
(*auto simp add*: *diff-is-0-eq* [*THEN iffD2*])

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 (*simp split*: *nat-diff-split*)

lemmas [*arith-split*] = *nat-diff-split split-min split-max*

lemma *le-square*: $m \leq m*(m::nat)$
by (*induct-tac m*, *auto*)

lemma *le-cube*: $(m::nat) \leq m*(m*m)$
by (*induct-tac m*, *auto*)

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*: $[|\ a < (b::nat);\ c \leq a\ |] ==> a-c < b-c$
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*: $k \leq j ==> (i \leq j-k) = (i+k \leq (j::nat))$
by *arith*

lemma *diff-diff-cancel* [*simp*]: $i \leq (n::nat) ==> n - (n - i) = i$
by *arith*

lemma *le-add-diff*: $k \leq (n::nat) ==> m \leq n + m - k$
by *arith*

lemma *diff-less*[*simp*]: $!!m::nat.\ [|\ 0<n;\ 0<m\ |] ==> m - n < m$
by *arith*

lemma *diff-diff-eq*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $((m-k) - (n-k)) = (m-n)$
by (*simp split add*: *nat-diff-split*)

lemma *eq-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k = n-k) = (m=n)$
by (*auto split add*: *nat-diff-split*)

lemma *less-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k < n-k) = (m<n)$
by (*auto split add*: *nat-diff-split*)

lemma *le-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k \leq n-k) = (m \leq n)$
by (*auto split add*: *nat-diff-split*)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq (n::nat)$ ==> $(m-l) \leq (n-l)$
by (*simp split add*: *nat-diff-split*)

lemma *diff-le-mono2*: $m \leq (n::nat)$ ==> $(l-n) \leq (l-m)$
by (*simp split add*: *nat-diff-split*)

lemma *diff-less-mono2*: [| $m < (n::nat)$; $m<l$ |] ==> $(l-n) < (l-m)$
by (*simp split add*: *nat-diff-split*)

lemma *diffs0-imp-equal*: !!$m::nat$. [| $m-n = 0$; $n-m = 0$ |] ==> $m=n$
by (*simp split add*: *nat-diff-split*)

Lemmas for ex/Factorization

lemma *one-less-mult*: [| *Suc 0* $<$ n; *Suc 0* $<$ m |] ==> *Suc 0* $<$ $m*n$
by (*case-tac m*, *auto*)

lemma *n-less-m-mult-n*: [| *Suc 0* $<$ n; *Suc 0* $<$ m |] ==> $n<m*n$
by (*case-tac m*, *auto*)

lemma *n-less-n-mult-m*: [| *Suc 0* $<$ n; *Suc 0* $<$ m |] ==> $n<n*m$
by (*case-tac m*, *auto*)

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j$ --> $i - (j - k) = i + (k::nat) - j$
by *arith*

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j$ ==> $m - Suc\ (j - k) = m + k - Suc\ j$
by *arith*

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j$ ==> $Suc\ (j - k) - m = Suc\ j - (k + m)$
by *arith*

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.

ML
⟪
val pred-nat-trancl-eq-le = thm pred-nat-trancl-eq-le;
val nat-diff-split = thm nat-diff-split;
val nat-diff-split-asm = thm nat-diff-split-asm;
val le-square = thm le-square;
val le-cube = thm le-cube;
val diff-less-mono = thm diff-less-mono;
val less-diff-conv = thm less-diff-conv;
val le-diff-conv = thm le-diff-conv;
val le-diff-conv2 = thm le-diff-conv2;
val diff-diff-cancel = thm diff-diff-cancel;
val le-add-diff = thm le-add-diff;
val diff-less = thm diff-less;
val diff-diff-eq = thm diff-diff-eq;
val eq-diff-iff = thm eq-diff-iff;
val less-diff-iff = thm less-diff-iff;
val le-diff-iff = thm le-diff-iff;
val diff-le-mono = thm diff-le-mono;
val diff-le-mono2 = thm diff-le-mono2;
val diff-less-mono2 = thm diff-less-mono2;
val diffs0-imp-equal = thm diffs0-imp-equal;
val one-less-mult = thm one-less-mult;
val n-less-m-mult-n = thm n-less-m-mult-n;
val n-less-n-mult-m = thm n-less-n-mult-m;
val diff-diff-right = thm diff-diff-right;
val diff-Suc-diff-eq1 = thm diff-Suc-diff-eq1;
val diff-Suc-diff-eq2 = thm diff-Suc-diff-eq2;
⟫

17.1 Embedding of the Naturals into any *comm-semiring-1-cancel*: *of-nat*

consts *of-nat* :: *nat* => *'a*::*comm-semiring-1-cancel*

primrec
of-nat-0: *of-nat 0 = 0*
of-nat-Suc: *of-nat* (*Suc m*) = *of-nat m + 1*

lemma *of-nat-1* [*simp*]: *of-nat 1 = 1*
by *simp*

lemma *of-nat-add* [*simp*]: *of-nat* (*m*+*n*) = *of-nat m* + *of-nat n*
apply (*induct m*)
apply (*simp-all add*: *add-ac*)
done

lemma *of-nat-mult* [*simp*]: *of-nat* (*m*∗*n*) = *of-nat m* ∗ *of-nat n*
apply (*induct m*)
apply (*simp-all add*: *mult-ac add-ac right-distrib*)
done

lemma *zero-le-imp-of-nat*: *0* ≤ (*of-nat m*::*'a*::*ordered-semidom*)
apply (*induct m*, *simp-all*)
apply (*erule order-trans*)
apply (*rule less-add-one* [*THEN order-less-imp-le*])
done

lemma *less-imp-of-nat-less*:
m < *n* ==> *of-nat m* < (*of-nat n*::*'a*::*ordered-semidom*)
apply (*induct m n rule*: *diff-induct*, *simp-all*)
apply (*insert add-le-less-mono* [*OF zero-le-imp-of-nat zero-less-one*], *force*)
done

lemma *of-nat-less-imp-less*:
of-nat m < (*of-nat n*::*'a*::*ordered-semidom*) ==> *m* < *n*
apply (*induct m n rule*: *diff-induct*, *simp-all*)
apply (*insert zero-le-imp-of-nat*)
apply (*force simp add*: *linorder-not-less* [*symmetric*])
done

lemma *of-nat-less-iff* [*simp*]:
(*of-nat m* < (*of-nat n*::*'a*::*ordered-semidom*)) = (*m*<*n*)
by (*blast intro*: *of-nat-less-imp-less less-imp-of-nat-less*)

Special cases where either operand is zero

lemmas *of-nat-0-less-iff* = *of-nat-less-iff* [*of 0*, *simplified*]
lemmas *of-nat-less-0-iff* = *of-nat-less-iff* [*of - 0*, *simplified*]
declare *of-nat-0-less-iff* [*simp*]
declare *of-nat-less-0-iff* [*simp*]

lemma *of-nat-le-iff* [*simp*]:
(*of-nat m* ≤ (*of-nat n*::*'a*::*ordered-semidom*)) = (*m* ≤ *n*)
by (*simp add*: *linorder-not-less* [*symmetric*])

Special cases where either operand is zero

lemmas *of-nat-0-le-iff* = *of-nat-le-iff* [*of 0*, *simplified*]
lemmas *of-nat-le-0-iff* = *of-nat-le-iff* [*of - 0*, *simplified*]
declare *of-nat-0-le-iff* [*simp*]
declare *of-nat-le-0-iff* [*simp*]

The ordering on the *comm-semiring-1-cancel* is necessary to exclude the

possibility of a finite field, which indeed wraps back to zero.

lemma *of-nat-eq-iff* [*simp*]:
  (*of-nat m* = (*of-nat n*::'*a*::*ordered-semidom*)) = (*m* = *n*)
by (*simp add*: *order-eq-iff*)

Special cases where either operand is zero

lemmas *of-nat-0-eq-iff* = *of-nat-eq-iff* [*of 0*, *simplified*]
lemmas *of-nat-eq-0-iff* = *of-nat-eq-iff* [*of - 0*, *simplified*]
declare *of-nat-0-eq-iff* [*simp*]
declare *of-nat-eq-0-iff* [*simp*]

lemma *of-nat-diff* [*simp*]:
  n ≤ *m* ==> *of-nat* (*m* − *n*) = *of-nat m* − (*of-nat n* :: '*a*::*comm-ring-1*)
by (*simp del*: *of-nat-add*
    add: *compare-rls of-nat-add* [*symmetric*] *split add*: *nat-diff-split*)

end

18 Datatype-Universe: Analogues of the Cartesian Product and Disjoint Sum for Datatypes

theory *Datatype-Universe*
imports *NatArith Sum-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
  apfst :: ['*a*=>'*c*, '*a*∗'*b*] => '*c*∗'*b*
  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 :: $[('a, 'b)\ dtree, ('a, 'b)\ dtree] \Rightarrow ('a, 'b)\ dtree$
In0 :: $('a, 'b)\ dtree \Rightarrow ('a, 'b)\ dtree$
In1 :: $('a, 'b)\ dtree \Rightarrow ('a, 'b)\ dtree$
Lim :: $('b \Rightarrow ('a, 'b)\ dtree) \Rightarrow ('a, 'b)\ dtree$

ntrunc :: $[nat, ('a, 'b)\ dtree] \Rightarrow ('a, 'b)\ dtree$

uprod :: $[('a, 'b)\ dtree\ set, ('a, 'b)\ dtree\ set] \Rightarrow ('a, 'b)\ dtree\ set$
usum :: $[('a, 'b)\ dtree\ set, ('a, 'b)\ dtree\ set] \Rightarrow ('a, 'b)\ dtree\ set$

Split :: $[[('a, 'b)\ dtree, ('a, 'b)\ dtree] \Rightarrow 'c, ('a, 'b)\ dtree] \Rightarrow 'c$
Case :: $[[('a, 'b)\ dtree] \Rightarrow 'c, [('a, 'b)\ dtree] \Rightarrow 'c, ('a, 'b)\ dtree] \Rightarrow 'c$

dprod :: $[(('a, 'b)\ dtree * ('a, 'b)\ dtree)set, (('a, 'b)\ dtree * ('a, 'b)\ dtree)set] \Rightarrow (('a, 'b)\ dtree * ('a, 'b)\ dtree)set$
dsum :: $[(('a, 'b)\ dtree * ('a, 'b)\ dtree)set, (('a, 'b)\ dtree * ('a, 'b)\ dtree)set] \Rightarrow (('a, 'b)\ dtree * ('a, 'b)\ dtree)set$

defs

Push-Node-def: *Push-Node* == (%*n x*. *Abs-Node* (*apfst* (*Push n*) (*Rep-Node x*)))

apfst-def: *apfst* == (%*f* (*x*,*y*). (*f*(*x*),*y*))
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-conv* [*simp*]: *apfst f (a,b) = (f(a),b)*
by (*simp add*: *apfst-def*)

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

18.1 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* = *Scons-not-Atom* [*THEN not-sym*, *standard*]
declare *Atom-not-Scons* [*iff*]

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* = *inj-Leaf* [*THEN injD*, *standard*]
declare *Leaf-inject* [*dest*!]

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* = *inj-Numb* [*THEN injD*, *standard*]
declare *Numb-inject* [*dest*!]

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* = *Scons-not-Leaf* [*THEN not-sym*, *standard*]
declare *Leaf-not-Scons* [*iff*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)
by (*simp add*: *Numb-def o-def Scons-not-Atom*)

lemmas *Numb-not-Scons* = *Scons-not-Numb* [*THEN not-sym*, *standard*]
declare *Numb-not-Scons* [*iff*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
by (*simp add*: *Leaf-def Numb-def*)

lemmas *Numb-not-Leaf* = *Leaf-not-Numb* [*THEN not-sym*, *standard*]
declare *Numb-not-Leaf* [*iff*]

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*)

18.2 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* = *In0-not-In1* [*THEN not-sym*, *standard*]
declare *In1-not-In0* [*iff*]

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*!]:
$[| w : dsum\ r\ s;$
$!!x\ x'.\ [| (x,x') : r;\ w = (In0(x), In0(x')) |] ==> P;$
$!!y\ y'.\ [| (y,y') : s;\ w = (In1(y), In1(y')) |] ==> P$
$|] ==> P$
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*, *stan-*

dard]

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*

ML
⟪
val apfst-conv = thm apfst-conv;
val apfst-convE = thm apfst-convE;
val Push-inject1 = thm Push-inject1;
val Push-inject2 = thm Push-inject2;
val Push-inject = thm Push-inject;
val Push-neq-K0 = thm Push-neq-K0;
val Abs-Node-inj = thm Abs-Node-inj;
val Node-K0-I = thm Node-K0-I;
val Node-Push-I = thm Node-Push-I;
val Scons-not-Atom = thm Scons-not-Atom;
val Atom-not-Scons = thm Atom-not-Scons;
val inj-Atom = thm inj-Atom;
val Atom-inject = thm Atom-inject;
val Atom-Atom-eq = thm Atom-Atom-eq;
val inj-Leaf = thm inj-Leaf;
val Leaf-inject = thm Leaf-inject;
val inj-Numb = thm inj-Numb;
val Numb-inject = thm Numb-inject;
val Push-Node-inject = thm Push-Node-inject;
val Scons-inject1 = thm Scons-inject1;
val Scons-inject2 = thm Scons-inject2;
val Scons-inject = thm Scons-inject;
val Scons-Scons-eq = thm Scons-Scons-eq;
val Scons-not-Leaf = thm Scons-not-Leaf;
val Leaf-not-Scons = thm Leaf-not-Scons;
val Scons-not-Numb = thm Scons-not-Numb;
val Numb-not-Scons = thm Numb-not-Scons;
val Leaf-not-Numb = thm Leaf-not-Numb;
val Numb-not-Leaf = thm Numb-not-Leaf;
val ndepth-K0 = thm ndepth-K0;
val ndepth-Push-Node-aux = thm ndepth-Push-Node-aux;
val ndepth-Push-Node = thm ndepth-Push-Node;
val ntrunc-0 = thm ntrunc-0;
val ntrunc-Atom = thm ntrunc-Atom;

val ntrunc-Leaf = thm ntrunc-Leaf;
val ntrunc-Numb = thm ntrunc-Numb;
val ntrunc-Scons = thm ntrunc-Scons;
val ntrunc-one-In0 = thm ntrunc-one-In0;
val ntrunc-In0 = thm ntrunc-In0;
val ntrunc-one-In1 = thm ntrunc-one-In1;
val ntrunc-In1 = thm ntrunc-In1;
val uprodI = thm uprodI;
val uprodE = thm uprodE;
val uprodE2 = thm uprodE2;
val usum-In0I = thm usum-In0I;
val usum-In1I = thm usum-In1I;
val usumE = thm usumE;
val In0-not-In1 = thm In0-not-In1;
val In1-not-In0 = thm In1-not-In0;
val In0-inject = thm In0-inject;
val In1-inject = thm In1-inject;
val In0-eq = thm In0-eq;
val In1-eq = thm In1-eq;
val inj-In0 = thm inj-In0;
val inj-In1 = thm inj-In1;
val Lim-inject = thm Lim-inject;
val ntrunc-subsetI = thm ntrunc-subsetI;
val ntrunc-subsetD = thm ntrunc-subsetD;
val ntrunc-equality = thm ntrunc-equality;
val ntrunc-o-equality = thm ntrunc-o-equality;
val uprod-mono = thm uprod-mono;
val usum-mono = thm usum-mono;
val Scons-mono = thm Scons-mono;
val In0-mono = thm In0-mono;
val In1-mono = thm In1-mono;
val Split = thm Split;
val Case-In0 = thm Case-In0;
val Case-In1 = thm Case-In1;
val ntrunc-UN1 = thm ntrunc-UN1;
val Scons-UN1-x = thm Scons-UN1-x;
val Scons-UN1-y = thm Scons-UN1-y;
val In0-UN1 = thm In0-UN1;
val In1-UN1 = thm In1-UN1;
val dprodI = thm dprodI;
val dprodE = thm dprodE;
val dsum-In0I = thm dsum-In0I;
val dsum-In1I = thm dsum-In1I;
val dsumE = thm dsumE;
val dprod-mono = thm dprod-mono;
val dsum-mono = thm dsum-mono;
val dprod-Sigma = thm dprod-Sigma;
val dprod-subset-Sigma = thm dprod-subset-Sigma;
val dprod-subset-Sigma2 = thm dprod-subset-Sigma2;

val dsum-Sigma = thm dsum-Sigma;
val dsum-subset-Sigma = thm dsum-subset-Sigma;
val Domain-dprod = thm Domain-dprod;
val Domain-dsum = thm Domain-dsum;
⟩⟩

end

19 Datatype: Datatypes

theory *Datatype*
imports *Datatype-Universe*
begin

19.1 Representing primitive types

rep-datatype *bool*
  distinct *True-not-False False-not-True*
  induction *bool-induct*

declare *case-split* [*cases type*: *bool*]
  — prefer plain propositional version

rep-datatype *unit*
  induction *unit-induct*

rep-datatype *prod*
  inject *Pair-eq*
  induction *prod-induct*

rep-datatype *sum*
  distinct *Inl-not-Inr Inr-not-Inl*
  inject *Inl-eq Inr-eq*
  induction *sum-induct*

ML ⟪⟪
  val [*sum-case-Inl*, *sum-case-Inr*] = *thms sum.cases*;
  bind-thm (*sum-case-Inl*, *sum-case-Inl*);
  bind-thm (*sum-case-Inr*, *sum-case-Inr*);
⟩⟩ — compatibility

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-case-Inl*)
  apply (*erule ssubst*)
  apply (*rule sum-case-Inr*)
  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 (*erule arg-cong*)

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 arbitrary)

Sumr :: (*'b => 'c*) *=> 'a + 'b => 'c*
Sumr == sum-case arbitrary

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*)

19.2 Finishing the datatype package setup

Belongs to theory *Datatype-Universe*; hides popular names.

hide *const Push Node Atom Leaf Numb Lim Split Case Suml Sumr*
hide *type node item*

19.3 Further cases/induct rules for tuples

lemma *prod-cases3* [*case-names fields*, *cases type*]:
(!!*a b c. y = (a, b, c) ==> P*) *==> P*
apply (*cases y*)
apply (*case-tac b*, *blast*)
done

lemma *prod-induct3* [*case-names fields*, *induct type*]:
(!!*a b c. P (a, b, c)*) *==> P x*

by (*cases x*) *blast*

lemma *prod-cases4* [*case-names fields*, *cases type*]:
(!!*a b c d*. *y* = (*a*, *b*, *c*, *d*) ==> *P*) ==> *P*
apply (*cases y*)
apply (*case-tac c*, *blast*)
done

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* [*case-names fields*, *cases type*]:
(!!*a b c d e*. *y* = (*a*, *b*, *c*, *d*, *e*) ==> *P*) ==> *P*
apply (*cases y*)
apply (*case-tac d*, *blast*)
done

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* [*case-names fields*, *cases type*]:
(!!*a b c d e f*. *y* = (*a*, *b*, *c*, *d*, *e*, *f*) ==> *P*) ==> *P*
apply (*cases y*)
apply (*case-tac e*, *blast*)
done

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* [*case-names fields*, *cases type*]:
(!!*a b c d e f g*. *y* = (*a*, *b*, *c*, *d*, *e*, *f*, *g*) ==> *P*) ==> *P*
apply (*cases y*)
apply (*case-tac f*, *blast*)
done

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*

19.4 The option type

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*

lemma *option-caseE*:
(*case x of None => P | Some y => Q y*) ==>
(*x = None ==> P ==> R*) ==>
(!!*y. x = Some y ==> Q y ==> R*) ==> *R*
by (*cases x*) *simp-all*

19.4.1 Operations

consts
the :: *'a option => 'a*
primrec
the (*Some x*) = *x*

consts
o2s :: *'a option => 'a set*
primrec
o2s None = {}
o2s (*Some x*) = {*x*}

lemma *ospec* [*dest*]: (*ALL x:o2s A. P x*) ==> *A = Some x ==> P x*
by *simp*

ML-setup ⟪ *claset-ref*() := *claset*() *addSD2* (*ospec*, *thm ospec*) ⟫

lemma *elem-o2s* [*iff*]: (*x : o2s xo*) = (*xo = Some x*)
by (*cases xo*) *auto*

lemma *o2s-empty-eq* [*simp*]: (*o2s xo* = {}) = (*xo = None*)
by (*cases xo*) *auto*

constdefs
option-map :: (*'a => 'b*) => (*'a option => 'b option*)
option-map == %*f y. case y of None => None | Some x => Some* (*f x*)

lemma *option-map-None* [*simp*]: *option-map f None = None*
by (*simp add*: *option-map-def*)

lemma *option-map-Some* [*simp*]: *option-map f* (*Some x*) = *Some* (*f x*)
by (*simp add*: *option-map-def*)

lemma *option-map-is-None*[*iff*]:
(*option-map f opt = None*) = (*opt = None*)
by (*simp add*: *option-map-def split add*: *option.split*)

lemma *option-map-eq-Some* [*iff*]:

```
    (option-map f xo = Some y) = (EX z. xo = Some z & f z = y)
by (simp add: option-map-def split add: option.split)

lemma option-map-comp:
 option-map f (option-map g opt) = option-map (f o g) opt
by (simp add: option-map-def split add: option.split)

lemma option-map-o-sum-case [simp]:
    option-map f o sum-case g h = sum-case (option-map f o g) (option-map f o h)
  apply (rule ext)
  apply (simp split add: sum.split)
  done

lemmas [code] = imp-conv-disj

end



theory Divides
imports Datatype
begin


axclass
  div < type

instance  nat :: div ..

consts
  div  :: 'a::div ⇒ 'a ⇒ 'a          (infixl 70)
  mod  :: 'a::div ⇒ 'a ⇒ 'a          (infixl 70)
  dvd  :: 'a::times ⇒ 'a ⇒ bool      (infixl 50)


defs

  mod-def:   m mod n == wfrec (trancl pred-nat)
                        (%f j. if j<n | n=0 then j else f (j-n)) m

  div-def:   m div n == wfrec (trancl pred-nat)
                        (%f j. if j<n | n=0 then 0 else Suc (f (j-n))) m


  dvd-def:   m dvd n == ∃k. n = m*k


constdefs
  quorem :: (nat*nat) * (nat*nat) => bool
```

quorem == %((*a*,*b*), (*q*,*r*)).
a = *b***q* + *r* &
(*if* 0<*b then* 0≤*r* & *r*<*b else b*<*r* & *r* ≤0)

19.5 Initial Lemmas

lemmas *wf-less-trans* =
def-wfrec [*THEN trans*, *OF eq-reflection wf-pred-nat* [*THEN wf-trancl*],
standard]

lemma *mod-eq*: (%*m*. *m mod n*) =
wfrec (*trancl pred-nat*) (%*f j*. *if j*<*n* | *n*=0 *then j else f* (*j*−*n*))
by (*simp add*: *mod-def*)

lemma *div-eq*: (%*m*. *m div n*) = *wfrec* (*trancl pred-nat*)
(%*f j*. *if j*<*n* | *n*=0 *then* 0 *else Suc* (*f* (*j*−*n*)))
by (*simp add*: *div-def*)

lemma *DIVISION-BY-ZERO-DIV* [*simp*]: *a div* 0 = (0::*nat*)
by (*rule div-eq* [*THEN wf-less-trans*], *simp*)

lemma *DIVISION-BY-ZERO-MOD* [*simp*]: *a mod* 0 = (*a*::*nat*)
by (*rule mod-eq* [*THEN wf-less-trans*], *simp*)

19.6 Remainder

lemma *mod-less* [*simp*]: *m*<*n* ==> *m mod n* = (*m*::*nat*)
by (*rule mod-eq* [*THEN wf-less-trans*], *simp*)

lemma *mod-geq*: ~ *m* < (*n*::*nat*) ==> *m mod n* = (*m*−*n*) *mod n*
apply (*case-tac n*=0, *simp*)
apply (*rule mod-eq* [*THEN wf-less-trans*])
apply (*simp add*: *cut-apply less-eq*)
done

lemma *le-mod-geq*: (*n*::*nat*) ≤ *m* ==> *m mod n* = (*m*−*n*) *mod n*
by (*simp add*: *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*: *mod-geq*)

lemma *mod-1* [*simp*]: *m mod Suc* 0 = 0
apply (*induct m*)
apply (*simp-all* (*no-asm-simp*) *add*: *mod-geq*)
done

lemma *mod-self* [*simp*]: *n mod n* = (*0*::*nat*)
apply (*case-tac n=0*)
apply (*simp-all add*: *mod-geq*)
done

lemma *mod-add-self2* [*simp*]: (*m*+*n*) *mod n* = *m mod* (*n*::*nat*)
apply (*subgoal-tac* (*n* + *m*) *mod n* = (*n*+*m*−*n*) *mod n*)
apply (*simp add*: *add-commute*)
apply (*subst mod-geq* [*symmetric*], *simp-all*)
done

lemma *mod-add-self1* [*simp*]: (*n*+*m*) *mod n* = *m mod* (*n*::*nat*)
by (*simp add*: *add-commute mod-add-self2*)

lemma *mod-mult-self1* [*simp*]: (*m* + *k*∗*n*) *mod n* = *m mod* (*n*::*nat*)
apply (*induct k*)
apply (*simp-all add*: *add-left-commute* [*of* - *n*])
done

lemma *mod-mult-self2* [*simp*]: (*m* + *n*∗*k*) *mod n* = *m mod* (*n*::*nat*)
by (*simp add*: *mult-commute mod-mult-self1*)

lemma *mod-mult-distrib*: (*m mod n*) ∗ (*k*::*nat*) = (*m*∗*k*) *mod* (*n*∗*k*)
apply (*case-tac n=0*, *simp*)
apply (*case-tac 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 *mod-mult-self-is-0* [*simp*]: (*m*∗*n*) *mod n* = (*0*::*nat*)
apply (*case-tac n=0*, *simp*)
apply (*induct m*, *simp*)
apply (*rename-tac k*)
apply (*cut-tac m* = *k*∗*n* **and** *n* = *n* **in** *mod-add-self2*)
apply (*simp add*: *add-commute*)
done

lemma *mod-mult-self1-is-0* [*simp*]: (*n*∗*m*) *mod n* = (*0*::*nat*)
by (*simp add*: *mult-commute mod-mult-self-is-0*)

19.7 Quotient

lemma *div-less* [*simp*]: *m*<*n* ==> *m div n* = (*0*::*nat*)
by (*rule div-eq* [*THEN wf-less-trans*], *simp*)

lemma *div-geq*: [| *0*<*n*; ~*m*<*n* |] ==> *m div n* = *Suc*((*m*−*n*) *div n*)
apply (*rule div-eq* [*THEN wf-less-trans*])
apply (*simp add*: *cut-apply less-eq*)
done

lemma *le-div-geq*: [| *0*<*n*; *n*≤*m* |] ==> *m div n* = *Suc*((*m*−*n*) *div n*)
by (*simp add*: *div-geq linorder-not-less*)

lemma *div-if*: *0*<*n* ==> *m div n* = (*if m*<*n then 0 else Suc*((*m*−*n*) *div n*))
by (*simp add*: *div-geq*)

lemma *mod-div-equality*: (*m div n*)∗*n* + *m mod n* = (*m*::*nat*)
apply (*case-tac n*=*0*, *simp*)
apply (*induct m rule*: *nat-less-induct*)
apply (*subst mod-if*)
apply (*simp-all* (*no-asm-simp*) *add*: *add-assoc div-geq add-diff-inverse*)
done

lemma *mod-div-equality2*: *n* ∗ (*m div n*) + *m mod n* = (*m*::*nat*)
apply(*cut-tac m* = *m* **and** *n* = *n* **in** *mod-div-equality*)
apply(*simp add*: *mult-commute*)
done

19.8 Simproc for Cancelling Div and Mod

lemma *div-mod-equality*: ((*m div n*)∗*n* + *m mod n*) + *k* = (*m*::*nat*) + *k*
apply(*simp add*: *mod-div-equality*)
done

lemma *div-mod-equality2*: (*n*∗(*m div n*) + *m mod n*) + *k* = (*m*::*nat*) + *k*
apply(*simp add*: *mod-div-equality2*)
done

ML
⟪
val div-mod-equality = *thm div-mod-equality*;
val div-mod-equality2 = *thm div-mod-equality2*;

structure CancelDivModData =
struct

val div-name = *Divides.op div*;
val mod-name = *Divides.op mod*;
val mk-binop = *HOLogic.mk-binop*;
val mk-sum = *NatArithUtils.mk-sum*;

val dest-sum = NatArithUtils.dest-sum;

(∗*logic*∗)

val div-mod-eqs = map mk-meta-eq [*div-mod-equality*,*div-mod-equality2*]

val trans = trans

val prove-eq-sums =
  let val simps = add-0 :: add-0-right :: add-ac
  in NatArithUtils.prove-conv all-tac (*NatArithUtils.simp-all-tac simps*) *end*;

end;

structure CancelDivMod = CancelDivModFun(*CancelDivModData*);

val cancel-div-mod-proc = NatArithUtils.prep-simproc
    (*cancel-div-mod*, [(*m*::*nat*) + *n*], *CancelDivMod.proc*);

Addsimprocs[*cancel-div-mod-proc*];
⟩⟩

lemma *mult-div-cancel*: (*n*::*nat*) ∗ (*m div n*) = *m* − (*m mod n*)
by (*cut-tac m* = *m* **and** *n* = *n* **in** *mod-div-equality2*, *arith*)

lemma *mod-less-divisor* [*simp*]: *0*<*n* ==> *m mod n* < (*n*::*nat*)
apply (*induct m rule*: *nat-less-induct*)
apply (*case-tac na*<*n*, *simp*)

case *n* ≤ *na*

apply (*simp add*: *mod-geq*)
done

lemma *mod-le-divisor*[*simp*]: *0* < *n* ⟹ *m mod n* ≤ (*n*::*nat*)
apply(*drule mod-less-divisor*[**where** *m* = *m*])
apply *simp*
done

lemma *div-mult-self-is-m* [*simp*]: *0*<*n* ==> (*m*∗*n*) *div n* = (*m*::*nat*)
by (*cut-tac m* = *m*∗*n* **and** *n* = *n* **in** *mod-div-equality*, *auto*)

lemma *div-mult-self1-is-m* [*simp*]: *0*<*n* ==> (*n*∗*m*) *div n* = (*m*::*nat*)
by (*simp add*: *mult-commute div-mult-self-is-m*)

19.9 Proving facts about Quotient and Remainder

lemma *unique-quotient-lemma*:
    [| *b*∗*q'* + *r'* ≤ *b*∗*q* + *r*; *x* < *b*; *r* < *b* |]

```
    ==> q' ≤ (q::nat)
apply (rule leI)
apply (subst less-iff-Suc-add)
apply (auto simp add: add-mult-distrib2)
done

lemma unique-quotient:
    [| quorem ((a,b), (q,r));  quorem ((a,b), (q',r'));  0 < b |]
    ==> q = q'
apply (simp add: split-ifs quorem-def)
apply (blast intro: order-antisym
           dest: order-eq-refl [THEN unique-quotient-lemma] sym)
done

lemma unique-remainder:
    [| quorem ((a,b), (q,r));  quorem ((a,b), (q',r'));  0 < b |]
    ==> r = r'
apply (subgoal-tac q = q')
prefer 2 apply (blast intro: unique-quotient)
apply (simp add: quorem-def)
done

lemma quorem-div-mod: 0 < b ==> quorem ((a, b), (a div b, a mod b))
by (auto simp add: quorem-def)

lemma quorem-div: [| quorem((a,b),(q,r));  0 < b |] ==> a div b = q
by (simp add: quorem-div-mod [THEN unique-quotient])

lemma quorem-mod: [| quorem((a,b),(q,r));  0 < b |] ==> a mod b = r
by (simp add: quorem-div-mod [THEN unique-remainder])



lemma div-0 [simp]: 0 div m = (0::nat)
by (case-tac m=0, simp-all)

lemma mod-0 [simp]: 0 mod m = (0::nat)
by (case-tac m=0, simp-all)



lemma quorem-mult1-eq:
    [| quorem((b,c),(q,r));  0 < c |]
    ==> quorem ((a*b, c), (a*q + a*r div c, a*r mod c))
apply (auto simp add: split-ifs mult-ac quorem-def add-mult-distrib2)
done

lemma div-mult1-eq: (a*b) div c = a*(b div c) + a*(b mod c) div (c::nat)
apply (case-tac c = 0, simp)
```

apply (*blast intro*: *quorem-div-mod* [*THEN quorem-mult1-eq*, *THEN quorem-div*])
done

lemma *mod-mult1-eq*: $(a*b)\ mod\ c = a*(b\ mod\ c)\ mod\ (c::nat)$
apply (*case-tac* $c = 0$, *simp*)
apply (*blast intro*: *quorem-div-mod* [*THEN quorem-mult1-eq*, *THEN quorem-mod*])
done

lemma *mod-mult1-eq'*: $(a*b)\ mod\ (c::nat) = ((a\ mod\ c) * b)\ mod\ c$
apply (*rule trans*)
apply (*rule-tac* $s = b*a\ mod\ c$ **in** *trans*)
apply (*rule-tac* [*2*] *mod-mult1-eq*)
apply (*simp-all* (*no-asm*) *add*: *mult-commute*)
done

lemma *mod-mult-distrib-mod*: $(a*b)\ mod\ (c::nat) = ((a\ mod\ c) * (b\ mod\ c))\ mod\ c$
apply (*rule mod-mult1-eq'* [*THEN trans*])
apply (*rule mod-mult1-eq*)
done

lemma *quorem-add1-eq*:
$[|\ quorem((a,c),(aq,ar));\ quorem((b,c),(bq,br));\ 0 < c\ |] ==> quorem\ ((a+b, c), (aq + bq + (ar+br)\ div\ c, (ar+br)\ mod\ c))$
by (*auto simp add*: *split-ifs mult-ac quorem-def add-mult-distrib2*)

lemma *div-add1-eq*:
$(a+b)\ div\ (c::nat) = a\ div\ c + b\ div\ c + ((a\ mod\ c + b\ mod\ c)\ div\ c)$
apply (*case-tac* $c = 0$, *simp*)
apply (*blast intro*: *quorem-add1-eq* [*THEN quorem-div*] *quorem-div-mod quorem-div-mod*)
done

lemma *mod-add1-eq*: $(a+b)\ mod\ (c::nat) = (a\ mod\ c + b\ mod\ c)\ mod\ c$
apply (*case-tac* $c = 0$, *simp*)
apply (*blast intro*: *quorem-div-mod quorem-div-mod*
quorem-add1-eq [*THEN quorem-mod*])
done

19.10 Proving *a div (b * c) = a div b div c*

lemma *mod-lemma*: $[|\ (0::nat) < c;\ r < b\ |] ==> b * (q\ mod\ c) + r < b * c$
apply (*cut-tac* $m = q$ **and** $n = c$ **in** *mod-less-divisor*)
apply (*drule-tac* [*2*] $m = q\ mod\ c$ **in** *less-imp-Suc-add*, *auto*)
apply (*erule-tac* $P = \%x.\ ?lhs < ?rhs\ x$ **in** *ssubst*)
apply (*simp add*: *add-mult-distrib2*)
done

lemma *quorem-mult2-eq*: [| *quorem* ((*a*,*b*), (*q*,*r*)); *0* < *b*; *0* < *c* |]
    ==> *quorem* ((*a*, *b*∗*c*), (*q div c*, *b*∗(*q mod c*) + *r*))
apply (*auto simp add*: *mult-ac quorem-def add-mult-distrib2* [*symmetric*] *mod-lemma*)
done

lemma *div-mult2-eq*: *a div* (*b*∗*c*) = (*a div b*) *div* (*c*::*nat*)
apply (*case-tac b*=*0*, *simp*)
apply (*case-tac c*=*0*, *simp*)
apply (*force simp add*: *quorem-div-mod* [*THEN quorem-mult2-eq*, *THEN quorem-div*])
done

lemma *mod-mult2-eq*: *a mod* (*b*∗*c*) = *b*∗(*a div b mod c*) + *a mod* (*b*::*nat*)
apply (*case-tac b*=*0*, *simp*)
apply (*case-tac c*=*0*, *simp*)
apply (*auto simp add*: *mult-commute quorem-div-mod* [*THEN quorem-mult2-eq*, *THEN quorem-mod*])
done

19.11 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 (*case-tac 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

19.12 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: *m div Suc 0* = *m*
apply (*induct m*)
apply (*simp-all* (*no-asm-simp*) *add*: *div-geq*)
done

lemma *div-self* [*simp*]: *0*<*n* ==> *n div n* = (*1*::*nat*)
by (*simp add*: *div-geq*)

lemma *div-add-self2*: *0*<*n* ==> (*m*+*n*) *div n* = *Suc* (*m div n*)
apply (*subgoal-tac* (*n* + *m*) *div n* = *Suc* ((*n*+*m*−*n*) *div n*))
apply (*simp add*: *add-commute*)
apply (*subst div-geq* [*symmetric*], *simp-all*)
done

lemma *div-add-self1*: *0<n ==> (n+m) div n = Suc (m div n)*
by (*simp add*: *add-commute div-add-self2*)

lemma *div-mult-self1* [*simp*]: !!*n*::*nat*. *0<n ==> (m + k*n) div n = k + m div n*
apply (*subst div-add1-eq*)
apply (*subst div-mult1-eq*, *simp*)
done

lemma *div-mult-self2* [*simp*]: *0<n ==> (m + n*k) div n = k + m div (n*::*nat)*
by (*simp add*: *mult-commute div-mult-self1*)

lemma *div-le-mono* [*rule-format* (*no-asm*)]:
    ∀ *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

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

lemma *nat-mod-div-trivial* [*simp*]: *m mod n div n* = (*0* :: *nat*)
by (*case-tac n*=*0*, *auto*)

lemma *nat-mod-mod-trivial* [*simp*]: *m mod n mod n* = (*m mod n* :: *nat*)
by (*case-tac n*=*0*, *auto*)

19.13 The Divides Relation

lemma *dvdI* [*intro?*]: *n* = *m* ∗ *k* ==> *m dvd n*
by (*unfold dvd-def*, *blast*)

lemma *dvdE* [*elim?*]: !!*P*. [|*m dvd n*; !!*k*. *n* = *m*∗*k* ==> *P*|] ==> *P*
by (*unfold dvd-def*, *blast*)

lemma *dvd-0-right* [*iff*]: *m dvd* (*0*::*nat*)
apply (*unfold dvd-def*)
apply (*blast intro*: *mult-0-right* [*symmetric*])
done

lemma *dvd-0-left*: *0 dvd m ==> m = (0*::*nat)*
by (*force simp add*: *dvd-def*)

lemma *dvd-0-left-iff* [*iff*]: (*0 dvd* (*m*::*nat*)) = (*m = 0*)
by (*blast intro*: *dvd-0-left*)

lemma *dvd-1-left* [*iff*]: *Suc 0 dvd k*
by (*unfold dvd-def*, *simp*)

lemma *dvd-1-iff-1* [*simp*]: (*m dvd Suc 0*) = (*m = Suc 0*)
by (*simp add*: *dvd-def*)

lemma *dvd-refl* [*simp*]: *m dvd* (*m*::*nat*)
apply (*unfold dvd-def*)
apply (*blast intro*: *mult-1-right* [*symmetric*])
done

lemma *dvd-trans* [*trans*]: [| *m dvd n*; *n dvd p* |] ==> *m dvd* (*p*::*nat*)
apply (*unfold dvd-def*)
apply (*blast intro*: *mult-assoc*)
done

lemma *dvd-anti-sym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n*::*nat*)
apply (*unfold dvd-def*)
apply (*force dest*: *mult-eq-self-implies-10 simp add*: *mult-assoc mult-eq-1-iff*)
done

lemma *dvd-add*: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m*+*n* :: *nat*)
apply (*unfold dvd-def*)
apply (*blast intro*: *add-mult-distrib2* [*symmetric*])
done

lemma *dvd-diff*: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m*−*n* :: *nat*)
apply (*unfold dvd-def*)
apply (*blast intro*: *diff-mult-distrib2* [*symmetric*])
done

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** *dvd-diff*, *auto*)

lemma *dvd-mult*: *k dvd n ==> k dvd* (*m*∗*n* :: *nat*)
apply (*unfold dvd-def*)
apply (*blast intro*: *mult-left-commute*)
done

lemma *dvd-mult2*: *k dvd m ==> k dvd* (*m*∗*n* :: *nat*)
apply (*subst mult-commute*)
apply (*erule dvd-mult*)
done

lemma *dvd-triv-right* [*iff*]: *k dvd* (*m*∗*k* :: *nat*)
by (*rule dvd-refl* [*THEN dvd-mult*])

lemma *dvd-triv-left* [*iff*]: *k dvd* (*k*∗*m* :: *nat*)
by (*rule dvd-refl* [*THEN dvd-mult2*])

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 dvd-diff*)
apply (*rule dvd-refl*)
done

lemma *dvd-mod*: !!*n*::*nat*. [| *f dvd m*; *f dvd n* |] ==> *f dvd m mod n*
apply (*unfold dvd-def*)
apply (*case-tac n*=*0*, *auto*)
apply (*blast intro*: *mod-mult-distrib2* [*symmetric*])
done

lemma *dvd-mod-imp-dvd*: [| (*k*::*nat*) *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

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*
apply (*unfold 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 *mult-dvd-mono*: [| *i dvd m*; *j dvd n*|] *==> i∗j dvd* (*m∗n* :: *nat*)
apply (*unfold dvd-def*, *clarify*)
apply (*rule-tac x = k∗ka* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma *dvd-mult-left*: (*i∗j* :: *nat*) *dvd k ==> i dvd k*
by (*simp add*: *dvd-def mult-assoc*, *blast*)

lemma *dvd-mult-right*: (*i∗j* :: *nat*) *dvd k ==> j dvd k*
apply (*unfold dvd-def*, *clarify*)
apply (*rule-tac x = i∗k* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma *dvd-imp-le*: [| *k dvd n*; *0 < n* |] *==> k ≤* (*n*::*nat*)
apply (*unfold dvd-def*, *clarify*)
apply (*simp-all* (*no-asm-use*) *add*: *zero-less-mult-iff*)
apply (*erule conjE*)
apply (*rule le-trans*)
apply (*rule-tac* [*2*] *le-refl* [*THEN mult-le-mono*])
apply (*erule-tac* [*2*] *Suc-leI*, *simp*)
done

lemma *dvd-eq-mod-eq-0*: !!*k*::*nat*. (*k dvd n*) = (*n mod k = 0*)
apply (*unfold dvd-def*)
apply (*case-tac k=0*, *simp*, *safe*)
apply (*simp add*: *mult-commute*)
apply (*rule-tac t = n* **and** *n1 = k* **in** *mod-div-equality* [*THEN subst*])
apply (*subst mult-commute*, *simp*)
done

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 *mod-eq-0-iff*: $(m\ mod\ d = 0) = (\exists\, q::nat.\ m = d*q)$
by (*auto simp add*: *dvd-eq-mod-eq-0* [*symmetric*] *dvd-def*)

lemmas *mod-eq-0D* = *mod-eq-0-iff* [*THEN iffD1*]
declare *mod-eq-0D* [*dest*!]

lemma *mod-eqD*: $(m\ mod\ d = r) ==> \exists\, q::nat.\ m = r + q*d$
apply (*cut-tac* $m = m$ **in** *mod-div-equality*)
apply (*simp only*: *add-ac*)
apply (*blast intro*: *sym*)
done

lemma *split-div*:
$P(n\ div\ k :: nat) =$
$((k = 0 \longrightarrow P\ 0) \wedge (k \neq 0 \longrightarrow (!i.\ !j<k.\ n = k*i + j \longrightarrow P\ i)))$
(**is** *?P* = *?Q* **is** - = (- ∧ (- ⟶ *?R*)))
proof
  assume *P*: *?P*
  show *?Q*
  proof (*cases*)
    assume $k = 0$
    with *P* **show** *?Q* **by**(*simp add*:*DIVISION-BY-ZERO-DIV*)
  next
    assume *not0*: $k \neq 0$
    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 $i \neq 0$
        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 add*:*DIVISION-BY-ZERO-DIV*)
  next
    assume *not0*: $k \neq 0$

```
    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:
  0 < n ⟹ (n * q ≤ m ∧ m < n * (Suc q)) = (q = ((m::nat) div n))
  apply (rule iffI)
  apply (rule-tac a=m and r = m - n * q and r' = m mod n in unique-quotient)
prefer 3 apply assumption
  apply (simp-all add: quorem-def)
  apply arith
  apply (rule conjI)
  apply (rule-tac P=%x. n * (m div n) ≤ x in
    subst [OF mod-div-equality [of - n]])
  apply (simp only: add: mult-ac)
  apply (rule-tac P=%x. x < n + n * (m div n) in
    subst [OF mod-div-equality [of - n]])
  apply (simp only: add: mult-ac add-ac)
  apply (rule add-less-mono1, simp)
  done

theorem split-div':
  P ((m::nat) div n) = ((n = 0 ∧ P 0) ∨
   (∃ q. (n * q ≤ m ∧ m < n * (Suc q)) ∧ P q))
  apply (case-tac 0 < n)
  apply (simp only: add: split-div-lemma)
  apply (simp-all add: DIVISION-BY-ZERO-DIV)
  done

lemma split-mod:
 P(n mod k :: nat) =
 ((k = 0 ⟶ P n) ∧ (k ≠ 0 ⟶ (!i. !j<k. n = k*i + j ⟶ P j)))
 (is ?P = ?Q is - = (- ∧ (- ⟶ ?R)))
proof
  assume P: ?P
  show ?Q
  proof (cases)
    assume k = 0
    with P show ?Q by(simp add:DIVISION-BY-ZERO-MOD)
  next
    assume not0: k ≠ 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 add:DIVISION-BY-ZERO-MOD)
  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
```

19.14 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
  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
```

ML

```
⟪
val div-def = thm div-def
val mod-def = thm mod-def
val dvd-def = thm dvd-def
val quorem-def = thm quorem-def

val wf-less-trans = thm wf-less-trans;
val mod-eq = thm mod-eq;
val div-eq = thm div-eq;
val DIVISION-BY-ZERO-DIV = thm DIVISION-BY-ZERO-DIV;
val DIVISION-BY-ZERO-MOD = thm DIVISION-BY-ZERO-MOD;
val mod-less = thm mod-less;
val mod-geq = thm mod-geq;
val le-mod-geq = thm le-mod-geq;
val mod-if = thm mod-if;
val mod-1 = thm mod-1;
val mod-self = thm mod-self;
val mod-add-self2 = thm mod-add-self2;
val mod-add-self1 = thm mod-add-self1;
val mod-mult-self1 = thm mod-mult-self1;
val mod-mult-self2 = thm mod-mult-self2;
val mod-mult-distrib = thm mod-mult-distrib;
val mod-mult-distrib2 = thm mod-mult-distrib2;
val mod-mult-self-is-0 = thm mod-mult-self-is-0;
val mod-mult-self1-is-0 = thm mod-mult-self1-is-0;
val div-less = thm div-less;
val div-geq = thm div-geq;
val le-div-geq = thm le-div-geq;
val div-if = thm div-if;
val mod-div-equality = thm mod-div-equality;
val mod-div-equality2 = thm mod-div-equality2;
val div-mod-equality = thm div-mod-equality;
val div-mod-equality2 = thm div-mod-equality2;
val mult-div-cancel = thm mult-div-cancel;
val mod-less-divisor = thm mod-less-divisor;
val div-mult-self-is-m = thm div-mult-self-is-m;
```

val div-mult-self1-is-m = thm div-mult-self1-is-m;
val unique-quotient-lemma = thm unique-quotient-lemma;
val unique-quotient = thm unique-quotient;
val unique-remainder = thm unique-remainder;
val div-0 = thm div-0;
val mod-0 = thm mod-0;
val div-mult1-eq = thm div-mult1-eq;
val mod-mult1-eq = thm mod-mult1-eq;
val mod-mult1-eq′ = thm mod-mult1-eq′;
val mod-mult-distrib-mod = thm mod-mult-distrib-mod;
val div-add1-eq = thm div-add1-eq;
val mod-add1-eq = thm mod-add1-eq;
val mod-lemma = thm mod-lemma;
val div-mult2-eq = thm div-mult2-eq;
val mod-mult2-eq = thm mod-mult2-eq;
val div-mult-mult-lemma = thm div-mult-mult-lemma;
val div-mult-mult1 = thm div-mult-mult1;
val div-mult-mult2 = thm div-mult-mult2;
val div-1 = thm div-1;
val div-self = thm div-self;
val div-add-self2 = thm div-add-self2;
val div-add-self1 = thm div-add-self1;
val div-mult-self1 = thm div-mult-self1;
val div-mult-self2 = thm div-mult-self2;
val div-le-mono = thm div-le-mono;
val div-le-mono2 = thm div-le-mono2;
val div-le-dividend = thm div-le-dividend;
val div-less-dividend = thm div-less-dividend;
val mod-Suc = thm mod-Suc;
val dvdI = thm dvdI;
val dvdE = thm dvdE;
val dvd-0-right = thm dvd-0-right;
val dvd-0-left = thm dvd-0-left;
val dvd-0-left-iff = thm dvd-0-left-iff;
val dvd-1-left = thm dvd-1-left;
val dvd-1-iff-1 = thm dvd-1-iff-1;
val dvd-refl = thm dvd-refl;
val dvd-trans = thm dvd-trans;
val dvd-anti-sym = thm dvd-anti-sym;
val dvd-add = thm dvd-add;
val dvd-diff = thm dvd-diff;
val dvd-diffD = thm dvd-diffD;
val dvd-diffD1 = thm dvd-diffD1;
val dvd-mult = thm dvd-mult;
val dvd-mult2 = thm dvd-mult2;
val dvd-reduce = thm dvd-reduce;
val dvd-mod = thm dvd-mod;
val dvd-mod-imp-dvd = thm dvd-mod-imp-dvd;
val dvd-mod-iff = thm dvd-mod-iff;

val dvd-mult-cancel = thm dvd-mult-cancel;
val dvd-mult-cancel1 = thm dvd-mult-cancel1;
val dvd-mult-cancel2 = thm dvd-mult-cancel2;
val mult-dvd-mono = thm mult-dvd-mono;
val dvd-mult-left = thm dvd-mult-left;
val dvd-mult-right = thm dvd-mult-right;
val dvd-imp-le = thm dvd-imp-le;
val dvd-eq-mod-eq-0 = thm dvd-eq-mod-eq-0;
val dvd-mult-div-cancel = thm dvd-mult-div-cancel;
val mod-eq-0-iff = thm mod-eq-0-iff;
val mod-eqD = thm mod-eqD;
⟫

end

20 Power: Exponentiation

theory *Power*
imports *Divides*
begin

20.1 Powers for Arbitrary Semirings

axclass *recpower* ⊆ *comm-semiring-1-cancel*, *power*
power-0 [*simp*]: *a ^ 0 = 1*
power-Suc: *a ^ (Suc n) = a * (a ^ n)*

lemma *power-0-Suc* [*simp*]: *(0::'a::recpower) ^ (Suc n) = 0*
by (*simp add*: *power-Suc*)

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))*
by (*induct n*, *auto*)

lemma *power-one* [*simp*]: *1^n = (1::'a::recpower)*
apply (*induct n*)
apply (*auto simp add*: *power-Suc*)
done

lemma *power-one-right* [*simp*]: *(a::'a::recpower) ^ 1 = a*
by (*simp add*: *power-Suc*)

lemma *power-add*: *(a::'a::recpower) ^ (m+n) = (a^m) * (a^n)*
apply (*induct n*)
apply (*simp-all add*: *power-Suc mult-ac*)
done

lemma *power-mult*: $(a::'a::recpower)\ \hat{}\ (m*n) = (a\hat{}m)\ \hat{}\ n$
apply (*induct n*)
apply (*simp-all add*: *power-Suc power-add*)
done

lemma *power-mult-distrib*: $((a::'a::recpower) * b)\ \hat{}\ n = (a\hat{}n) * (b\hat{}n)$
apply (*induct n*)
apply (*auto simp add*: *power-Suc mult-ac*)
done

lemma *zero-less-power*:
$0 < (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 0 < a\hat{}n$
apply (*induct n*)
apply (*simp-all add*: *power-Suc zero-less-one mult-pos-pos*)
done

lemma *zero-le-power*:
$0 \leq (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 0 \leq a\hat{}n$
apply (*simp add*: *order-le-less*)
apply (*erule disjE*)
apply (*simp-all add*: *zero-less-power zero-less-one power-0-left*)
done

lemma *one-le-power*:
$1 \leq (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 1 \leq a\hat{}n$
apply (*induct n*)
apply (*simp-all add*: *power-Suc*)
apply (*rule order-trans* [*OF - mult-mono* [*of 1 - 1*]])
apply (*simp-all add*: *zero-le-one order-trans* [*OF zero-le-one*])
done

lemma *gt1-imp-ge0*: $1 < a ==> 0 \leq (a::'a::ordered\text{-}semidom)$
by (*simp add*: *order-trans* [*OF zero-le-one order-less-imp-le*])

lemma *power-gt1-lemma*:
assumes *gt1*: $1 < (a::'a::\{ordered\text{-}semidom,recpower\})$
shows $1 < a * a\hat{}n$
proof −
have $1*1 < a*1$ **using** *gt1* **by** *simp*
also have $\ldots \leq a * a\hat{}n$ **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 *power-gt1*:
$1 < (a::'a::\{ordered\text{-}semidom,recpower\}) ==> 1 < a\ \hat{}\ (Suc\ n)$
by (*simp add*: *power-gt1-lemma power-Suc*)

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 add*: *power-Suc*)
    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: *power-Suc 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-semidom*}) < *a*; *a*^*m* < *a*^*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* ≤ *b*; (*0*::'*a*::{*recpower*,*ordered-semidom*}) ≤ *a*|] ==> *a*^*n* ≤ *b*^*n*
apply (*induct n*)
apply (*simp-all add*: *power-Suc*)
apply (*auto intro*: *mult-mono zero-le-power order-trans* [*of 0 a b*])
done

lemma *power-strict-mono* [*rule-format*]:
    [|*a* < *b*; (*0*::'*a*::{*recpower*,*ordered-semidom*}) ≤ *a*|]
     ==> *0* < *n* --> *a*^*n* < *b*^*n*
apply (*induct n*)
apply (*auto simp add*: *mult-strict-mono zero-le-power power-Suc*
                  order-le-less-trans [*of 0 a b*])

done

lemma *power-eq-0-iff* [*simp*]:
    $(a\hat{}n = 0) = (a = (0::'a::\{ordered\text{-}idom,recpower\}) \ \&\ 0<n)$
apply (*induct n*)
apply (*auto simp add*: *power-Suc zero-neq-one* [*THEN not-sym*])
done

lemma *field-power-eq-0-iff* [*simp*]:
    $(a\hat{}n = 0) = (a = (0::'a::\{field,recpower\}) \ \&\ 0<n)$
apply (*induct n*)
apply (*auto simp add*: *power-Suc field-mult-eq-0-iff zero-neq-one*[*THEN not-sym*])
done

lemma *field-power-not-zero*: $a \neq (0::'a::\{field,recpower\}) ==> a\hat{}n \neq 0$
by *force*

lemma *nonzero-power-inverse*:
  $a \neq 0 ==> inverse\ ((a::'a::\{field,recpower\})\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
apply (*induct n*)
apply (*auto simp add*: *power-Suc nonzero-inverse-mult-distrib mult-commute*)
done

Perhaps these should be simprules.

lemma *power-inverse*:
  $inverse\ ((a::'a::\{field,division\text{-}by\text{-}zero,recpower\})\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
apply (*induct n*)
apply (*auto simp add*: *power-Suc inverse-mult-distrib*)
done

lemma *power-one-over*: $1\ /\ (a::'a::\{field,division\text{-}by\text{-}zero,recpower\})\hat{}n =$
    $(1\ /\ a)\hat{}n$
apply (*simp add*: *divide-inverse*)
apply (*rule power-inverse*)
done

lemma *nonzero-power-divide*:
    $b \neq 0 ==> (a/b)\ \hat{}\ n = ((a::'a::\{field,recpower\})\ \hat{}\ n)\ /\ (b\ \hat{}\ n)$
by (*simp add*: *divide-inverse power-mult-distrib nonzero-power-inverse*)

lemma *power-divide*:
    $(a/b)\ \hat{}\ n = ((a::'a::\{field,division\text{-}by\text{-}zero,recpower\})\ \hat{}\ n\ /\ b\ \hat{}\ n)$
apply (*case-tac b=0*, *simp add*: *power-0-left*)
apply (*rule nonzero-power-divide*)
apply *assumption*
done

lemma *power-abs*: $abs(a\ \hat{}\ n) = abs(a::'a::\{ordered\text{-}idom,recpower\})\ \hat{}\ n$
apply (*induct n*)

apply (*auto simp add*: *power-Suc abs-mult*)
done

lemma *zero-less-power-abs-iff* [*simp*]:
(*0* < (*abs a*) ^ *n*) = (*a* ≠ (*0*::'*a*::{*ordered-idom*,*recpower*}) | *n*=*0*)
proof (*induct n*)
case *0*
show *?case* **by** (*simp add*: *zero-less-one*)
next
case (*Suc n*)
show *?case* **by** (*force simp add*: *prems power-Suc zero-less-mult-iff*)
qed

lemma *zero-le-power-abs* [*simp*]:
(*0*::'*a*::{*ordered-idom*,*recpower*}) ≤ (*abs a*) ^ *n*
apply (*induct n*)
apply (*auto simp add*: *zero-le-one zero-le-power*)
done

lemma *power-minus*: (−*a*) ^ *n* = (− *1*) ^ *n* * (*a*::'*a*::{*comm-ring-1*,*recpower*}) ^ *n*
proof −
have −*a* = (− *1*) * *a* **by** (*simp add*: *minus-mult-left* [*symmetric*])
thus *?thesis* **by** (*simp only*: *power-mult-distrib*)
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*: *power-Suc 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 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*: *zero-le-power zero-less-one 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*: *power-Suc le-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac a * a^m ≤ 1 * a^n, simp*)
apply (*rule mult-mono*)
apply (*auto simp add*: *zero-le-power zero-le-one*)
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*: *power-Suc 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*] *zero-le-power*)
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*: *power-Suc 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 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*] *zero-le-power*
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*]:
  $1 < (b::'a::\{ordered\text{-}semidom,recpower\}) ==> (b \hat{} x < b \hat{} y) = (x < y)$
by (*blast intro*: *power-less-imp-less-exp power-strict-increasing*)

lemma *power-le-imp-le-base*:
assumes *le*: $a \hat{} Suc\ n \leq b \hat{} Suc\ n$
  and *xnonneg*: $(0::'a::\{ordered\text{-}semidom,recpower\}) \leq a$
  and *ynonneg*: $0 \leq b$
shows $a \leq b$
proof (*rule ccontr*)
assume ~ $a \leq b$
then have $b < a$ **by** (*simp only*: *linorder-not-le*)
then have $b \hat{} Suc\ n < a \hat{} Suc\ n$
by (*simp only*: *prems power-strict-mono*)
from *le* **and** *this* **show** *False*
by (*simp add*: *linorder-not-less* [*symmetric*])
qed

lemma *power-inject-base*:
  $[|\ a \hat{} Suc\ n = b \hat{} Suc\ n;\ 0 \leq a;\ 0 \leq b\ |]$
  $==> a = (b::'a::\{ordered\text{-}semidom,recpower\})$
by (*blast intro*: *power-le-imp-le-base order-antisym order-eq-refl sym*)

20.2 Exponentiation for the Natural Numbers

primrec (*power*)
$p \hat{} 0 = 1$
$p \hat{} (Suc\ n) = (p::nat) * (p \hat{} n)$

instance *nat* :: *recpower*
proof
fix *z n* :: *nat*
show $z\hat{}0 = 1$ **by** *simp*
show $z\hat{}(Suc\ n) = z * (z\hat{}n)$ **by** *simp*
qed

lemma *nat-one-le-power* [*simp*]: $1 \leq i ==> Suc\ 0 \leq i\hat{}n$
by (*insert one-le-power* [*of i n*], *simp*)

lemma *le-imp-power-dvd*: !!$i::nat.\ m \leq n ==> i\hat{}m\ dvd\ i\hat{}n$
apply (*unfold dvd-def*)
apply (*erule linorder-not-less* [*THEN iffD2*, *THEN add-diff-inverse*, *THEN subst*])
apply (*simp add*: *power-add*)
done

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*: !!$i::nat.$ $[|\ 0 < i;\ i\hat{}m < i\hat{}n\ |] ==> m < n$
apply (*rule ccontr*)

apply (*drule leI* [*THEN le-imp-power-dvd*, *THEN dvd-imp-le*, *THEN leD*])
apply (*erule zero-less-power*, *auto*)
done

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

lemma *power-le-dvd* [*rule-format*]: $k\hat{}j$ *dvd* $n \longrightarrow i \leq j \longrightarrow k\hat{}i$ *dvd* $(n::nat)$
apply (*induct j*)
apply (*simp-all add*: *le-Suc-eq*)
apply (*blast dest*!: *dvd-mult-right*)
done

lemma *power-dvd-imp-le*: $[|i\hat{}m$ *dvd* $i\hat{}n;\ (1::nat) < i|] \Longrightarrow m \leq n$
apply (*rule power-le-imp-le-exp*, *assumption*)
apply (*erule dvd-imp-le*, *simp*)
done

lemma *power-diff*:
  assumes *nz*: $a \neq 0$
  shows $n <= m \Longrightarrow (a::'a::\{recpower, field\})\ \hat{}\ (m-n) = (a\hat{}m) / (a\hat{}n)$
  by (*induct m n rule*: *diff-induct*)
    (*simp-all add*: *power-Suc nonzero-mult-divide-cancel-left nz*)

ML bindings for the general exponentiation theorems

ML
⟪
val power-0 = thmpower-0;
val power-Suc = thmpower-Suc;
val power-0-Suc = thmpower-0-Suc;
val power-0-left = thmpower-0-left;
val power-one = thmpower-one;
val power-one-right = thmpower-one-right;
val power-add = thmpower-add;
val power-mult = thmpower-mult;
val power-mult-distrib = thmpower-mult-distrib;
val zero-less-power = thmzero-less-power;
val zero-le-power = thmzero-le-power;
val one-le-power = thmone-le-power;
val gt1-imp-ge0 = thmgt1-imp-ge0;
val power-gt1-lemma = thmpower-gt1-lemma;
val power-gt1 = thmpower-gt1;
val power-le-imp-le-exp = thmpower-le-imp-le-exp;
val power-inject-exp = thmpower-inject-exp;
val power-less-imp-less-exp = thmpower-less-imp-less-exp;
val power-mono = thmpower-mono;
val power-strict-mono = thmpower-strict-mono;
val power-eq-0-iff = thmpower-eq-0-iff;
val field-power-eq-0-iff = thmfield-power-eq-0-iff;

val field-power-not-zero = thmfield-power-not-zero;
val power-inverse = thmpower-inverse;
val nonzero-power-divide = thmnonzero-power-divide;
val power-divide = thmpower-divide;
val power-abs = thmpower-abs;
val zero-less-power-abs-iff = thmzero-less-power-abs-iff;
val zero-le-power-abs = thm zero-le-power-abs;
val power-minus = thmpower-minus;
val power-Suc-less = thmpower-Suc-less;
val power-strict-decreasing = thmpower-strict-decreasing;
val power-decreasing = thmpower-decreasing;
val power-Suc-less-one = thmpower-Suc-less-one;
val power-increasing = thmpower-increasing;
val power-strict-increasing = thmpower-strict-increasing;
val power-le-imp-le-base = thmpower-le-imp-le-base;
val power-inject-base = thmpower-inject-base;
⟩⟩

ML bindings for the remaining theorems

ML
⟨⟨
val nat-one-le-power = thmnat-one-le-power;
val le-imp-power-dvd = thmle-imp-power-dvd;
val nat-power-less-imp-less = thmnat-power-less-imp-less;
val nat-zero-less-power-iff = thmnat-zero-less-power-iff;
val power-le-dvd = thmpower-le-dvd;
val power-dvd-imp-le = thmpower-dvd-imp-le;
⟩⟩

end

21 Finite-Set: Finite sets

theory *Finite-Set*
imports *Power Inductive Lattice-Locales*
begin

21.1 Definition and basic properties

consts *Finites* :: *'a set set*
syntax
finite :: *'a set => bool*
translations
finite A == A : Finites

inductive *Finites*
intros

emptyI [*simp*, *intro*!]: {} : *Finites*
insertI [*simp*, *intro*!]: *A* : *Finites* ==> *insert a A* : *Finites*

axclass *finite* ⊆ *type*
finite: *finite UNIV*

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 *prems* **have** *A* ≠ *UNIV* **by** *blast*
thus *?thesis* **by** *blast*
qed

lemma *finite-induct* [*case-names empty insert*, *induct set*: *Finites*]:
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* {} **.**
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 insert(4) by simp
  next
    assume F ≠ {} thus ?thesis using insert by blast
  qed
qed

lemma finite-subset-induct [consumes 2, case-names empty insert]:
  finite F ==> F ⊆ A ==>
    P {} ==> (!!a F. finite F ==> a ∈ A ==> a ∉ F ==> P F ==> P (insert
a F)) ==>
    P F
proof −
  assume P {} and insert:
    !!a F. finite F ==> a ∈ A ==> a ∉ F ==> P F ==> P (insert a F)
  assume finite F
  thus F ⊆ A ==> P F
  proof induct
    show P {} .
    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 .
    qed
  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 ∃ (n::nat) f. A = f ` {i. i<n} & inj-on f {i. i<n}
using fin
proof induct
  case empty
  show ?case
  proof show ∃f. {} = f ` {i::nat. i < 0} & inj-on f {i. i<0} by simp
  qed
next
  case (insert a A)
  have notinA: a ∉ A .
  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*} ⟹ *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 ∃ *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*)

21.1.1 Finiteness and set theoretic constructions

lemma *finite-UnI*: *finite F* ==> *finite G* ==> *finite* (*F Un G*)
— The union of two finite sets is finite.
by (*induct set*: *Finites*) *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*}) **.**
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 by (rule insert-Diff)
      finally show ?thesis .
    next
      show A ⊆ F ==> ?thesis .
      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-Int [simp, intro]: finite F | finite G ==> finite (F Int G)
  — The converse obviously fails.
  by (blast intro: finite-subset)

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:
  finite A ==>
  P A ==> (!!a A. finite A ==> a:A ==> P A ==> P (A - {a})) ==> P {}
proof -
  assume finite A
    and P A and !!a A. finite A ==> a:A ==> P A ==> P (A - {a})
  have P (A - A)
  proof -
    fix c b :: 'a set
    presume c: finite c and b: finite b
     and P1: P b and P2: !!x y. finite y ==> x ∈ y ==> P y ==> P (y - {x})
    from c show c ⊆ b ==> P (b - 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
  next
   show A ⊆ A ..
  qed
  thus P {} by simp
qed

lemma finite-Diff [simp]: finite B ==> finite (B − Ba)
  by (rule Diff-subset [THEN finite-subset])

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: Finites) 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)
  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*: *Finites*, *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*: *Finites*) *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-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-UnionD*: *finite*(⋃ *A*) ⟹ *finite A*

by(*blast intro*: *finite-subset*[*OF subset-Pow-Union*])

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*: *Finites*)
apply (*auto simp add*: *Field-def Domain-insert Range-insert*)
done

lemma *trancl-subset-Field2*: *r*^+ <= *Field r* × *Field r*
apply *clarify*
apply (*erule trancl-induct*)
apply (*auto simp add*: *Field-def*)
done

lemma *finite-trancl*: *finite* (*r*^+) = *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

21.2 A fold functional for finite sets

The intended behaviour is *fold f g z* $\{x_1, ..., x_n\} = f\ (g\ x_1)\ (\ldots\ (f\ (g\ x_n)\ z)\ldots)$ if f is associative-commutative. For an application of *fold* se the definitions of sums and products over finite sets.

consts
foldSet :: (′*a* => ′*a* => ′*a*) => (′*b* => ′*a*) => ′*a* => (′*b set* × ′*a*) *set*

inductive *foldSet f g z*
intros
emptyI [*intro*]: $(\{\}, z) : \mathit{foldSet}\ f\ g\ z$
insertI [*intro*]:
$\llbracket\ x \notin A;\ (A, y) : \mathit{foldSet}\ f\ g\ z\ \rrbracket$
$\Longrightarrow (\mathit{insert}\ x\ A, f\ (g\ x)\ y) : \mathit{foldSet}\ f\ g\ z$

inductive-cases *empty-foldSetE* [*elim*!]: $(\{\}, x) : \mathit{foldSet}\ f\ g\ z$

constdefs
fold :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ \mathit{set} \Rightarrow 'a$
$\mathit{fold}\ f\ g\ z\ A == \mathit{THE}\ x.\ (A, x) : \mathit{foldSet}\ f\ g\ z$

A tempting alternative for the definiens is *if finite A then THE x.* $(A, x) \in$ *foldSet f g e else e*. It allows the removal of finiteness assumptions from the theorems *fold-commute*, *fold-reindex* and *fold-distrib*. The proofs become ugly, with *rule-format*. It is not worth the effort.

lemma *Diff1-foldSet*:
$(A - \{x\}, y) : \mathit{foldSet}\ f\ g\ z \Longrightarrow x \in A \Longrightarrow (A, f\ (g\ x)\ y) : \mathit{foldSet}\ f\ g\ z$
by (*erule insert-Diff* [*THEN subst*], *rule foldSet.intros*, *auto*)

lemma *foldSet-imp-finite*: $(A, x) : \mathit{foldSet}\ f\ g\ z \Longrightarrow \mathit{finite}\ A$
by (*induct set*: *foldSet*) *auto*

lemma *finite-imp-foldSet*: $\mathit{finite}\ A \Longrightarrow \mathit{EX}\ x.\ (A, x) : \mathit{foldSet}\ f\ g\ z$
by (*induct set*: *Finites*) *auto*

21.2.1 Commutative monoids

locale *ACf* =
fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\cdot$ *70*)
assumes *commute*: $x \cdot y = y \cdot x$
and *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

locale *ACe* = *ACf* +
fixes $e :: 'a$
assumes *ident* [*simp*]: $x \cdot e = x$

locale *ACIf* = *ACf* +
assumes *idem*: $x \cdot x = x$

lemma (**in** *ACf*) *left-commute*: $x \cdot (y \cdot z) = y \cdot (x \cdot z)$
proof −
have $x \cdot (y \cdot z) = (y \cdot z) \cdot x$ **by** (*simp only*: *commute*)
also have $\ldots = y \cdot (z \cdot x)$ **by** (*simp only*: *assoc*)
also have $z \cdot x = x \cdot z$ **by** (*simp only*: *commute*)
finally show *?thesis* **.**
qed

lemmas (**in** *ACf*) *AC* = *assoc commute left-commute*

lemma (**in** *ACe*) *left-ident* [*simp*]: $e \cdot x = x$
proof −
have $x \cdot e = x$ **by** (*rule ident*)
thus *?thesis* **by** (*subst commute*)
qed

lemma (**in** *ACIf*) *idem2*: $x \cdot (x \cdot y) = x \cdot y$
proof −
have $x \cdot (x \cdot y) = (x \cdot x) \cdot y$ **by**(*simp add*:*assoc*)
also have $\ldots = x \cdot y$ **by**(*simp add*:*idem*)
finally show *?thesis* **.**
qed

lemmas (**in** *ACIf*) *ACI* = *AC idem idem2*

Interpretation of locales:

interpretation *AC-add*: *ACe* [*op* + *0*::*'a*::*comm-monoid-add*]
by(*auto intro*: *ACf.intro ACe-axioms.intro add-assoc add-commute*)

interpretation *AC-mult*: *ACe* [*op* ∗ *1*::*'a*::*comm-monoid-mult*]
apply −
apply (*fast intro*: *ACf.intro mult-assoc mult-commute*)
apply (*fastsimp intro*: *ACe-axioms.intro mult-assoc mult-commute*)
done

21.2.2 From *foldSet* 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*:
$[|insert\ (h\ m)\ A = h$ ‘ $\{i.\ i < Suc\ m\};\ h\ m \notin A;$
$inj\text{-}on\ h\ \{i.\ i < Suc\ m\}|]$
$==> A = h$ ‘ $\{i.\ i < m\}$
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*: $a \notin A$
and *inj-on*: $inj\text{-}on\ h\ \{i$::*nat*. $i<n\}$
shows $\exists hm\ m.\ inj\text{-}on\ hm\ \{i$::*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*: $n = Suc\ m$ **.**

```
  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 = 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 (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

lemma (in ACf) foldSet-determ-aux:
  !!A x x' h. ⟦ A = h`{i::nat. i<n}; inj-on h {i. i<n};
                (A,x) : foldSet f g z; (A,x') : foldSet f g z ⟧
   ⟹ x' = x
proof (induct n 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};
               (A,x) ∈ foldSet f g z; (A, x') ∈ foldSet f g z⟧ ⟹ x' = x .
    have Afoldx: (A,x) ∈ foldSet f g z and Afoldx': (A,x') ∈ foldSet f g z
     and A: A = h`{i. i<n} and injh: inj-on h {i. i<n} .
    show ?case
    proof (rule foldSet.cases [OF Afoldx])
      assume (A, x) = ({}, z)
      with Afoldx' show x' = x by blast
    next
      fix B b u
      assume (A,x) = (insert b B, g b · u) and notinB: b ∉ B
        and Bu: (B,u) ∈ foldSet f g z
      hence AbB: A = insert b B and x: x = g b · u by auto
      show x'=x
      proof (rule foldSet.cases [OF Afoldx'])
        assume (A, x') = ({}, z)
        with AbB show x' = x by blast
      next
        fix C c v
```

```
      assume (A,x') = (insert c C, g c · v) and notinC: c ∉ C
        and Cv: (C,v) ∈ foldSet f g z
      hence AcC: A = insert c C and x': x' = g c · v by auto
      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 foldSet-imp-finite[OF Afoldx])
        with AbB have finite ?D by simp
        then obtain d where Dfoldd: (?D,d) ∈ foldSet f g z
          using finite-imp-foldSet by iprover
        moreover have cinB: c ∈ B using B by auto
        ultimately have (B,g c · d) ∈ foldSet f g z
          by(rule Diff1-foldSet)
        hence g c · d = u by (rule IH [OF lessB Beq inj-onB Bu])
        moreover have g b · d = v
        proof (rule IH[OF lessC Ceq inj-onC Cv])
          show (C, g b · d) ∈ foldSet f g z using C notinB Dfoldd
            by fastsimp
        qed
        ultimately show ?thesis using x x' by (auto simp: AC)
      qed
    qed
  qed
qed


lemma (in ACf) foldSet-determ:
  (A,x) : foldSet f g z ==> (A,y) : foldSet f g z ==> y = x
apply (frule foldSet-imp-finite [THEN finite-imp-nat-seg-image-inj-on])
apply (blast intro: foldSet-determ-aux [rule-format])
done
```

lemma (**in** *ACf*) *fold-equality*: (*A*, *y*) : *foldSet f g z* ==> *fold f g z A* = *y*
by (*unfold fold-def*) (*blast intro*: *foldSet-determ*)

The base case for *fold*:

lemma *fold-empty* [*simp*]: *fold f g z* {} = *z*
by (*unfold fold-def*) *blast*

lemma (**in** *ACf*) *fold-insert-aux*: *x* ∉ *A* ==>
((*insert x A*, *v*) : *foldSet f g z*) =
(*EX y*. (*A*, *y*) : *foldSet f g z* & *v* = *f* (*g x*) *y*)
apply *auto*
apply (*rule-tac A1* = *A* **and** *f1* = *f* **in** *finite-imp-foldSet* [*THEN exE*])
apply (*fastsimp dest*: *foldSet-imp-finite*)
apply (*blast intro*: *foldSet-determ*)
done

The recursion equation for *fold*:

lemma (**in** *ACf*) *fold-insert*[*simp*]:
finite A ==> *x* ∉ *A* ==> *fold f g z* (*insert x A*) = *f* (*g x*) (*fold f g z A*)
apply (*unfold fold-def*)
apply (*simp add*: *fold-insert-aux*)
apply (*rule the-equality*)
apply (*auto intro*: *finite-imp-foldSet*
cong add: *conj-cong simp add*: *fold-def* [*symmetric*] *fold-equality*)
done

lemma (**in** *ACf*) *fold-rec*:
assumes *fin*: *finite A* **and** *a*: *a*:*A*
shows *fold f g z A* = *f* (*g a*) (*fold f g z* (*A* − {*a*}))
proof−
have *A*: *A* = *insert a* (*A* − {*a*}) **using** *a* **by** *blast*
hence *fold f g z A* = *fold f g z* (*insert a* (*A* − {*a*})) **by** *simp*
also have ... = *f* (*g a*) (*fold f g z* (*A* − {*a*}))
by(*rule fold-insert*) (*simp add*:*fin*)+
finally show *?thesis* **.**
qed

A simplified version for idempotent functions:

lemma (**in** *ACIf*) *fold-insert-idem*:
assumes *finA*: *finite A*
shows *fold f g z* (*insert a A*) = *g a* · *fold f g z A*
proof *cases*
assume *a* ∈ *A*
then obtain *B* **where** *A*: *A* = *insert a B* **and** *disj*: *a* ∉ *B*
by(*blast dest*: *mk-disjoint-insert*)
show *?thesis*
proof −
from *finA A* **have** *finB*: *finite B* **by**(*blast intro*: *finite-subset*)

```
    have fold f g z (insert a A) = fold f g z (insert a B) using A by simp
    also have ... = (g a) · (fold f g z B)
      using finB disj by simp
    also have ... = g a · fold f g z A
      using A finB disj by(simp add:idem assoc[symmetric])
    finally show ?thesis .
  qed
next
  assume a ∉ A
  with finA show ?thesis by simp
qed

lemma (in ACIf) foldI-conv-id:
  finite A ⟹ fold f g z A = fold f id z (g ` A)
by(erule finite-induct)(simp-all add: fold-insert-idem del: fold-insert)
```

21.2.3 Lemmas about *fold*

```
lemma (in ACf) fold-commute:
  finite A ==> (!!z. f x (fold f g z A) = fold f g (f x z) A)
  apply (induct set: Finites, simp)
  apply (simp add: left-commute [of x])
  done

lemma (in ACf) fold-nest-Un-Int:
  finite A ==> finite B
    ==> fold f g (fold f g z B) A = fold f g (fold f g z (A Int B)) (A Un B)
  apply (induct set: Finites, simp)
  apply (simp add: fold-commute Int-insert-left insert-absorb)
  done

lemma (in ACf) fold-nest-Un-disjoint:
  finite A ==> finite B ==> A Int B = {}
    ==> fold f g z (A Un B) = fold f g (fold f g z B) A
  by (simp add: fold-nest-Un-Int)

lemma (in ACf) fold-reindex:
assumes fin: finite A
shows inj-on h A ⟹ fold f g z (h ` A) = fold f (g ∘ h) z A
using fin apply induct
 apply simp
apply simp
done

lemma (in ACe) fold-Un-Int:
  finite A ==> finite B ==>
    fold f g e A · fold f g e B =
    fold f g e (A Un B) · fold f g e (A Int B)
  apply (induct set: Finites, simp)
```

apply (*simp add*: *AC insert-absorb Int-insert-left*)
done

corollary (**in** *ACe*) *fold-Un-disjoint*:
finite A ==> finite B ==> A Int B = {} ==>
fold f g e (A Un B) = fold f g e A · fold f g e B
by (*simp add*: *fold-Un-Int*)

lemma (**in** *ACe*) *fold-UN-disjoint*:
⟦ *finite I*; *ALL i:I. finite (A i)*;
ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {} ⟧
⟹ *fold f g e (UNION I A) =*
fold f (%i. fold f g e (A i)) e I
apply (*induct set*: *Finites*, *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

Fusion theorem, as described in Graham Hutton's paper, A Tutorial on the Universality and Expressiveness of Fold, JFP 9:4 (355-372), 1999.

lemma (**in** *ACf*) *fold-fusion*:
includes *ACf g*
shows
finite A ==>
(!!x y. h (g x y) = f x (h y)) ==>
h (fold g j w A) = fold f j (h w) A
by (*induct set*: *Finites*, *simp-all*)

lemma (**in** *ACf*) *fold-cong*:
finite A ⟹ *(!!x. x:A ==> g x = h x) ==> fold f g z A = fold f h z A*
apply (*subgoal-tac ALL C. C <= A --> (ALL x:C. g x = h x) --> fold f g z C = fold f 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

lemma (**in** *ACe*) *fold-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*

fold f (%x. fold f (g x) e (B x)) e A =
fold f (split g) e (SIGMA x:A. B x)
apply (*subst Sigma-def*)
apply (*subst fold-UN-disjoint*, *assumption*, *simp*)
apply *blast*
apply (*erule fold-cong*)
apply (*subst fold-UN-disjoint*, *simp*, *simp*)
apply *blast*
apply *simp*
done

lemma (**in** *ACe*) *fold-distrib*: *finite A* $\Longrightarrow$
fold f (%x. f (g x) (h x)) e A = f (fold f g e A) (fold f h e A)
apply (*erule finite-induct*, *simp*)
apply (*simp add*:*AC*)
done

21.3 Generalized summation over a set

constdefs
setsum :: *('a => 'b) => 'a set => 'b*::*comm-monoid-add*
setsum f A == if finite A then fold (op +) f 0 A else 0

Now: lot's of fancy syntax. First, *setsum* $(\lambda x.\ e)\ A$ is written $\sum x{\in}A.\ e$.

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$\sum$-$\in$-. -) [0, 51, 10] 10*)
syntax (*HTML* **output**)
-setsum :: *pttrn => 'a set => 'b => 'b*::*comm-monoid-add* (*(3$\sum$-$\in$-. -) [0, 51, 10] 10*)

translations — Beware of argument permutation!
SUM i:A. b == setsum (%i. b) A
$\sum i{\in}A.\ b$ == *setsum (%i. b) A*

Instead of $\sum x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\sum x|P.\ e$.

syntax
-qsetsum :: *pttrn* $\Rightarrow$ *bool* $\Rightarrow$ *'a* $\Rightarrow$ *'a* (*(3SUM - |/ -./ -) [0,0,10] 10*)
syntax (*xsymbols*)
-qsetsum :: *pttrn* $\Rightarrow$ *bool* $\Rightarrow$ *'a* $\Rightarrow$ *'a* (*(3$\sum$ - | (-)./ -) [0,0,10] 10*)
syntax (*HTML* **output**)
-qsetsum :: *pttrn* $\Rightarrow$ *bool* $\Rightarrow$ *'a* $\Rightarrow$ *'a* (*(3$\sum$ - | (-)./ -) [0,0,10] 10*)

translations
SUM x|P. t => setsum (%x. t) {x. P}
$\sum x|P.\ t$ => *setsum (%x. t) {x. P}*

Finally we abbreviate $\sum x{\in}A.\ x$ by $\sum A$.

syntax
-Setsum :: *'a set => 'a::comm-monoid-mult* ($\sum$ - [*1000*] *999*)

parse-translation ⟪
let
fun Setsum-tr [*A*] = *Syntax.const setsum* $ *Abs* (, *dummyT*, *Bound 0*) $ *A*
in [(*-Setsum*, *Setsum-tr*)] *end*;
⟫

print-translation ⟪
let
fun setsum-tr' [*Abs*(-,-,*Bound 0*), *A*] = *Syntax.const -Setsum* $ *A*
| *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 AC-add.fold-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-cong*:
A = *B ==>* (!!*x*. *x*:*B ==> f x* = *g x*) *==> setsum f A = setsum g B*
by(*fastsimp simp*: *setsum-def intro*: *AC-add.fold-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*: *AC-add.fold-cong*)

lemma *setsum-cong2*: $[\![\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x]\!] \Longrightarrow setsum\ f\ A = setsum\ g\ A$
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 AC-add.fold-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-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 AC-add.fold-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*) ==>
(∑ *x*∈*A*. (∑ *y*∈*B x*. *f x y*)) = (∑ (*x*,*y*)∈(*SIGMA x*:*A*. *B x*). *f x y*)

by(*simp add*:*setsum-def AC-add.fold-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 AC-add.fold-distrib*)

21.3.1 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*: *Finites*) *auto*

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*: *ring-eq-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*: *ring-eq-simps*)

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-group-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* ∈ *A* ⟶ (∑ *x* ∈ *A*. *f x*) = *f a* + (∑ *x* ∈ (*A* − {*a*}). *f x*)
apply (*erule finite-induct*[**where** *F*=*A* **and** *P*=% *A*. (*a* ∈ *A* ⟶ (∑ *x* ∈ *A*. *f x*) = *f a* + (∑ *x* ∈ (*A* − {*a*}). *f x*))])
apply (*auto simp add*: *insert-Diff-if add-ac*)
done

lemma *setsum-diff-nat*:
assumes *finB*: *finite B*
shows *B* ⊆ *A* $\Longrightarrow$ (*setsum f* (*A* − *B*) :: *nat*) = (*setsum f A*) − (*setsum f B*)
using *finB*
proof (*induct*)
show *setsum f* (*A* − {}) = (*setsum f A*) − (*setsum f* {}) **by** *simp*
next
fix *F x* **assume** *finF*: *finite F* **and** *xnotinF*: *x* ∉ *F*
and *xFinA*: *insert x F* ⊆ *A*
and *IH*: *F* ⊆ *A* $\Longrightarrow$ *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 force
  qed
next
  case False
  thus ?thesis
    by (simp add: setsum-def)
qed

lemma setsum-strict-mono:
fixes f :: 'a ⇒ 'b::{pordered-cancel-ab-semigroup-add,comm-monoid-add}
assumes fin-ne: finite A  A ≠ {}
shows (!!x. x:A ⟹ f x < g x) ⟹ setsum f A < setsum g A
using fin-ne
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: Finites, 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*: $\forall x{\in}A.\ (0::'a::\{pordered\text{-}ab\text{-}semigroup\text{-}add,comm\text{-}monoid\text{-}add\}) \leq f\ x$
shows $0 \leq setsum\ f\ A$
proof (*cases finite A*)
case *True* **thus** *?thesis* **using** *nn*
apply (*induct set*: *Finites*, *auto*)
apply (*subgoal-tac* $0 + 0 \leq f\ x + setsum\ f\ F$, *simp*)
apply (*blast intro*: *add-mono*)
done
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*
apply (*induct set*: *Finites*, *auto*)
apply (*subgoal-tac* $f\ x + setsum\ f\ F \leq 0 + 0$, *simp*)
apply (*blast intro*: *add-mono*)
done
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-mult:
  fixes f :: 'a => ('b::semiring-0-cancel)
  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-0-cancel) = (\<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) = (∑ n∈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*::*lordered-ab-group-abs*)
shows *abs* (*setsum f A*) ≤ *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*::*lordered-ab-group-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 intro*: *order-trans*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *abs-setsum-abs*[*simp*]:
fixes *f* :: *'a* => (*'b*::*lordered-ab-group-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 $|\sum a{\in}insert\ a\ A.\ |f\ a|| = ||f\ a| + (\sum a{\in}A.\ |f\ a|)|$ **by** *simp*
also have $\ldots = ||f\ a| + |\sum a{\in}A.\ |f\ a|||$ **using** *insert* **by** *simp*
also have $\ldots = |f\ a| + |\sum a{\in}A.\ |f\ a||$
by (*simp del*: *abs-of-nonneg*)
also have $\ldots = (\sum a{\in}insert\ a\ A.\ |f\ a|)$ **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 $(\%(i,\ j).\ (j,\ i))\ (A \times B)$
by (*unfold inj-on-def*) *fast*

lemma *swap-product*:
$(\%(i,\ j).\ (j,\ i))\ `\ (A \times B) = B \times A$
by (*simp add*: *split-def image-def*) *blast*

lemma *setsum-commute*:
$(\sum i{\in}A.\ \sum j{\in}B.\ f\ i\ j) = (\sum j{\in}B.\ \sum i{\in}A.\ f\ i\ j)$
proof (*simp add*: *setsum-cartesian-product*)
have $(\sum (x,y) \in A <*> B.\ f\ x\ y) =$
$(\sum (y,x) \in (\%(i,\ j).\ (j,\ i))\ `\ (A \times B).\ f\ x\ y)$
(**is** *?s* = -)
apply (*simp add*: *setsum-reindex* [**where** $f = \%(i,\ j).\ (j,\ i)$] *swap-inj-on*)
apply (*simp add*: *split-def*)
done
also have ... $= (\sum (y,x){\in}B \times A.\ f\ x\ y)$
(**is** - = *?t*)
apply (*simp add*: *swap-product*)
done
finally show *?s* = *?t* **.**
qed

21.4 Generalized product over a set

constdefs
setprod :: $('a => 'b) => 'a\ set => 'b{::}comm\text{-}monoid\text{-}mult$
setprod f A == *if finite A then fold* $(op *)$ *f 1 A else 1*

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*$\prod$ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*$\prod$ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
PROD i:*A*. *b* == *setprod* (%*i*. *b*) *A*
$\prod$ *i*∈*A*. *b* == *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*$\prod$ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*$\prod$ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations
PROD x|*P*. *t* => *setprod* (%*x*. *t*) {*x*. *P*}
$\prod x|P.\ t$ => *setprod* (%*x*. *t*) {*x*. *P*}

Finally we abbreviate $\prod x{\in}A.\ x$ by $\prod A$.

syntax
-Setprod :: *'a set* => *'a*::*comm-monoid-mult* ($\prod$ - [*1000*] *999*)

parse-translation ⟪
let
fun Setprod-tr [*A*] = *Syntax.const setprod* $ *Abs* (, *dummyT*, *Bound 0*) $ *A*
in [(*-Setprod*, *Setprod-tr*)] *end*;
⟫
print-translation ⟪
let fun setprod-tr' [*Abs*(*x*,*Tx*,*t*), *A*] =
if t = *Bound 0 then Syntax.const -Setprod* $ *A else raise Match*
in
[(*setprod*, *setprod-tr'*)]
end
⟫

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 AC-mult.fold-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*: *AC-mult.fold-cong*)

lemma *strong-setprod-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*: *AC-mult.fold-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 *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 AC-mult.fold-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-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 AC-mult.fold-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 AC-mult.fold-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 AC-mult.fold-distrib*)

21.4.1 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*: *Finites*) *auto*

lemma *setprod-zero*:
finite A ==> EX x: *A. f x = (0*::*'a*::*comm-semiring-1-cancel*) ==> *setprod f A = 0*
apply (*induct set*: *Finites*, *force*, *clarsimp*)
apply (*erule disjE*, *auto*)
done

lemma *setprod-nonneg* [*rule-format*]:
(*ALL x*: *A*. (*0*::*'a*::*ordered-idom*) ≤ *f x*) --> *0* ≤ *setprod f A*
apply (*case-tac finite A*)
apply (*induct set*: *Finites*, *force*, *clarsimp*)
apply (*subgoal-tac 0* ∗ *0* ≤ *f x* ∗ *setprod f F*, *force*)

apply (*rule mult-mono*, *assumption*+)
apply (*auto simp add*: *setprod-def*)
done

lemma *setprod-pos* [*rule-format*]: (*ALL x*: *A*. (*0*::'*a*::*ordered-idom*) < *f x*)
−−> *0* < *setprod f A*
apply (*case-tac finite A*)
apply (*induct set*: *Finites*, *force*, *clarsimp*)
apply (*subgoal-tac 0* ∗ *0* < *f x* ∗ *setprod f F*, *force*)
apply (*rule mult-strict-mono*, *assumption*+)
apply (*auto simp add*: *setprod-def*)
done

lemma *setprod-nonzero* [*rule-format*]:
(*ALL x y*. (*x*::'*a*::*comm-semiring-1-cancel*) ∗ *y* = *0* −−> *x* = *0* | *y* = *0*) ==>
finite A ==> (*ALL x*: *A*. *f x* ≠ (*0*::'*a*)) −−> *setprod f A* ≠ *0*
apply (*erule finite-induct*, *auto*)
done

lemma *setprod-zero-eq*:
(*ALL x y*. (*x*::'*a*::*comm-semiring-1-cancel*) ∗ *y* = *0* −−> *x* = *0* | *y* = *0*) ==>
finite A ==> (*setprod f A* = (*0*::'*a*)) = (*EX x*: *A*. *f x* = *0*)
apply (*insert setprod-zero* [*of A f*] *setprod-nonzero* [*of A f*], *blast*)
done

lemma *setprod-nonzero-field*:
finite A ==> (*ALL x*: *A*. *f x* ≠ (*0*::'*a*::*field*)) ==> *setprod f A* ≠ *0*
apply (*rule setprod-nonzero*, *auto*)
done

lemma *setprod-zero-eq-field*:
finite A ==> (*setprod f A* = (*0*::'*a*::*field*)) = (*EX x*: *A*. *f x* = *0*)
apply (*rule setprod-zero-eq*, *auto*)
done

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*)
apply (*subst setprod-Un-Int* [*symmetric*], *auto*)
apply (*subgoal-tac finite* (*A Int B*))
apply (*frule setprod-nonzero-field* [*of A Int B f*], *assumption*)
apply (*subst times-divide-eq-right* [*THEN sym*], *auto simp add*: *divide-self*)
done

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*)
apply (*erule finite-induct*)
apply (*auto simp add*: *insert-Diff-if*)

apply (*subgoal-tac f a * setprod f F / f a = setprod f F * f a / f a*)
apply (*erule ssubst*)
apply (*subst times-divide-eq-right* [*THEN sym*])
apply (*auto simp add*: *mult-ac times-divide-eq-right divide-self*)
done

lemma *setprod-inversef*: *finite A* ==>
ALL x: *A*. *f x* ≠ (*0*::'*a*::{*field*,*division-by-zero*}) ==>
setprod (*inverse* ∘ *f*) *A* = *inverse* (*setprod f A*)
apply (*erule finite-induct*)
apply (*simp*, *simp*)
done

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

21.5 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:

constdefs
card :: '*a set* => *nat*
card A == *setsum* (%*x*. *1*::*nat*) *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* ∉ *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*]: *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-insert*: *finite A ==> card (insert x A) = Suc (card (A - {x}))*
by (*simp add*: *card-insert-if card-Suc-Diff1*)

lemma *card-insert-le*: *finite A ==> card A <= card (insert x A)*
by (*simp add*: *card-insert-if*)

lemma *card-mono*: $[\![$ *finite B*; $A \subseteq B$ $]\!] \Longrightarrow$ *card A* $\leq$ *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*: *Finites*, *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-def 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*)
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*)
apply (*rule less-trans*)
prefer *2* **apply** (*auto intro*!: *card-Diff1-less*)
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*

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* - *⋃ (insert x F)*])
done

lemma *setsum-constant* [*simp*]: $(\sum x \in A.\ y) = \mathit{of\text{-}nat}(\mathit{card}\ A) * y$
apply (*cases finite A*)
apply (*erule finite-induct*)
apply (*auto simp add*: *ring-distrib add-ac*)
done

lemma *setprod-constant*: *finite A* ==> $(\prod x{\in} A.\ (y{::}'a{::}\mathit{recpower})) = y\hat{}(\mathit{card}\ A)$
apply (*erule finite-induct*)
apply (*auto simp add*: *power-Suc*)
done

lemma *setsum-bounded*:
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow f\,i \leq (K{::}'a{::}\{\mathit{comm\text{-}semiring\text{-}1\text{-}cancel}, \mathit{pordered\text{-}ab\text{-}semigroup\text{-}add}\})$
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

21.5.1 Cardinality of unions

lemma *of-nat-id*[*simp*]: (*of-nat n* :: *nat*) = *n*
by(*induct n*, *auto*)

lemma *card-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. *i* ≠ *j* −−> *A i Int A j* = {}) ==>
card (*UNION I A*) = $(\sum i{\in}I.\ \mathit{card}(A\ i))$
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

21.5.2 Cardinality of image

The image of a finite set can be expressed using *fold*.

lemma *image-eq-fold*: *finite A ==> f ' A = fold (op Un) (%x. {f x}) {} A*
apply (*erule finite-induct*, *simp*)
apply (*subst ACf.fold-insert*)
apply (*auto simp add*: *ACf-def*)
done

lemma *card-image-le*: *finite A ==> card (f ' A) <= card A*
apply (*induct set*: *Finites*, *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*, *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*)

21.5.3 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*)

21.5.4 Cardinality of the Powerset

lemma *card-Pow*: *finite A* ==> *card* (*Pow A*) = *Suc* (*Suc 0*) ^ *card A*
apply (*induct set*: *Finites*)
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's.

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*: *Finites*, *simp-all*, *clarify*)
apply (*subst card-Un-disjoint*)
apply (*auto simp add*: *dvd-add disjoint-eq-subset-Compl*)
done

21.6 A fold functional for non-empty sets

Does not require start value.

consts
fold1Set :: ('*a* => '*a* => '*a*) => ('*a set* × '*a*) *set*

inductive *fold1Set f*
intros
fold1Set-insertI [*intro*]:
⟦ (*A*,*x*) ∈ *foldSet f id a*; *a* ∉ *A* ⟧ ⟹ (*insert a A*, *x*) ∈ *fold1Set f*

constdefs
fold1 :: $('a \Rightarrow 'a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow 'a$
$fold1\ f\ A == THE\ x.\ (A, x) : fold1Set\ f$

lemma *fold1Set-nonempty*:
$(A, x) : fold1Set\ f \Longrightarrow A \neq \{\}$
by(*erule fold1Set.cases*, *simp-all*)

inductive-cases *empty-fold1SetE* [*elim*!]: $(\{\}, x) : fold1Set\ f$

inductive-cases *insert-fold1SetE* [*elim*!]: $(insert\ a\ X, x) : fold1Set\ f$

lemma *fold1Set-sing* [*iff*]: $((\{a\},b) : fold1Set\ f) = (a = b)$
by (*blast intro*: *foldSet.intros elim*: *foldSet.cases*)

lemma *fold1-singleton*[*simp*]: $fold1\ f\ \{a\} = a$
by (*unfold fold1-def*) *blast*

lemma *finite-nonempty-imp-fold1Set*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow EX\ x.\ (A, x) : fold1Set\ f$
apply (*induct A rule*: *finite-induct*)
apply (*auto dest*: *finite-imp-foldSet* [*of - f id*])
done

First, some lemmas about *foldSet*.

lemma (**in** *ACf*) *foldSet-insert-swap*:
assumes *fold*: $(A,y) \in foldSet\ f\ id\ b$
shows $b \notin A \Longrightarrow (insert\ b\ A, z \cdot y) \in foldSet\ f\ id\ z$
using *fold*
proof (*induct rule*: *foldSet.induct*)
case *emptyI* **thus** *?case* **by** (*force simp add*: *fold-insert-aux commute*)
next
case (*insertI A x y*)
have $(insert\ x\ (insert\ b\ A), x \cdot (z \cdot y)) \in foldSet\ f\ (\lambda u.\ u)\ z$
using *insertI* **by** *force* — how does *id* get unfolded?
thus *?case* **by** (*simp add*: *insert-commute AC*)
qed

lemma (**in** *ACf*) *foldSet-permute-diff*:
assumes *fold*: $(A,x) \in foldSet\ f\ id\ b$
shows $!!a.\ [\![a \in A;\ b \notin A]\!] \Longrightarrow (insert\ b\ (A-\{a\}), x) \in foldSet\ f\ id\ a$
using *fold*
proof (*induct rule*: *foldSet.induct*)
case *emptyI* **thus** *?case* **by** *simp*
next
case (*insertI A x y*)
have $a = x \vee a \in A$ **using** *insertI* **by** *simp*

```
  thus ?case
  proof
    assume a = x
    with insertI show ?thesis
      by (simp add: id-def [symmetric], blast intro: foldSet-insert-swap)
  next
    assume ainA: a ∈ A
    hence (insert x (insert b (A − {a})), x · y) ∈ foldSet f id a
      using insertI by (force simp: id-def)
    moreover
    have insert x (insert b (A − {a})) = insert b (insert x A − {a})
      using ainA insertI by blast
    ultimately show ?thesis by (simp add: id-def)
  qed
qed

lemma (in ACf) fold1-eq-fold:
    [|finite A; a ∉ A|] ==> fold1 f (insert a A) = fold f id a A
apply (simp add: fold1-def fold-def)
apply (rule the-equality)
apply (best intro: foldSet-determ theI dest: finite-imp-foldSet [of - f id])
apply (rule sym, clarify)
apply (case-tac Aa=A)
 apply (best intro: the-equality foldSet-determ)
apply (subgoal-tac (A,x) ∈ foldSet f id a)
 apply (best intro: the-equality foldSet-determ)
apply (subgoal-tac insert aa (Aa − {a}) = A)
 prefer 2 apply (blast elim: equalityE)
apply (auto dest: foldSet-permute-diff [where a=a])
done

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 (in ACf) fold1-insert:
  assumes nonempty: A ≠ {} and A: finite A x ∉ A
  shows fold1 f (insert x A) = f x (fold1 f A)
proof −
  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

lemma (in ACIf) fold1-insert-idem [simp]:
```

```
  assumes nonempty: A ≠ {} and A: finite A
  shows fold1 f (insert x A) = f x (fold1 f A)
proof −
  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: idem)
    next
      assume A' ≠ {}
      with prems show ?thesis
        by (simp add: fold1-insert assoc [symmetric] idem)
    qed
  next
    assume a ≠ x
    with prems show ?thesis
      by (simp add: insert-commute fold1-eq-fold fold-insert-idem)
  qed
qed
```

Now the recursion rules for definitions:

```
lemma fold1-singleton-def: g ≡ fold1 f ⟹ g {a} = a
by(simp add:fold1-singleton)

lemma (in ACf) fold1-insert-def:
  ⟦ g ≡ fold1 f; finite A; x ∉ A; A ≠ {} ⟧ ⟹ g(insert x A) = x · (g A)
by(simp add:fold1-insert)

lemma (in ACIf) fold1-insert-idem-def:
  ⟦ g ≡ fold1 f; finite A; A ≠ {} ⟧ ⟹ g(insert x A) = x · (g A)
by(simp add:fold1-insert-idem)
```

21.6.1 Determinacy for *fold1Set*

Not actually used!!

```
lemma (in ACf) foldSet-permute:
  [|(insert a A, x) ∈ foldSet f id b; a ∉ A; b ∉ A|]
   ==> (insert b A, x) ∈ foldSet f id a
apply (case-tac a=b)
apply (auto dest: foldSet-permute-diff)
done

lemma (in ACf) fold1Set-determ:
  (A, x) ∈ fold1Set f ==> (A, y) ∈ fold1Set f ==> y = x
proof (clarify elim!: fold1Set.cases)
```

fix *A x B y a b*
assume *Ax*: $(A, x) \in$ *foldSet f id a*
assume *By*: $(B, y) \in$ *foldSet f id b*
assume *anotA*: $a \notin A$
assume *bnotB*: $b \notin B$
assume *eq*: *insert a A* = *insert b B*
show *y*=*x*
proof *cases*
assume *same*: *a*=*b*
hence *A*=*B* **using** *anotA bnotB eq* **by** (*blast elim*!: *equalityE*)
thus *?thesis* **using** *Ax By same* **by** (*blast intro*: *foldSet-determ*)
next
assume *diff*: *a*≠*b*
let *?D* = *B* − {*a*}
have *B*: *B* = *insert a ?D* **and** *A*: *A* = *insert b ?D*
and *aB*: $a \in B$ **and** *bA*: $b \in A$
using *eq anotA bnotB diff* **by** (*blast elim*!:*equalityE*)+
with *aB bnotB By*
have (*insert b ?D*, *y*) ∈ *foldSet f id a*
by (*auto intro*: *foldSet-permute simp add*: *insert-absorb*)
moreover
have (*insert b ?D*, *x*) ∈ *foldSet f id a*
by (*simp add*: *A* [*symmetric*] *Ax*)
ultimately show *?thesis* **by** (*blast intro*: *foldSet-determ*)
qed
qed

lemma (**in** *ACf*) *fold1Set-equality*: (*A*, *y*) : *fold1Set f* ==> *fold1 f A* = *y*
by (*unfold fold1-def*) (*blast intro*: *fold1Set-determ*)

declare
empty-foldSetE [*rule del*] *foldSet.intros* [*rule del*]
empty-fold1SetE [*rule del*] *insert-fold1SetE* [*rule del*]
— No more proves involve these relations.

21.6.2 Semi-Lattices

locale *ACIfSL* = *ACIf* +
fixes *below* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** ⊑ *50*)
assumes *below-def*: (*x* ⊑ *y*) = (*x*·*y* = *x*)

locale *ACIfSLlin* = *ACIfSL* +
assumes *lin*: *x*·*y* ∈ {*x*,*y*}

lemma (**in** *ACIfSL*) *below-refl*[*simp*]: *x* ⊑ *x*
by(*simp add*: *below-def idem*)

lemma (**in** *ACIfSL*) *below-f-conv*[*simp*]: *x* ⊑ *y* · *z* = (*x* ⊑ *y* ∧ *x* ⊑ *z*)
proof

```
  assume x ⊑ y · z
  hence xyzx: x · (y · z) = x  by(simp add: below-def)
  have x · y = x
  proof −
    have x · y = (x · (y · z)) · y by(rule subst[OF xyzx], rule refl)
    also have ... = x · (y · z) by(simp add:ACI)
    also have ... = x by(rule xyzx)
    finally show ?thesis .
  qed
  moreover have x · z = x
  proof −
    have x · z = (x · (y · z)) · z by(rule subst[OF xyzx], rule refl)
    also have ... = x · (y · z) by(simp add:ACI)
    also have ... = x by(rule xyzx)
    finally show ?thesis .
  qed
  ultimately show x ⊑ y ∧ x ⊑ z by(simp add: below-def)
next
  assume a: x ⊑ y ∧ x ⊑ z
  hence y: x · y = x and z: x · z = x by(simp-all add: below-def)
  have x · (y · z) = (x · y) · z by(simp add:assoc)
  also have x · y = x using a by(simp-all add: below-def)
  also have x · z = x using a by(simp-all add: below-def)
  finally show x ⊑ y · z by(simp-all add: below-def)
qed

lemma (in ACIfSLlin) above-f-conv:
 x · y ⊑ z = (x ⊑ z ∨ y ⊑ z)
proof
  assume a: x · y ⊑ z
  have x · y = x ∨ x · y = y using lin[of x y] by simp
  thus x ⊑ z ∨ y ⊑ z
  proof
    assume x · y = x hence x ⊑ z by(rule subst)(rule a) thus ?thesis ..
  next
    assume x · y = y hence y ⊑ z by(rule subst)(rule a) thus ?thesis ..
  qed
next
  assume x ⊑ z ∨ y ⊑ z
  thus x · y ⊑ z
  proof
    assume a: x ⊑ z
    have (x · y) · z = (x · z) · y by(simp add:ACI)
    also have x · z = x using a by(simp add:below-def)
    finally show x · y ⊑ z by(simp add:below-def)
  next
    assume a: y ⊑ z
    have (x · y) · z = x · (y · z) by(simp add:ACI)
    also have y · z = y using a by(simp add:below-def)
```

finally show $x \cdot y \sqsubseteq z$ **by**(*simp add*:*below-def*)
qed
qed

21.6.3 Lemmas about *fold1*

lemma (**in** *ACf*) *fold1-Un*:
assumes *A*: *finite A* $A \neq \{\}$
shows *finite B* $\Longrightarrow$ $B \neq \{\} \Longrightarrow$ *A Int B* $= \{\} \Longrightarrow$
fold1 f (*A Un B*) = *f* (*fold1 f A*) (*fold1 f B*)
using *A*
proof(*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by**(*simp add*:*fold1-insert*)
next
case *insert* **thus** *?case* **by** (*simp add*:*fold1-insert assoc*)
qed

lemma (**in** *ACIf*) *fold1-Un2*:
assumes *A*: *finite A* $A \neq \{\}$
shows *finite B* $\Longrightarrow$ $B \neq \{\} \Longrightarrow$
fold1 f (*A Un B*) = *f* (*fold1 f A*) (*fold1 f B*)
using *A*
proof(*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by**(*simp add*:*fold1-insert-idem*)
next
case *insert* **thus** *?case* **by** (*simp add*:*fold1-insert-idem assoc*)
qed

lemma (**in** *ACf*) *fold1-in*:
assumes *A*: *finite* (*A*) $A \neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x \cdot y \in \{x,y\}$
shows *fold1 f A* $\in A$
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

lemma (**in** *ACIfSL*) *below-fold1-iff*:
assumes *A*: *finite A* $A \neq \{\}$
shows $x \sqsubseteq$ *fold1 f A* $= (\forall\ a{\in}A.\ x \sqsubseteq a)$
using *A*
by(*induct rule*:*finite-ne-induct*) *simp-all*

lemma (**in** *ACIfSL*) *fold1-belowI*:
assumes *A*: *finite A* $A \neq \{\}$
shows $a \in A \Longrightarrow$ *fold1 f A* $\sqsubseteq a$
using *A*
proof (*induct rule*:*finite-ne-induct*)

```
  case singleton thus ?case by simp
next
  case (insert x F)
  from insert(5) have a = x ∨ a ∈ F by simp
  thus ?case
  proof
    assume a = x thus ?thesis using insert by(simp add:below-def ACI)
  next
    assume a ∈ F
    hence bel: fold1 f F ⊑ a by(rule insert)
    have fold1 f (insert x F) · a = x · (fold1 f F · a)
      using insert by(simp add:below-def ACI)
    also have fold1 f F · a = fold1 f F
      using bel  by(simp add:below-def ACI)
    also have x · ... = fold1 f (insert x F)
      using insert by(simp add:below-def ACI)
    finally show ?thesis  by(simp add:below-def)
  qed
qed

lemma (in ACIfSLlin) fold1-below-iff:
assumes A: finite A A ≠ {}
shows fold1 f A ⊑ x = (∃ a∈A. a ⊑ x)
using A
by(induct rule:finite-ne-induct)(simp-all add:above-f-conv)
```

21.6.4 Lattices

```
locale Lattice = lattice +
  fixes Inf :: 'a set ⇒ 'a (⨅ - [900] 900)
  and Sup :: 'a set ⇒ 'a (⨆ - [900] 900)
  defines Inf == fold1 inf  and Sup == fold1 sup

locale Distrib-Lattice = distrib-lattice + Lattice
```

Lattices are semilattices

```
lemma (in Lattice) ACf-inf: ACf inf
by(blast intro: ACf.intro inf-commute inf-assoc)

lemma (in Lattice) ACf-sup: ACf sup
by(blast intro: ACf.intro sup-commute sup-assoc)

lemma (in Lattice) ACIf-inf: ACIf inf
apply(rule ACIf.intro)
apply(rule ACf-inf)
apply(rule ACIf-axioms.intro)
apply(rule inf-idem)
done
```

lemma (**in** *Lattice*) *ACIf-sup*: *ACIf sup*
apply(*rule ACIf.intro*)
apply(*rule ACf-sup*)
apply(*rule ACIf-axioms.intro*)
apply(*rule sup-idem*)
done

lemma (**in** *Lattice*) *ACIfSL-inf*: *ACIfSL inf* (*op* $\sqsubseteq$)
apply(*rule ACIfSL.intro*)
apply(*rule ACf-inf*)
apply(*rule ACIf.axioms*[*OF ACIf-inf*])
apply(*rule ACIfSL-axioms.intro*)
apply(*rule iffI*)
apply(*blast intro*: *antisym inf-le1 inf-le2 inf-least refl*)
apply(*erule subst*)
apply(*rule inf-le2*)
done

lemma (**in** *Lattice*) *ACIfSL-sup*: *ACIfSL sup* (%*x y*. *y* $\sqsubseteq$ *x*)
apply(*rule ACIfSL.intro*)
apply(*rule ACf-sup*)
apply(*rule ACIf.axioms*[*OF ACIf-sup*])
apply(*rule ACIfSL-axioms.intro*)
apply(*rule iffI*)
apply(*blast intro*: *antisym sup-ge1 sup-ge2 sup-greatest refl*)
apply(*erule subst*)
apply(*rule sup-ge2*)
done

21.6.5 Fold laws in lattices

lemma (**in** *Lattice*) *Inf-le-Sup*[*simp*]: $[\![$ *finite A*; *A* $\neq$ {} $]\!] \Longrightarrow \bigsqcap A \sqsubseteq \bigsqcup A$
apply(*unfold Sup-def Inf-def*)
apply(*subgoal-tac EX a*. *a*:*A*)
prefer *2* **apply** *blast*
apply(*erule exE*)
apply(*rule trans*)
apply(*erule* (*2*) *ACIfSL.fold1-belowI*[*OF ACIfSL-inf*])
apply(*erule* (*2*) *ACIfSL.fold1-belowI*[*OF ACIfSL-sup*])
done

lemma (**in** *Lattice*) *sup-Inf-absorb*[*simp*]:
$[\![$ *finite A*; *A* $\neq$ {}; *a* $\in$ *A* $]\!] \Longrightarrow (a \sqcup \bigsqcap A) = a$
apply(*subst sup-commute*)
apply(*simp add*:*Inf-def sup-absorb ACIfSL.fold1-belowI*[*OF ACIfSL-inf*])
done

lemma (**in** *Lattice*) *inf-Sup-absorb*[*simp*]:
$[\![$ *finite A*; *A* $\neq$ {}; *a* $\in$ *A* $]\!] \Longrightarrow (a \sqcap \bigsqcup A) = a$

by(*simp add*:*Sup-def inf-absorb ACIfSL.fold1-belowI*[*OF ACIfSL-sup*])

lemma (**in** *Distrib-Lattice*) *sup-Inf1-distrib*:
assumes *A*: *finite A* $A \neq \{\}$
shows $(x \sqcup \bigsqcap A) = \bigsqcap \{x \sqcup a | a.\ a \in A\}$
using *A*
proof (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by**(*simp add*:*Inf-def*)
next
case (*insert y A*)
have *fin*: *finite* $\{x \sqcup a \mid a.\ a \in A\}$
by(*fast intro*: *finite-surj*[**where** $f = \%a.\ x \sqcup a$, *OF insert*(*1*)])
have $x \sqcup \bigsqcap (\mathit{insert}\ y\ A) = x \sqcup (y \sqcap \bigsqcap A)$
using *insert* **by**(*simp add*:*ACf.fold1-insert-def*[*OF ACf-inf Inf-def*])
also have $\ldots = (x \sqcup y) \sqcap (x \sqcup \bigsqcap A)$ **by**(*rule sup-inf-distrib1*)
also have $x \sqcup \bigsqcap A = \bigsqcap \{x \sqcup a | a.\ a \in A\}$ **using** *insert* **by** *simp*
also have $(x \sqcup y) \sqcap \ldots = \bigsqcap (\mathit{insert}\ (x \sqcup y)\ \{x \sqcup a \mid a.\ a \in A\})$
using *insert* **by**(*simp add*:*ACIf.fold1-insert-idem-def*[*OF ACIf-inf Inf-def fin*])
also have *insert* $(x \sqcup y)$ $\{x \sqcup a \mid a.\ a \in A\} = \{x \sqcup a \mid a.\ a \in \mathit{insert}\ y\ A\}$
by *blast*
finally show *?case* **.**
qed

lemma (**in** *Distrib-Lattice*) *sup-Inf2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows $(\bigsqcap A \sqcup \bigsqcap B) = \bigsqcap \{a \sqcup b | a\ b.\ a \in A \wedge b \in B\}$
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-def*])
next
case (*insert x A*)
have *finB*: *finite* $\{x \sqcup b \mid b.\ b \in B\}$
by(*fast intro*: *finite-surj*[**where** $f = \%b.\ x \sqcup b$, *OF B*(*1*)])
have *finAB*: *finite* $\{a \sqcup b \mid a\ b.\ a \in A \wedge b \in B\}$
proof −
have $\{a \sqcup b \mid a\ b.\ a \in A \wedge b \in B\} = (\mathit{UN}\ a{:}A.\ \mathit{UN}\ b{:}B.\ \{a \sqcup b\})$
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: $\{a \sqcup b \mid a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B* **by** *blast*
have $\bigsqcap (\mathit{insert}\ x\ A) \sqcup \bigsqcap B = (x \sqcap \bigsqcap A) \sqcup \bigsqcap B$
using *insert* **by**(*simp add*:*ACIf.fold1-insert-idem-def*[*OF ACIf-inf Inf-def*])
also have $\ldots = (x \sqcup \bigsqcap B) \sqcap (\bigsqcap A \sqcup \bigsqcap B)$ **by**(*rule sup-inf-distrib2*)
also have $\ldots = \bigsqcap \{x \sqcup b | b.\ b \in B\} \sqcap \bigsqcap \{a \sqcup b | a\ b.\ a \in A \wedge b \in B\}$
using *insert* **by**(*simp add*:*sup-Inf1-distrib*[*OF B*])
also have $\ldots = \bigsqcap (\{x \sqcup b \mid b.\ b \in B\} \cup \{a \sqcup b \mid a\ b.\ a \in A \wedge b \in B\})$
(**is** - = $\bigsqcap$ *?M*)

using *B insert*
by(*simp add*:*Inf-def ACIf.fold1-Un2*[*OF ACIf-inf finB - finAB ne*])
also have *?M* = {*a* ⊔ *b* |*a b*. *a* ∈ *insert x A* ∧ *b* ∈ *B*}
by *blast*
finally show *?case* **.**
qed

21.7 Min and Max

As an application of *fold1* we define the minimal and maximal element of a (non-empty) set over a linear order.

constdefs
Min :: ('*a*::*linorder*)*set* => '*a*
Min == *fold1 min*

Max :: ('*a*::*linorder*)*set* => '*a*
Max == *fold1 max*

Before we can do anything, we need to show that *min* and *max* are ACI and the ordering is linear:

interpretation *min*: *ACf* [*min*:: '*a*::*linorder* ⇒ '*a* ⇒ '*a*]
apply(*rule ACf.intro*)
apply(*auto simp*:*min-def*)
done

interpretation *min*: *ACIf* [*min*:: '*a*::*linorder* ⇒ '*a* ⇒ '*a*]
apply(*rule ACIf-axioms.intro*)
apply(*auto simp*:*min-def*)
done

interpretation *max*: *ACf* [*max* :: '*a*::*linorder* ⇒ '*a* ⇒ '*a*]
apply(*rule ACf.intro*)
apply(*auto simp*:*max-def*)
done

interpretation *max*: *ACIf* [*max*:: '*a*::*linorder* ⇒ '*a* ⇒ '*a*]
apply(*rule ACIf-axioms.intro*)
apply(*auto simp*:*max-def*)
done

interpretation *min*:
ACIfSL [*min*:: '*a*::*linorder* ⇒ '*a* ⇒ '*a op* ≤]
apply(*rule ACIfSL-axioms.intro*)
apply(*auto simp*:*min-def*)
done

interpretation *min*:
ACIfSLlin [*min* :: '*a*::*linorder* ⇒ '*a* ⇒ '*a op* ≤]

apply(*rule ACIfSLlin-axioms.intro*)
apply(*auto simp*:*min-def*)
done

interpretation *max*:
ACIfSL [*max* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a* %*x y*. *y*≤*x*]
apply(*rule ACIfSL-axioms.intro*)
apply(*auto simp*:*max-def*)
done

interpretation *max*:
ACIfSLlin [*max* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a* %*x y*. *y*≤*x*]
apply(*rule ACIfSLlin-axioms.intro*)
apply(*auto simp*:*max-def*)
done

interpretation *min-max*:
Lattice [*op* ≤ *min* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a max Min Max*]
apply −
apply(*rule Min-def*)
apply(*rule Max-def*)
done

interpretation *min-max*:
Distrib-Lattice [*op* ≤ *min* :: *'a*::*linorder* ⇒ *'a* ⇒ *'a max Min Max*]
.

Now we instantiate the recursion equations and declare them simplification rules:

lemmas *Min-singleton* = *fold1-singleton-def* [*OF Min-def*]
lemmas *Max-singleton* = *fold1-singleton-def* [*OF Max-def*]
lemmas *Min-insert* = *min.fold1-insert-idem-def* [*OF Min-def*]
lemmas *Max-insert* = *max.fold1-insert-idem-def* [*OF Max-def*]

declare *Min-singleton* [*simp*] *Max-singleton* [*simp*]
declare *Min-insert* [*simp*] *Max-insert* [*simp*]

Now we instantiate some *fold1* properties:

lemma *Min-in* [*simp*]:
shows *finite A* ⟹ *A* ≠ {} ⟹ *Min A* ∈ *A*
using *min.fold1-in*
by(*fastsimp simp*: *Min-def min-def*)

lemma *Max-in* [*simp*]:
shows *finite A* ⟹ *A* ≠ {} ⟹ *Max A* ∈ *A*
using *max.fold1-in*
by(*fastsimp simp*: *Max-def max-def*)

lemma *Min-le* [*simp*]: ⟦ *finite A*; *A* ≠ {}; *x* ∈ *A* ⟧ ⟹ *Min A* ≤ *x*
by(*simp add*: *Min-def min.fold1-belowI*)

lemma *Max-ge* [*simp*]: ⟦ *finite A*; *A* ≠ {}; *x* ∈ *A* ⟧ ⟹ *x* ≤ *Max A*
by(*simp add*: *Max-def max.fold1-belowI*)

lemma *Min-ge-iff*[*simp*]:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ (*x* ≤ *Min A*) = (∀ *a*∈*A*. *x* ≤ *a*)
by(*simp add*: *Min-def min.below-fold1-iff*)

lemma *Max-le-iff*[*simp*]:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ (*Max A* ≤ *x*) = (∀ *a*∈*A*. *a* ≤ *x*)
by(*simp add*: *Max-def max.below-fold1-iff*)

lemma *Min-le-iff*:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ (*Min A* ≤ *x*) = (∃ *a*∈*A*. *a* ≤ *x*)
by(*simp add*: *Min-def min.fold1-below-iff*)

lemma *Max-ge-iff*:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ (*x* ≤ *Max A*) = (∃ *a*∈*A*. *x* ≤ *a*)
by(*simp add*: *Max-def max.fold1-below-iff*)

21.8 Properties of axclass *finite*

Many of these are by Brian Huffman.

lemma *finite-set*: *finite* (*A*::'*a*::*finite set*)
by (*rule finite-subset* [*OF subset-UNIV finite*])

instance *unit* :: *finite*
proof
have *finite* {()} **by** *simp*
also have {()} = *UNIV* **by** *auto*
finally show *finite* (*UNIV* :: *unit set*) **.**
qed

instance *bool* :: *finite*
proof
have *finite* {*True*, *False*} **by** *simp*
also have {*True*, *False*} = *UNIV* **by** *auto*
finally show *finite* (*UNIV* :: *bool set*) **.**
qed

instance ∗ :: (*finite*, *finite*) *finite*
proof
show *finite* (*UNIV* :: ('*a* × '*b*) *set*)
proof (*rule finite-Prod-UNIV*)
show *finite* (*UNIV* :: '*a set*) **by** (*rule finite*)

```
    show finite (UNIV :: 'b set) by (rule finite)
  qed
qed

instance + :: (finite, finite) finite
proof
  have a: finite (UNIV :: 'a set) by (rule finite)
  have b: finite (UNIV :: 'b set) by (rule finite)
  from a b have finite ((UNIV :: 'a set) <+> (UNIV :: 'b set))
    by (rule finite-Plus)
  thus finite (UNIV :: ('a + 'b) set) by simp
qed


instance set :: (finite) finite
proof
  have finite (UNIV :: 'a set) by (rule finite)
  hence finite (Pow (UNIV :: 'a set))
    by (rule finite-Pow-iff [THEN iffD2])
  thus finite (UNIV :: 'a set set) by simp
qed

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}
    show finite (range ?graph) by (rule finite-set)
    show inj ?graph by (rule inj-graph)
  qed
qed

end
```

22 Wellfounded-Relations: Well-founded Relations

```
theory Wellfounded-Relations
imports Finite-Set
begin
```

Derived WF relations such as inverse image, lexicographic product and measure. The simple relational product, in which (x', y') precedes (x, y) if $x' < x$ and $y' < y$, is a subset of the lexicographic product, and therefore does not need to be defined separately.

constdefs
less-than :: $(nat*nat)set$
less-than == *trancl pred-nat*

measure :: $('a => nat) => ('a * 'a)set$
measure == *inv-image less-than*

lex-prod :: $[('a*'a)set, ('b*'b)set] => (('a*'b)*('a*'b))set$
(**infixr** $<*lex*>$ *80*)
ra $<*lex*>$ *rb* == $\{((a,b),(a',b')).\ (a,a') : ra \mid a=a' \ \&\ (b,b') : rb\}$

finite-psubset :: $('a\ set * 'a\ set)\ set$
— finite proper subset
finite-psubset == $\{(A,B).\ A < B \ \&\ finite\ B\}$

same-fst :: $('a => bool) => ('a => ('b * 'b)set) => (('a*'b)*('a*'b))set$
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. See *Library/While-Combinator.thy* for an application.

22.1 Measure Functions make Wellfounded Relations

22.1.1 'Less than' on the natural numbers

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-def*)

lemma *full-nat-induct*:
assumes *ih*: (!!*n*. (*ALL m*. *Suc m* <= *n* −−> *P m*) ==> *P n*)
shows *P n*
apply (*rule wf-less-than* [*THEN wf-induct*])
apply (*rule ih*, *auto*)
done

22.1.2 The Inverse Image into a Wellfounded Relation is Wellfounded.

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

22.1.3 Finally, All Measures are Wellfounded.

lemma *wf-measure* [*iff*]: *wf* (*measure f*)
apply (*unfold measure-def*)
apply (*rule wf-less-than* [*THEN wf-inv-image*])
done

lemmas *measure-induct* =
wf-measure [*THEN wf-induct*, *unfolded measure-def inv-image-def*,
simplified, *standard*]

22.2 Other Ways of Constructing Wellfounded Relations

Wellfoundedness of lexicographic combinations

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

Transitivity of WF combinators.

lemma *trans-lex-prod* [*intro*!]:
[| *trans R1*; *trans R2* |] ==> *trans* (*R1* <*lex*> *R2*)
by (*unfold trans-def lex-prod-def*, *blast*)

22.2.1 Wellfoundedness of proper subset on finite sets.

lemma *wf-finite-psubset*: *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-def* [*symmetric*])
apply (*fast elim*!: *psubset-card-mono*)
done

lemma *trans-finite-psubset*: *trans finite-psubset*
by (*simp add*: *finite-psubset-def psubset-def trans-def*, *blast*)

22.2.2 Wellfoundedness of finite acyclic relations

This proof belongs in this theory because it needs Finite.

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*)

22.2.3 Wellfoundedness of *same-fst*

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

22.3 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.

This material does not appear to be used any longer.

lemma *lemma1*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^* |] ==> (*f* (*i*+*k*), *f i*) : *r*^*
apply (*induct-tac k*, *simp-all*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *lemma2*: [| *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** *lemma1*)

apply (*blast elim*: *rtranclE dest*: *rtrancl-into-trancl1*)
done

lemma *wf-weak-decr-stable*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^*; *wf* (*r*^+) |]
    ==> *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
apply (*drule-tac x* = *0* **in** *lemma2* [*THEN spec*], *auto*)
done

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

ML
⟪
val less-than-def = *thm less-than-def*;
val measure-def = *thm measure-def*;
val lex-prod-def = *thm lex-prod-def*;
val finite-psubset-def = *thm finite-psubset-def*;

val wf-less-than = *thm wf-less-than*;
val trans-less-than = *thm trans-less-than*;
val less-than-iff = *thm less-than-iff*;
val full-nat-induct = *thm full-nat-induct*;
val wf-inv-image = *thm wf-inv-image*;
val wf-measure = *thm wf-measure*;
val measure-induct = *thm measure-induct*;
val wf-lex-prod = *thm wf-lex-prod*;
val trans-lex-prod = *thm trans-lex-prod*;
val wf-finite-psubset = *thm wf-finite-psubset*;
val trans-finite-psubset = *thm trans-finite-psubset*;
val finite-acyclic-wf = *thm finite-acyclic-wf*;
val finite-acyclic-wf-converse = *thm finite-acyclic-wf-converse*;
val wf-iff-acyclic-if-finite = *thm wf-iff-acyclic-if-finite*;
val wf-weak-decr-stable = *thm wf-weak-decr-stable*;
val weak-decr-stable = *thm weak-decr-stable*;
val same-fstI = *thm same-fstI*;
val wf-same-fst = *thm wf-same-fst*;
⟫

end

23 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *Relation Finite-Set*
begin

23.1 Equivalence relations

locale *equiv* =
  fixes A **and** r
  assumes *refl*: *refl* 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 ==> $r^{-1}\ O\ r \subseteq r$
  by (*unfold trans-def sym-def converse-def*) *blast*

lemma *refl-comp-subset*: *refl* A r ==> $r \subseteq r^{-1}\ O\ r$
  by (*unfold refl-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-comp-subset*)+
  done

Second half.

lemma *comp-equivI*:
    $r^{-1}\ O\ r = r$ ==> *Domain* $r = A$ ==> *equiv* A r
  apply (*unfold equiv-def refl-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

23.2 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-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-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-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*)

23.3 Quotients

constdefs
quotient :: ['*a set*, ('*a**'*a*) *set*] => '*a set set* (**infixl** '//' *90*)
$A//r == \bigcup x \in A.\ \{r``\{x\}\}$ — 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-def quotient-def*) *blast*

lemma *quotient-disj*:

equiv A r ==> X ∈ A//r ==> Y ∈ A//r ==> X = Y | (X ∩ 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 ∈ A//r*; *Y ∈ A//r*; *x ∈ X*; *y ∈ Y*; *(x,y) ∈ 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 ∈ A//r*; *Y ∈ A//r*; *x ∈ X*; *y ∈ Y*|] ==> *(X = Y) = ((x,y) ∈ 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 *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-on (%a. {a}//r) A*; *a ∈ A* ⟧ ⟹ *(A − {a})//r = A//r − {a}//r*
apply(*simp add*:*quotient-def inj-on-def*)
apply *blast*
done

23.4 Defining unary operations upon equivalence classes

A congruence-preserving function

locale *congruent* =

fixes r **and** f
assumes *congruent*: $(y,z) \in r \Longrightarrow f\ y = f\ z$

syntax
$RESPECTS :: ['a \Rightarrow 'b, ('a * 'a)\ set] \Rightarrow bool$ (**infixr** *respects* 80)

translations
$f\ respects\ r == congruent\ r\ f$

lemma *UN-constant-eq*: $a \in A \Longrightarrow \forall y \in A.\ f\ y = c \Longrightarrow (\bigcup y \in A.\ f(y)) = c$
— lemma required to prove *UN-equiv-class*
by *auto*

lemma *UN-equiv-class*:
$equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow a \in A$
$\Longrightarrow (\bigcup x \in r``\{a\}.\ f\ x) = f\ a$
— 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*:
$equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow X \in A//r \Longrightarrow$
$(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\bigcup x \in X.\ f\ x) \in B$
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*:
$equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow$
$(\bigcup x \in X.\ f\ x) = (\bigcup y \in Y.\ f\ y) \Longrightarrow X \in A//r \Longrightarrow Y \in A//r$
$\Longrightarrow (!!x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \Longrightarrow (x,\ y) \in r)$
$\Longrightarrow X = Y$
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

23.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

locale *congruent2* =
  fixes *r1* **and** *r2* **and** *f*
  assumes *congruent2*:
    $(y1,z1) \in r1 \Longrightarrow (y2,z2) \in r2 \Longrightarrow f\ y1\ y2 = f\ z1\ z2$

Abbreviation for the common case where the relations are identical

syntax
  RESPECTS2 ::$['a \Rightarrow 'b, ('a * 'a)\ set] \Rightarrow bool$ (**infixr** *respects2* 80)

translations
  f *respects2* $r \Rightarrow$ *congruent2* $r\ r\ f$

lemma *congruent2-implies-congruent*:
    equiv A r1 $\Longrightarrow$ *congruent2 r1 r2 f* $\Longrightarrow$ $a \in A \Longrightarrow$ *congruent r2* $(f\ a)$
  by (*unfold congruent-def congruent2-def equiv-def refl-def*) *blast*

lemma *congruent2-implies-congruent-UN*:
  equiv A1 r1 $\Longrightarrow$ *equiv A2 r2* $\Longrightarrow$ *congruent2 r1 r2 f* $\Longrightarrow$ $a \in A2 \Longrightarrow$
    congruent r1 $(\lambda x1.\ \bigcup x2 \in r2``\{a\}.\ f\ x1\ x2)$
  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-def*)
  apply (*blast del*: *equalityI*)
  done

lemma *UN-equiv-class2*:
  equiv A1 r1 $\Longrightarrow$ *equiv A2 r2* $\Longrightarrow$ *congruent2 r1 r2 f* $\Longrightarrow$ $a1 \in A1 \Longrightarrow$ $a2 \in A2$
    $\Longrightarrow (\bigcup x1 \in r1``\{a1\}.\ \bigcup x2 \in r2``\{a2\}.\ f\ x1\ x2) = f\ a1\ a2$
  by (*simp add*: *UN-equiv-class congruent2-implies-congruent*
    congruent2-implies-congruent-UN)

lemma *UN-equiv-class-type2*:
  equiv A1 r1 $\Longrightarrow$ *equiv A2 r2* $\Longrightarrow$ *congruent2 r1 r2 f*
    $\Longrightarrow X1 \in A1//r1 \Longrightarrow X2 \in A2//r2$
    $\Longrightarrow (!!x1\ x2.\ x1 \in A1 \Longrightarrow x2 \in A2 \Longrightarrow f\ x1\ x2 \in B)$
    $\Longrightarrow (\bigcup x1 \in X1.\ \bigcup x2 \in X2.\ f\ x1\ x2) \in B$
  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-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 (assumption | rule congt |
       erule equivA [THEN equiv-type, THEN subsetD, THEN SigmaE2])+
  done
```

23.6 Cardinality results

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*])
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* (λ*x*. {*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

ML
⟪
val UN-UN-split-split-eq = thm UN-UN-split-split-eq;
val UN-constant-eq = thm UN-constant-eq;
val UN-equiv-class = thm UN-equiv-class;
val UN-equiv-class2 = thm UN-equiv-class2;
val UN-equiv-class-inject = thm UN-equiv-class-inject;
val UN-equiv-class-type = thm UN-equiv-class-type;
val UN-equiv-class-type2 = thm UN-equiv-class-type2;
val Union-quotient = thm Union-quotient;
val comp-equivI = thm comp-equivI;
val congruent2I = thm congruent2I;
val congruent2-commuteI = thm congruent2-commuteI;
val congruent2-def = thm congruent2-def;
val congruent2-implies-congruent = thm congruent2-implies-congruent;
val congruent2-implies-congruent-UN = thm congruent2-implies-congruent-UN;
val congruent-def = thm congruent-def;
val eq-equiv-class = thm eq-equiv-class;
val eq-equiv-class-iff = thm eq-equiv-class-iff;
val equiv-class-eq = thm equiv-class-eq;
val equiv-class-eq-iff = thm equiv-class-eq-iff;
val equiv-class-nondisjoint = thm equiv-class-nondisjoint;
val equiv-class-self = thm equiv-class-self;
val equiv-comp-eq = thm equiv-comp-eq;
val equiv-def = thm equiv-def;
val equiv-imp-dvd-card = thm equiv-imp-dvd-card;
val equiv-type = thm equiv-type;
val finite-equiv-class = thm finite-equiv-class;

val finite-quotient = thm finite-quotient;
val quotientE = thm quotientE;
val quotientI = thm quotientI;
val quotient-def = thm quotient-def;
val quotient-disj = thm quotient-disj;
val refl-comp-subset = thm refl-comp-subset;
val subset-equiv-class = thm subset-equiv-class;
val sym-trans-comp-subset = thm sym-trans-comp-subset;
⟩⟩

end

24 IntDef: The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *IntDef*
imports *Equiv-Relations NatArith*
begin

constdefs
intrel :: $((nat * nat) * (nat * nat))$ *set*
— the equivalence relation underlying the integers
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*)

instance *int* :: {*ord*, *zero*, *one*, *plus*, *times*, *minus*} **..**

constdefs
int :: *nat* => *int*
int m == *Abs-Integ*(*intrel* `` $\{(m,0)\}$)

defs (**overloaded**)

Zero-int-def: *0* == *int 0*
One-int-def: *1* == *int 1*

minus-int-def:
$-\ z$ == *Abs-Integ* ($\bigcup(x,y) \in$ *Rep-Integ z*. *intrel*``$\{(y,x)\}$)

add-int-def:
$z + w$ ==
Abs-Integ ($\bigcup(x,y) \in$ *Rep-Integ z*. $\bigcup(u,v) \in$ *Rep-Integ w*.
intrel``$\{(x+u,\ y+v)\}$)

diff-int-def: $z - (w::int) == z + (-w)$

mult-int-def:
$z * w ==$
$\quad Abs\text{-}Integ\ (\bigcup(x,y) \in Rep\text{-}Integ\ z.\ \bigcup(u,v) \in Rep\text{-}Integ\ w.$
$\qquad intrel``\{(x*u + y*v,\ x*v + y*u)\})$

le-int-def:
$z \leq (w::int) ==$
$\exists x\ y\ u\ v.\ x+v \leq u+y\ \&\ (x,y) \in Rep\text{-}Integ\ z\ \&\ (u,v) \in Rep\text{-}Integ\ w$

less-int-def: $(z < (w::int)) == (z \leq w\ \&\ z \neq w)$

24.1 Construction of the Integers

24.1.1 Preliminary Lemmas about the Equivalence Relation

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-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* = *eq-equiv-class-iff* [*OF equiv-intrel UNIV-I UNIV-I*]

declare *equiv-intrel-iff* [*simp*]

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*] *Abs-Integ-inverse* [*simp*]

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*]:
$\quad (!!x\ y.\ z = Abs\text{-}Integ(intrel``\{(x,y)\}) ==> P) ==> P$
apply (*rule Abs-Integ-cases* [*of z*])
apply (*auto simp add*: *Integ-def quotient-def*)
done

24.1.2 *int*: Embedding the Naturals into the Integers

lemma *inj-int*: *inj int*

by (*simp add*: *inj-on-def int-def*)

lemma *int-int-eq* [*iff*]: (*int m* = *int n*) = (*m* = *n*)
by (*fast elim*!: *inj-int* [*THEN injD*])

24.1.3 Integer Unary Negation

lemma *minus*: − *Abs-Integ*(*intrel*``{(*x*,*y*)}) = *Abs-Integ*(*intrel* `` {(*y*,*x*)})
proof −
  have (λ(*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 *zminus-zminus*: − (− *z*) = (*z*::*int*)
by (*cases z*, *simp add*: *minus*)

lemma *zminus-0*: − *0* = (*0*::*int*)
by (*simp add*: *int-def Zero-int-def minus*)

24.2 Integer Addition

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

lemma *zminus-zadd-distrib*: − (*z* + *w*) = (− *z*) + (− *w*::*int*)
by (*cases z*, *cases w*, *simp add*: *minus add*)

lemma *zadd-commute*: (*z*::*int*) + *w* = *w* + *z*
by (*cases z*, *cases w*, *simp add*: *add-ac add*)

lemma *zadd-assoc*: ((*z1*::*int*) + *z2*) + *z3* = *z1* + (*z2* + *z3*)
by (*cases z1*, *cases z2*, *cases z3*, *simp add*: *add add-assoc*)

lemma *zadd-left-commute*: *x* + (*y* + *z*) = *y* + ((*x* + *z*) ::*int*)
  apply (*rule mk-left-commute* [*of op* +])
  apply (*rule zadd-assoc*)
  apply (*rule zadd-commute*)
  done

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

lemmas *zmult-ac = OrderedGroup.mult-ac*

lemma *zadd-int*: $(int\ m) + (int\ n) = int\ (m + n)$
by (*simp add*: *int-def add*)

lemma *zadd-int-left*: $(int\ m) + (int\ n + z) = int\ (m + n) + z$
by (*simp add*: *zadd-int zadd-assoc* [*symmetric*])

lemma *int-Suc*: $int\ (Suc\ m) = 1 + (int\ m)$
by (*simp add*: *One-int-def zadd-int*)

lemma *zadd-0*: $(0::int) + z = z$
apply (*simp add*: *Zero-int-def int-def*)
apply (*cases z*, *simp add*: *add*)
done

lemma *zadd-0-right*: $z + (0::int) = z$
by (*rule trans* [*OF zadd-commute zadd-0*])

lemma *zadd-zminus-inverse2*: $(-\ z) + z = (0::int)$
by (*cases z*, *simp add*: *int-def Zero-int-def minus add*)

24.3 Integer Multiplication

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*:
$Abs\text{-}Integ((intrel``\{(x,y)\})) * Abs\text{-}Integ((intrel``\{(u,v)\})) =$
$Abs\text{-}Integ(intrel\ ``\ \{(x*u + y*v,\ x*v + y*u)\})$
by (*simp add*: *mult-int-def UN-UN-split-split-eq mult-congruent2*
UN-equiv-class2 [*OF equiv-intrel equiv-intrel*])

lemma *zmult-zminus*: $(-\ z) * w = -\ (z * (w::int))$

by (*cases z*, *cases w*, *simp add*: *minus mult add-ac*)

lemma *zmult-commute*: (*z*::*int*) ∗ *w* = *w* ∗ *z*
by (*cases z*, *cases w*, *simp add*: *mult add-ac mult-ac*)

lemma *zmult-assoc*: ((*z1*::*int*) ∗ *z2*) ∗ *z3* = *z1* ∗ (*z2* ∗ *z3*)
by (*cases z1*, *cases z2*, *cases z3*, *simp add*: *mult add-mult-distrib2 mult-ac*)

lemma *zadd-zmult-distrib*: ((*z1*::*int*) + *z2*) ∗ *w* = (*z1* ∗ *w*) + (*z2* ∗ *w*)
by (*cases z1*, *cases z2*, *cases w*, *simp add*: *add mult add-mult-distrib2 mult-ac*)

lemma *zadd-zmult-distrib2*: (*w*::*int*) ∗ (*z1* + *z2*) = (*w* ∗ *z1*) + (*w* ∗ *z2*)
by (*simp add*: *zmult-commute* [*of w*] *zadd-zmult-distrib*)

lemma *zdiff-zmult-distrib*: ((*z1*::*int*) − *z2*) ∗ *w* = (*z1* ∗ *w*) − (*z2* ∗ *w*)
by (*simp add*: *diff-int-def zadd-zmult-distrib zmult-zminus*)

lemma *zdiff-zmult-distrib2*: (*w*::*int*) ∗ (*z1* − *z2*) = (*w* ∗ *z1*) − (*w* ∗ *z2*)
by (*simp add*: *zmult-commute* [*of w*] *zdiff-zmult-distrib*)

lemmas *int-distrib* =
zadd-zmult-distrib zadd-zmult-distrib2
zdiff-zmult-distrib zdiff-zmult-distrib2

lemma *int-mult*: *int* (*m* ∗ *n*) = (*int m*) ∗ (*int n*)
by (*simp add*: *int-def mult*)

Compatibility binding

lemmas *zmult-int* = *int-mult* [*symmetric*]

lemma *zmult-1*: (*1*::*int*) ∗ *z* = *z*
by (*cases z*, *simp add*: *One-int-def int-def mult*)

lemma *zmult-1-right*: *z* ∗ (*1*::*int*) = *z*
by (*rule trans* [*OF zmult-commute zmult-1*])

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** (*simp add*: *zadd-assoc*)
show *i* + *j* = *j* + *i* **by** (*simp add*: *zadd-commute*)
show *0* + *i* = *i* **by** (*rule zadd-0*)
show − *i* + *i* = *0* **by** (*rule zadd-zminus-inverse2*)
show *i* − *j* = *i* + (−*j*) **by** (*simp add*: *diff-int-def*)
show (*i* ∗ *j*) ∗ *k* = *i* ∗ (*j* ∗ *k*) **by** (*rule zmult-assoc*)
show *i* ∗ *j* = *j* ∗ *i* **by** (*rule zmult-commute*)
show *1* ∗ *i* = *i* **by** (*rule zmult-1*)
show (*i* + *j*) ∗ *k* = *i* ∗ *k* + *j* ∗ *k* **by** (*simp add*: *int-distrib*)

show *0 ≠ (1::int)*
by (*simp only*: *Zero-int-def One-int-def One-nat-def int-int-eq*)
qed

24.4 The ≤ Ordering

lemma *le*:
(*Abs-Integ*(*intrel*``{(*x*,*y*)}) ≤ *Abs-Integ*(*intrel*``{(*u*,*v*)})) = (*x*+*v* ≤ *u*+*y*)
by (*force simp add*: *le-int-def*)

lemma *zle-refl*: *w* ≤ (*w*::*int*)
by (*cases w*, *simp add*: *le*)

lemma *zle-trans*: [| *i* ≤ *j*; *j* ≤ *k* |] ==> *i* ≤ (*k*::*int*)
by (*cases i*, *cases j*, *cases k*, *simp add*: *le*)

lemma *zle-anti-sym*: [| *z* ≤ *w*; *w* ≤ *z* |] ==> *z* = (*w*::*int*)
by (*cases w*, *cases z*, *simp add*: *le*)

lemma *zless-le*: ((*w*::*int*) < *z*) = (*w* ≤ *z* & *w* ≠ *z*)
by (*simp add*: *less-int-def*)

instance *int* :: *order*
by *intro-classes*
(*assumption* |
rule zle-refl zle-trans zle-anti-sym zless-le)+

lemma *zle-linear*: (*z*::*int*) ≤ *w* | *w* ≤ *z*
by (*cases z*, *cases w*) (*simp add*: *le linorder-linear*)

instance *int* :: *linorder*
by *intro-classes* (*rule zle-linear*)

lemmas *zless-linear* = *linorder-less-linear* [**where** *'a* = *int*]
lemmas *linorder-neqE-int* = *linorder-neqE*[**where** *'a* = *int*]

lemma *int-eq-0-conv* [*simp*]: (*int n* = *0*) = (*n* = *0*)
by (*simp add*: *Zero-int-def*)

lemma *zless-int* [*simp*]: (*int m* < *int n*) = (*m*<*n*)
by (*simp add*: *le add int-def linorder-not-le* [*symmetric*])

lemma *int-less-0-conv* [*simp*]: ~ (*int k* < *0*)
by (*simp add*: *Zero-int-def*)

lemma *zero-less-int-conv* [*simp*]: $(0 < int\ n) = (0 < n)$
by (*simp add*: *Zero-int-def*)

lemma *int-0-less-1*: $0 < (1::int)$
by (*simp only*: *Zero-int-def One-int-def One-nat-def zless-int*)

lemma *int-0-neq-1* [*simp*]: $0 \neq (1::int)$
by (*simp only*: *Zero-int-def One-int-def One-nat-def int-int-eq*)

lemma *zle-int* [*simp*]: $(int\ m \leq int\ n) = (m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *zero-zle-int* [*simp*]: $(0 \leq int\ n)$
by (*simp add*: *Zero-int-def*)

lemma *int-le-0-conv* [*simp*]: $(int\ n \leq 0) = (n = 0)$
by (*simp add*: *Zero-int-def*)

lemma *int-0* [*simp*]: $int\ 0 = (0::int)$
by (*simp add*: *Zero-int-def*)

lemma *int-1* [*simp*]: $int\ 1 = 1$
by (*simp add*: *One-int-def*)

lemma *int-Suc0-eq-1*: $int\ (Suc\ 0) = 1$
by (*simp add*: *One-int-def One-nat-def*)

24.5 Monotonicity results

lemma *zadd-left-mono*: $i \leq j \Longrightarrow k + i \leq k + (j::int)$
by (*cases i*, *cases j*, *cases k*, *simp add*: *le add*)

lemma *zadd-strict-right-mono*: $i < j \Longrightarrow i + k < j + (k::int)$
apply (*cases i*, *cases j*, *cases k*)
apply (*simp add*: *linorder-not-le* [**where** $'a = int$, *symmetric*]
linorder-not-le [**where** $'a = nat$] *le add*)
done

lemma *zadd-zless-mono*: $[\!|\ w'<w;\ z'\leq z\ |\!] \Longrightarrow w' + z' < w + (z::int)$
by (*rule order-less-le-trans* [*OF zadd-strict-right-mono zadd-left-mono*])

24.6 Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on k¿0

lemma *zmult-zless-mono2-lemma*:
$i<j \Longrightarrow 0<k \Longrightarrow int\ k * i < int\ k * j$
apply (*induct k*, *simp*)
apply (*simp add*: *int-Suc*)
apply (*case-tac k=0*)

apply (*simp-all add*: *zadd-zmult-distrib int-Suc0-eq-1 order-le-less*)
apply (*simp add*: *zadd-zless-mono int-Suc0-eq-1 order-le-less*)
done

lemma *zero-le-imp-eq-int*: $0 \leq k \Longrightarrow \exists n.\ k = int\ n$
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 *zmult-zless-mono2*: $[\!|\ i<j;\ \ (0::int) < k\ |\!] \Longrightarrow k*i < k*j$
apply (*frule order-less-imp-le* [*THEN zero-le-imp-eq-int*])
apply (*auto simp add*: *zmult-zless-mono2-lemma*)
done

defs (**overloaded**)
zabs-def: $abs(i::int) == \textit{if } i < 0 \textit{ then } -i \textit{ else } i$

The integers form an ordered *comm-ring-1*

instance *int* :: *ordered-idom*
proof
fix *i j k* :: *int*
show $i \leq j \Longrightarrow k + i \leq k + j$ **by** (*rule zadd-left-mono*)
show $i < j \Longrightarrow 0 < k \Longrightarrow k * i < k * j$ **by** (*rule zmult-zless-mono2*)
show $|i| = (\textit{if } i < 0 \textit{ then } -i \textit{ else } i)$ **by** (*simp only*: *zabs-def*)
qed

lemma *zless-imp-add1-zle*: $w<z \Longrightarrow w + (1::int) \leq z$
apply (*cases w*, *cases z*)
apply (*simp add*: *linorder-not-le* [*symmetric*] *le int-def add One-int-def*)
done

24.7 Magnitide of an Integer, as a Natural Number: *nat*

constdefs
nat :: *int* => *nat*
nat z == *contents* $(\bigcup(x,y) \in \textit{Rep-Integ } z.\ \{\ x-y\ \})$

lemma *nat*: *nat* (*Abs-Integ* (*intrel*``{(*x*,*y*)})) = *x*−*y*
proof −
have $(\lambda(x,y).\ \{x-y\})$ *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*(*int n*) = *n*

by (*simp add*: *nat int-def*)

lemma *nat-zero* [*simp*]: *nat 0 = 0*
by (*simp only*: *Zero-int-def nat-int*)

lemma *int-nat-eq* [*simp*]: *int* (*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 ≤ z ==> int* (*nat z*) = *z*
by *simp*

lemma *nat-le-0* [*simp*]: *z ≤ 0 ==> nat z = 0*
by (*cases z*, *simp add*: *nat le int-def Zero-int-def*)

lemma *nat-le-eq-zle*: *0 < w | 0 ≤ z ==>* (*nat w ≤ nat z*) = (*w≤z*)
apply (*cases w*, *cases z*)
apply (*simp add*: *nat le linorder-not-le* [*symmetric*] *int-def Zero-int-def*, *arith*)
done

An alternative condition is (*0*::*'a*) ≤ *w*

corollary *nat-mono-iff*: *0 < z ==>* (*nat w < nat z*) = (*w < z*)
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

corollary *nat-less-eq-zless*: *0 ≤ w ==>* (*nat w < nat z*) = (*w<z*)
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

lemma *zless-nat-conj*: (*nat w < nat z*) = (*0 < z & w < z*)
apply (*cases w*, *cases z*)
apply (*simp add*: *nat le int-def Zero-int-def linorder-not-le* [*symmetric*], *arith*)
done

lemma *nonneg-eq-int*: [| *0 ≤ z*; !!*m*. *z = int m ==> P* |] *==> P*
by (*blast dest*: *nat-0-le sym*)

lemma *nat-eq-iff*: (*nat w = m*) = (*if 0 ≤ w then w = int 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 ≤ w then w = int m else m=0*)
by (*simp only*: *eq-commute* [*of m*] *nat-eq-iff*)

lemma *nat-less-iff*: *0 ≤ w ==>* (*nat w < m*) = (*w < int m*)
apply (*cases w*)
apply (*simp add*: *nat le int-def Zero-int-def linorder-not-le* [*symmetric*], *arith*)
done

lemma *int-eq-iff*: (*int m = z*) = (*m = nat z & 0 ≤ 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 int-def 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 int-def Zero-int-def*)

lemma *nat-zminus-int* [*simp*]: *nat* (− (*int n*)) = *0*
by (*simp add*: *int-def minus nat Zero-int-def*)

lemma *zless-nat-eq-int-zless*: (*m* < *nat z*) = (*int m* < *z*)
by (*cases z*, *simp add*: *nat le int-def linorder-not-le* [*symmetric*], *arith*)

24.8 Lemmas about the Function *int* and Orderings

lemma *negative-zless-0*: − (*int* (*Suc n*)) < *0*
by (*simp add*: *order-less-le*)

lemma *negative-zless* [*iff*]: − (*int* (*Suc n*)) < *int m*
by (*rule negative-zless-0* [*THEN order-less-le-trans*], *simp*)

lemma *negative-zle-0*: − *int n* ≤ *0*
by (*simp add*: *minus-le-iff*)

lemma *negative-zle* [*iff*]: − *int n* ≤ *int m*
by (*rule order-trans* [*OF negative-zle-0 zero-zle-int*])

lemma *not-zle-0-negative* [*simp*]: ~ (*0* ≤ − (*int* (*Suc n*)))
by (*subst le-minus-iff*, *simp*)

lemma *int-zle-neg*: (*int n* ≤ − *int m*) = (*n* = *0* & *m* = *0*)
by (*simp add*: *int-def le minus Zero-int-def*)

lemma *not-int-zless-negative* [*simp*]: ~ (*int n* < − *int m*)
by (*simp add*: *linorder-not-less*)

lemma *negative-eq-positive* [*simp*]: (− *int n* = *int m*) = (*n* = *0* & *m* = *0*)
by (*force simp add*: *order-eq-iff* [*of* − *int n*] *int-zle-neg*)

lemma *zle-iff-zadd*: (*w* ≤ *z*) = (∃ *n*. *z* = *w* + *int n*)
proof (*cases w*, *cases z*, *simp add*: *le add int-def*)
fix *a b c d*
assume *w* = *Abs-Integ* (*intrel* `` {(*a*,*b*)}) *z* = *Abs-Integ* (*intrel* `` {(*c*,*d*)})
show (*a*+*d* ≤ *c*+*b*) = (∃ *n*. *c*+*b* = *a*+*n*+*d*)

```
  proof
    assume a + d ≤ c + b
    thus ∃ n. c + b = a + n + d
      by (auto intro!: exI [where x=c+b - (a+d)])
  next
    assume ∃ n. c + b = a + n + d
    then obtain n where c + b = a + n + d ..
    thus a + d ≤ c + b by arith
  qed
qed
```

lemma *abs-int-eq* [*simp*]: *abs* (*int m*) = *int m*
by (*simp add*: *abs-if*)

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*]:
    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*)

24.9 The Constants *neg* and *iszero*

constdefs

```
  neg   :: 'a::ordered-idom => bool
  neg(Z) == Z < 0


  iszero :: 'a::comm-semiring-1-cancel => bool
  iszero z == z = (0)
```

lemma *not-neg-int* [*simp*]: ~ *neg*(*int n*)
by (*simp add*: *neg-def*)

lemma *neg-zminus-int* [*simp*]: *neg*(− (*int* (*Suc n*)))
by (*simp add*: *neg-def neg-less-0-iff-less*)

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*)

24.10 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 *iszero-0*: *iszero 0*
by (*simp add*: *iszero-def*)

lemma *not-iszero-1*: ~ *iszero 1*
by (*simp add*: *iszero-def eq-commute*)

lemma *neg-nat*: *neg z* ==> *nat z* = *0*
by (*simp add*: *neg-def order-less-imp-le*)

lemma *not-neg-nat*: ~ *neg z* ==> *int* (*nat z*) = *z*
by (*simp add*: *linorder-not-less neg-def*)

24.11 The Set of Natural Numbers

constdefs
Nats :: *'a*::*comm-semiring-1-cancel set*
Nats == *range of-nat*

syntax (*xsymbols*) *Nats* :: *'a set* (ℕ)

lemma *of-nat-in-Nats* [*simp*]: *of-nat n* ∈ *Nats*
by (*simp add*: *Nats-def*)

lemma *Nats-0* [*simp*]: *0* ∈ *Nats*
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-0* [*symmetric*])
done

lemma *Nats-1* [*simp*]: *1* ∈ *Nats*
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-1* [*symmetric*])
done

lemma *Nats-add* [*simp*]: [|*a* ∈ *Nats*; *b* ∈ *Nats*|] ==> *a*+*b* ∈ *Nats*
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-add* [*symmetric*])
done

lemma *Nats-mult* [*simp*]: [|*a* ∈ *Nats*; *b* ∈ *Nats*|] ==> *a*∗*b* ∈ *Nats*
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-mult* [*symmetric*])

done

Agreement with the specific embedding for the integers

lemma *int-eq-of-nat*: *int* = (*of-nat* :: *nat* => *int*)
proof
  fix *n*
  show *int n* = *of-nat n* **by** (*induct n*, *simp-all add*: *int-Suc add-ac*)
qed

lemma *of-nat-eq-id* [*simp*]: *of-nat* = (*id* :: *nat* => *nat*)
proof
  fix *n*
  show *of-nat n* = *id n* **by** (*induct n*, *simp-all*)
qed

24.12 Embedding of the Integers into any *comm-ring-1*: *of-int*

constdefs
  of-int :: *int* => 'a::*comm-ring-1*
  of-int z == *contents* ($\bigcup$(*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 compare-rls 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 int-def*)

lemma *of-int-1* [*simp*]: *of-int 1* = *1*
by (*simp add*: *of-int One-int-def int-def*)

lemma *of-int-add* [*simp*]: *of-int* (*w*+*z*) = *of-int w* + *of-int z*
by (*cases w*, *cases z*, *simp add*: *compare-rls of-int add*)

lemma *of-int-minus* [*simp*]: *of-int* (−*z*) = − (*of-int z*)
by (*cases z*, *simp add*: *compare-rls of-int minus*)

lemma *of-int-diff* [*simp*]: *of-int* (*w*−*z*) = *of-int w* − *of-int z*
by (*simp add*: *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*: *compare-rls of-int left-diff-distrib right-diff-distrib*

mult add-ac)
done

lemma *of-int-le-iff* [*simp*]:
(*of-int w* ≤ (*of-int z*::'*a*::*ordered-idom*)) = (*w* ≤ *z*)
apply (*cases w*)
apply (*cases z*)
apply (*simp add*: *compare-rls of-int le diff-int-def add minus*
of-nat-add [*symmetric*] *del*: *of-nat-add*)
done

Special cases where either operand is zero

lemmas *of-int-0-le-iff* = *of-int-le-iff* [*of 0*, *simplified*]
lemmas *of-int-le-0-iff* = *of-int-le-iff* [*of - 0*, *simplified*]
declare *of-int-0-le-iff* [*simp*]
declare *of-int-le-0-iff* [*simp*]

lemma *of-int-less-iff* [*simp*]:
(*of-int w* < (*of-int z*::'*a*::*ordered-idom*)) = (*w* < *z*)
by (*simp add*: *linorder-not-le* [*symmetric*])

Special cases where either operand is zero

lemmas *of-int-0-less-iff* = *of-int-less-iff* [*of 0*, *simplified*]
lemmas *of-int-less-0-iff* = *of-int-less-iff* [*of - 0*, *simplified*]
declare *of-int-0-less-iff* [*simp*]
declare *of-int-less-0-iff* [*simp*]

The ordering on the *comm-ring-1* is necessary. See *of-nat-eq-iff* above.

lemma *of-int-eq-iff* [*simp*]:
(*of-int w* = (*of-int z*::'*a*::*ordered-idom*)) = (*w* = *z*)
by (*simp add*: *order-eq-iff*)

Special cases where either operand is zero

lemmas *of-int-0-eq-iff* = *of-int-eq-iff* [*of 0*, *simplified*]
lemmas *of-int-eq-0-iff* = *of-int-eq-iff* [*of - 0*, *simplified*]
declare *of-int-0-eq-iff* [*simp*]
declare *of-int-eq-0-iff* [*simp*]

lemma *of-int-eq-id* [*simp*]: *of-int* = (*id* :: *int* => *int*)
proof
fix *z*
show *of-int z* = *id z*
by (*cases z*,
simp add: *of-int add minus int-eq-of-nat* [*symmetric*] *int-def diff-minus*)
qed

24.13 The Set of Integers

constdefs

Ints :: *'a*::*comm-ring-1 set*
Ints == *range of-int*

syntax (*xsymbols*)
Ints :: *'a set* (ℤ)

lemma *Ints-0* [*simp*]: *0* ∈ *Ints*
apply (*simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-0* [*symmetric*])
done

lemma *Ints-1* [*simp*]: *1* ∈ *Ints*
apply (*simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-1* [*symmetric*])
done

lemma *Ints-add* [*simp*]: [|*a* ∈ *Ints*; *b* ∈ *Ints*|] ==> *a*+*b* ∈ *Ints*
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-add* [*symmetric*])
done

lemma *Ints-minus* [*simp*]: *a* ∈ *Ints* ==> −*a* ∈ *Ints*
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-minus* [*symmetric*])
done

lemma *Ints-diff* [*simp*]: [|*a* ∈ *Ints*; *b* ∈ *Ints*|] ==> *a*−*b* ∈ *Ints*
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-diff* [*symmetric*])
done

lemma *Ints-mult* [*simp*]: [|*a* ∈ *Ints*; *b* ∈ *Ints*|] ==> *a*∗*b* ∈ *Ints*
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-mult* [*symmetric*])
done

Collapse nested embeddings

lemma *of-int-of-nat-eq* [*simp*]: *of-int* (*of-nat n*) = *of-nat n*
by (*induct n*, *auto*)

lemma *of-int-int-eq* [*simp*]: *of-int* (*int n*) = *of-nat n*
by (*simp add*: *int-eq-of-nat*)

lemma *Ints-cases* [*case-names of-int*, *cases set*: *Ints*]:
  $q \in \mathbb{Z} \Longrightarrow (!!z.\ q = \mathit{of\text{-}int}\ z \Longrightarrow C) \Longrightarrow C$
proof (*simp add*: *Ints-def*)
  assume $!!z.\ q = \mathit{of\text{-}int}\ z \Longrightarrow C$
  assume $q \in \mathit{range\ of\text{-}int}$ **thus** C **..**
qed

lemma *Ints-induct* [*case-names of-int*, *induct set*: *Ints*]:
  $q \in \mathbb{Z} \Longrightarrow (!!z.\ P\ (\mathit{of\text{-}int}\ z)) \Longrightarrow P\ q$
  by (*rule Ints-cases*) *auto*

declare *int-Suc* [*simp*]

24.14 More Properties of *setsum* and *setprod*

By Jeremy Avigad

lemma *of-nat-setsum*: $\mathit{of\text{-}nat}\ (\mathit{setsum}\ f\ A) = (\sum x{\in}A.\ \mathit{of\text{-}nat}(f\ x))$
  apply (*case-tac finite A*)
  apply (*erule finite-induct*, *auto*)
  done

lemma *of-int-setsum*: $\mathit{of\text{-}int}\ (\mathit{setsum}\ f\ A) = (\sum x{\in}A.\ \mathit{of\text{-}int}(f\ x))$
  apply (*case-tac finite A*)
  apply (*erule finite-induct*, *auto*)
  done

lemma *int-setsum*: $\mathit{int}\ (\mathit{setsum}\ f\ A) = (\sum x{\in}A.\ \mathit{int}(f\ x))$
  by (*simp add*: *int-eq-of-nat of-nat-setsum*)

lemma *of-nat-setprod*: $\mathit{of\text{-}nat}\ (\mathit{setprod}\ f\ A) = (\prod x{\in}A.\ \mathit{of\text{-}nat}(f\ x))$
  apply (*case-tac finite A*)
  apply (*erule finite-induct*, *auto*)
  done

lemma *of-int-setprod*: $\mathit{of\text{-}int}\ (\mathit{setprod}\ f\ A) = (\prod x{\in}A.\ \mathit{of\text{-}int}(f\ x))$
  apply (*case-tac finite A*)
  apply (*erule finite-induct*, *auto*)
  done

lemma *int-setprod*: $\mathit{int}\ (\mathit{setprod}\ f\ A) = (\prod x{\in}A.\ \mathit{int}(f\ x))$
  by (*simp add*: *int-eq-of-nat of-nat-setprod*)

lemma *setprod-nonzero-nat*:
    $\mathit{finite}\ A \Longrightarrow (\forall x \in A.\ f\ x \neq (0{::}\mathit{nat})) \Longrightarrow \mathit{setprod}\ f\ A \neq 0$
  by (*rule setprod-nonzero*, *auto*)

lemma *setprod-zero-eq-nat*:
finite A ==> (setprod f A = (0::nat)) = (∃ x ∈ A. f x = 0)
by (*rule setprod-zero-eq*, *auto*)

lemma *setprod-nonzero-int*:
finite A ==> (∀ x ∈ A. f x ≠ (0::int)) ==> setprod f A ≠ 0
by (*rule setprod-nonzero*, *auto*)

lemma *setprod-zero-eq-int*:
finite A ==> (setprod f A = (0::int)) = (∃ x ∈ A. f x = 0)
by (*rule setprod-zero-eq*, *auto*)

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

lemma *zless-iff-Suc-zadd*:
(w < z) = (∃ n. z = w + int(Suc n))
apply (*cases z*, *cases w*)
apply (*auto simp add*: *le add int-def linorder-not-le* [*symmetric*])
apply (*rename-tac a b c d*)
apply (*rule-tac x=a+d - Suc(c+b)* **in** *exI*)
apply *arith*
done

lemma *negD*: *x<0 ==> ∃ n. x = - (int (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

theorem *int-cases* [*cases type*: *int*, *case-names nonneg neg*]:
[|!! n. z = int n ==> P; !! n. z = - (int (Suc n)) ==> P |] ==> P
apply (*case-tac z < 0*, *blast dest*!: *negD*)
apply (*simp add*: *linorder-not-less*)
apply (*blast dest*: *nat-0-le* [*THEN sym*])
done

theorem *int-induct* [*induct type*: *int*, *case-names nonneg neg*]:
[|!! n. P (int n); !!n. P (- (int (Suc n))) |] ==> P z
by (*cases z*) *auto*

Contributed by Brian Huffman

theorem *int-diff-cases* [*case-names diff*]:
assumes *prem*: *!!m n. z = int m - int n ==> P* **shows** *P*
apply (*rule-tac z=z* **in** *int-cases*)
apply (*rule-tac m=n* **and** *n=0* **in** *prem*, *simp*)
apply (*rule-tac m=0* **and** *n=Suc n* **in** *prem*, *simp*)
done

lemma *of-nat-nat*: *0 ≤ z ==> of-nat (nat z) = of-int z*

apply (*cases z*)
apply (*simp-all add*: *not-zle-0-negative del*: *int-Suc*)
done

24.15 Configuration of the code generator

types-code
int (*int*)
attach (*term-of*) ⟪
val term-of-int = HOLogic.mk-int o IntInf.fromInt;
⟫
attach (*test*) ⟪
fun gen-int i = one-of [~*1*, *1*] ∗ *random-range 0 i*;
⟫

constdefs
int-aux :: *int* ⇒ *nat* ⇒ *int*
int-aux i n == (*i* + *int n*)
nat-aux :: *nat* ⇒ *int* ⇒ *nat*
nat-aux n i == (*n* + *nat i*)

lemma [*code*]:
int-aux i 0 = *i*
int-aux i (*Suc n*) = *int-aux* (*i* + *1*) *n* — tail recursive
int n = *int-aux 0 n*
by (*simp add*: *int-aux-def*)+

lemma [*code*]: *nat-aux n i* = (*if i* <= *0 then n else nat-aux* (*Suc n*) (*i* − *1*))
— 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 0 i*
by (*simp add*: *nat-aux-def*)

consts-code
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* ((- <=/ -))
neg ((- < *0*))

ML ⟪
fun number-of-codegen thy defs gr dep module b (*Const* (*Numeral.number-of*,
Type (*fun*, [-, *T as Type* (*IntDef.int*, [])])) $ *bin*) =
(*SOME* (*fst* (*Codegen.invoke-tycodegen thy defs dep module false* (*gr*, *T*)),
Pretty.str (*IntInf.toString* (*HOLogic.dest-binum bin*))) *handle TERM* -

```
=> NONE)
  | number-of-codegen thy defs gr s thyname b (Const (Numeral.number-of,
      Type (fun, [-, Type (nat, [])])) $ bin) =
        SOME (Codegen.invoke-codegen thy defs s thyname b (gr,
          Const (IntDef.nat, HOLogic.intT --> HOLogic.natT) $
          (Const (Numeral.number-of, HOLogic.binT --> HOLogic.intT) $ bin)))
  | number-of-codegen - - - - - - - = NONE;
⟫

setup ⟪ [Codegen.add-codegen number-of-codegen number-of-codegen] ⟫

quickcheck-params [default-type = int]
```

ML

```
⟪
val zabs-def = thm zabs-def

val int-0 = thm int-0;
val int-1 = thm int-1;
val int-Suc0-eq-1 = thm int-Suc0-eq-1;
val neg-eq-less-0 = thm neg-eq-less-0;
val not-neg-eq-ge-0 = thm not-neg-eq-ge-0;
val not-neg-0 = thm not-neg-0;
val not-neg-1 = thm not-neg-1;
val iszero-0 = thm iszero-0;
val not-iszero-1 = thm not-iszero-1;
val int-0-less-1 = thm int-0-less-1;
val int-0-neq-1 = thm int-0-neq-1;
val negative-zless = thm negative-zless;
val negative-zle = thm negative-zle;
val not-zle-0-negative = thm not-zle-0-negative;
val not-int-zless-negative = thm not-int-zless-negative;
val negative-eq-positive = thm negative-eq-positive;
val zle-iff-zadd = thm zle-iff-zadd;
val abs-int-eq = thm abs-int-eq;
val abs-split = thmabs-split;
val nat-int = thm nat-int;
val nat-zminus-int = thm nat-zminus-int;
val nat-zero = thm nat-zero;
val not-neg-nat = thm not-neg-nat;
val neg-nat = thm neg-nat;
val zless-nat-eq-int-zless = thm zless-nat-eq-int-zless;
val nat-0-le = thm nat-0-le;
val nat-le-0 = thm nat-le-0;
val zless-nat-conj = thm zless-nat-conj;
val int-cases = thm int-cases;
```

val int-def = thm int-def;
val Zero-int-def = thm Zero-int-def;
val One-int-def = thm One-int-def;
val diff-int-def = thm diff-int-def;

val inj-int = thm inj-int;
val zminus-zminus = thm zminus-zminus;
val zminus-0 = thm zminus-0;
val zminus-zadd-distrib = thm zminus-zadd-distrib;
val zadd-commute = thm zadd-commute;
val zadd-assoc = thm zadd-assoc;
val zadd-left-commute = thm zadd-left-commute;
val zadd-ac = thms zadd-ac;
val zmult-ac = thms zmult-ac;
val zadd-int = thm zadd-int;
val zadd-int-left = thm zadd-int-left;
val int-Suc = thm int-Suc;
val zadd-0 = thm zadd-0;
val zadd-0-right = thm zadd-0-right;
val zmult-zminus = thm zmult-zminus;
val zmult-commute = thm zmult-commute;
val zmult-assoc = thm zmult-assoc;
val zadd-zmult-distrib = thm zadd-zmult-distrib;
val zadd-zmult-distrib2 = thm zadd-zmult-distrib2;
val zdiff-zmult-distrib = thm zdiff-zmult-distrib;
val zdiff-zmult-distrib2 = thm zdiff-zmult-distrib2;
val int-distrib = thms int-distrib;
val zmult-int = thm zmult-int;
val zmult-1 = thm zmult-1;
val zmult-1-right = thm zmult-1-right;
val int-int-eq = thm int-int-eq;
val int-eq-0-conv = thm int-eq-0-conv;
val zless-int = thm zless-int;
val int-less-0-conv = thm int-less-0-conv;
val zero-less-int-conv = thm zero-less-int-conv;
val zle-int = thm zle-int;
val zero-zle-int = thm zero-zle-int;
val int-le-0-conv = thm int-le-0-conv;
val zle-refl = thm zle-refl;
val zle-linear = thm zle-linear;
val zle-trans = thm zle-trans;
val zle-anti-sym = thm zle-anti-sym;

val Ints-def = thm Ints-def;
val Nats-def = thm Nats-def;

val of-nat-0 = thm of-nat-0;
val of-nat-Suc = thm of-nat-Suc;
val of-nat-1 = thm of-nat-1;

val of-nat-add = thm of-nat-add;
val of-nat-mult = thm of-nat-mult;
val zero-le-imp-of-nat = thm zero-le-imp-of-nat;
val less-imp-of-nat-less = thm less-imp-of-nat-less;
val of-nat-less-imp-less = thm of-nat-less-imp-less;
val of-nat-less-iff = thm of-nat-less-iff;
val of-nat-le-iff = thm of-nat-le-iff;
val of-nat-eq-iff = thm of-nat-eq-iff;
val Nats-0 = thm Nats-0;
val Nats-1 = thm Nats-1;
val Nats-add = thm Nats-add;
val Nats-mult = thm Nats-mult;
val int-eq-of-nat = thmint-eq-of-nat;
val of-int = thm of-int;
val of-int-0 = thm of-int-0;
val of-int-1 = thm of-int-1;
val of-int-add = thm of-int-add;
val of-int-minus = thm of-int-minus;
val of-int-diff = thm of-int-diff;
val of-int-mult = thm of-int-mult;
val of-int-le-iff = thm of-int-le-iff;
val of-int-less-iff = thm of-int-less-iff;
val of-int-eq-iff = thm of-int-eq-iff;
val Ints-0 = thm Ints-0;
val Ints-1 = thm Ints-1;
val Ints-add = thm Ints-add;
val Ints-minus = thm Ints-minus;
val Ints-diff = thm Ints-diff;
val Ints-mult = thm Ints-mult;
val of-int-of-nat-eq = thmof-int-of-nat-eq;
val Ints-cases = thm Ints-cases;
val Ints-induct = thm Ints-induct;
⟩⟩

end

25 Numeral: Arithmetic on Binary Integers

theory *Numeral*
imports *IntDef Datatype*
uses *../Tools/numeral-syntax.ML*
begin

The file *numeral-syntax.ML* hides the constructors Pls and Min. Only qualified access Numeral.Pls and Numeral.Min is allowed. The datatype constructors bit.B0 and bit.B1 are similarly hidden. We do not hide Bit because we need the BIT infix syntax.

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Integ/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*.

typedef (*Bin*)
bin = UNIV::*int set*
by (*auto*)

This datatype avoids the use of type *bool*, which would make all of the rewrite rules higher-order. If the use of datatype causes problems, this two-element type can easily be formalized using typedef.

datatype *bit = B0 | B1*

constdefs
Pls :: *bin*
Pls == Abs-Bin 0

Min :: *bin*
Min == Abs-Bin (− 1)

Bit :: [*bin*,*bit*] => *bin* (**infixl** *BIT 90*)
— That is, 2w+b
w BIT b == Abs-Bin ((case b of B0 => 0 | B1 => 1) + Rep-Bin w + Rep-Bin w)

axclass
number < type — for numeric types: nat, int, real, ...

consts
number-of :: *bin* => *'a*::*number*

syntax
-Numeral :: *num-const* => *'a* (-)
Numeral0 :: *'a*
Numeral1 :: *'a*

translations
Numeral0 == number-of Numeral.Pls
Numeral1 == number-of (Numeral.Pls BIT bit.B1)

setup *NumeralSyntax.setup*

syntax (*xsymbols*)
-square :: *'a* => *'a* ((-^{2}) [*1000*] *999*)

syntax (*HTML* **output**)
-square :: *'a* => *'a* ((-^{2}) [*1000*] *999*)
syntax (**output**)
-square :: *'a* => *'a* ((- ^/ *2*) [*81*] *80*)
translations
x^2 == *x^2*
x^2 <= *x^(2::nat)*

lemma *Let-number-of* [*simp*]: *Let* (*number-of v*) *f* == *f* (*number-of v*)
— Unfold all *let*s involving constants
by (*simp add*: *Let-def*)

lemma *Let-0* [*simp*]: *Let 0 f* == *f 0*
by (*simp add*: *Let-def*)

lemma *Let-1* [*simp*]: *Let 1 f* == *f 1*
by (*simp add*: *Let-def*)

constdefs
bin-succ :: *bin=>bin*
bin-succ w == *Abs-Bin*(*Rep-Bin w* + *1*)

bin-pred :: *bin=>bin*
bin-pred w == *Abs-Bin*(*Rep-Bin w* − *1*)

bin-minus :: *bin=>bin*
bin-minus w == *Abs-Bin*(− (*Rep-Bin w*))

bin-add :: [*bin*,*bin*]=>*bin*
bin-add v w == *Abs-Bin*(*Rep-Bin v* + *Rep-Bin w*)

bin-mult :: [*bin*,*bin*]=>*bin*
bin-mult v w == *Abs-Bin*(*Rep-Bin v* ∗ *Rep-Bin w*)

lemmas *Bin-simps* =
bin-succ-def bin-pred-def bin-minus-def bin-add-def bin-mult-def
Pls-def Min-def Bit-def Abs-Bin-inverse Rep-Bin-inverse Bin-def

Removal of leading zeroes

lemma *Pls-0-eq* [*simp*]: *Numeral.Pls BIT bit.B0* = *Numeral.Pls*
by (*simp add*: *Bin-simps*)

lemma *Min-1-eq* [*simp*]: *Numeral.Min BIT bit.B1* = *Numeral.Min*
by (*simp add*: *Bin-simps*)

25.1 The Functions *bin-succ*, *bin-pred* and *bin-minus*

lemma *bin-succ-Pls* [*simp*]: *bin-succ Numeral.Pls = Numeral.Pls BIT bit.B1*
by (*simp add*: *Bin-simps*)

lemma *bin-succ-Min* [*simp*]: *bin-succ Numeral.Min = Numeral.Pls*
by (*simp add*: *Bin-simps*)

lemma *bin-succ-1* [*simp*]: *bin-succ*(*w BIT bit.B1*) = (*bin-succ w*) *BIT bit.B0*
by (*simp add*: *Bin-simps add-ac*)

lemma *bin-succ-0* [*simp*]: *bin-succ*(*w BIT bit.B0*) = *w BIT bit.B1*
by (*simp add*: *Bin-simps add-ac*)

lemma *bin-pred-Pls* [*simp*]: *bin-pred Numeral.Pls = Numeral.Min*
by (*simp add*: *Bin-simps*)

lemma *bin-pred-Min* [*simp*]: *bin-pred Numeral.Min = Numeral.Min BIT bit.B0*
by (*simp add*: *Bin-simps diff-minus*)

lemma *bin-pred-1* [*simp*]: *bin-pred*(*w BIT bit.B1*) = *w BIT bit.B0*
by (*simp add*: *Bin-simps*)

lemma *bin-pred-0* [*simp*]: *bin-pred*(*w BIT bit.B0*) = (*bin-pred w*) *BIT bit.B1*
by (*simp add*: *Bin-simps diff-minus add-ac*)

lemma *bin-minus-Pls* [*simp*]: *bin-minus Numeral.Pls = Numeral.Pls*
by (*simp add*: *Bin-simps*)

lemma *bin-minus-Min* [*simp*]: *bin-minus Numeral.Min = Numeral.Pls BIT bit.B1*
by (*simp add*: *Bin-simps*)

lemma *bin-minus-1* [*simp*]:
bin-minus (*w BIT bit.B1*) = *bin-pred* (*bin-minus w*) *BIT bit.B1*
by (*simp add*: *Bin-simps add-ac diff-minus*)

lemma *bin-minus-0* [*simp*]: *bin-minus*(*w BIT bit.B0*) = (*bin-minus w*) *BIT bit.B0*
by (*simp add*: *Bin-simps*)

25.2 Binary Addition and Multiplication: *bin-add* and *bin-mult*

lemma *bin-add-Pls* [*simp*]: *bin-add Numeral.Pls w = w*
by (*simp add*: *Bin-simps*)

lemma *bin-add-Min* [*simp*]: *bin-add Numeral.Min w = bin-pred w*
by (*simp add*: *Bin-simps diff-minus add-ac*)

lemma *bin-add-BIT-11* [*simp*]:
bin-add (*v BIT bit.B1*) (*w BIT bit.B1*) = *bin-add v* (*bin-succ w*) *BIT bit.B0*

by (*simp add*: *Bin-simps add-ac*)

lemma *bin-add-BIT-10* [*simp*]:
bin-add (*v BIT bit.B1*) (*w BIT bit.B0*) = (*bin-add v w*) *BIT bit.B1*
by (*simp add*: *Bin-simps add-ac*)

lemma *bin-add-BIT-0* [*simp*]:
bin-add (*v BIT bit.B0*) (*w BIT y*) = *bin-add v w BIT y*
by (*simp add*: *Bin-simps add-ac*)

lemma *bin-add-Pls-right* [*simp*]: *bin-add w Numeral.Pls* = *w*
by (*simp add*: *Bin-simps*)

lemma *bin-add-Min-right* [*simp*]: *bin-add w Numeral.Min* = *bin-pred w*
by (*simp add*: *Bin-simps diff-minus*)

lemma *bin-mult-Pls* [*simp*]: *bin-mult Numeral.Pls w* = *Numeral.Pls*
by (*simp add*: *Bin-simps*)

lemma *bin-mult-Min* [*simp*]: *bin-mult Numeral.Min w* = *bin-minus w*
by (*simp add*: *Bin-simps*)

lemma *bin-mult-1* [*simp*]:
bin-mult (*v BIT bit.B1*) *w* = *bin-add* ((*bin-mult v w*) *BIT bit.B0*) *w*
by (*simp add*: *Bin-simps add-ac left-distrib*)

lemma *bin-mult-0* [*simp*]: *bin-mult* (*v BIT bit.B0*) *w* = (*bin-mult v w*) *BIT bit.B0*
by (*simp add*: *Bin-simps left-distrib*)

25.3 Converting Numerals to Rings: *number-of*

axclass *number-ring* ⊆ *number*, *comm-ring-1*
number-of-eq: *number-of w* = *of-int* (*Rep-Bin w*)

lemma *number-of-succ*:
number-of(*bin-succ w*) = (*1* + *number-of w* ::*'a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps*)

lemma *number-of-pred*:
number-of(*bin-pred w*) = (− *1* + *number-of w* ::*'a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps*)

lemma *number-of-minus*:
number-of(*bin-minus w*) = (− (*number-of w*)::*'a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps*)

lemma *number-of-add*:
number-of(*bin-add v w*) = (*number-of v* + *number-of w*::*'a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps*)

lemma *number-of-mult*:
number-of(*bin-mult v w*) = (*number-of v* ∗ *number-of w*::'*a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps*)

The correctness of shifting. But it doesn't seem to give a measurable speed-up.

lemma *double-number-of-BIT*:
(*1*+*1*) ∗ *number-of w* = (*number-of* (*w BIT bit.B0*) ::'*a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps left-distrib*)

Converting numerals 0 and 1 to their abstract versions

lemma *numeral-0-eq-0* [*simp*]: *Numeral0* = (*0*::'*a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps*)

lemma *numeral-1-eq-1* [*simp*]: *Numeral1* = (*1*::'*a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps*)

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*
by (*simp add*: *number-of-eq Bin-simps*)

lemma *mult-minus1* [*simp*]: −*1* ∗ *z* = −(*z*::'*a*::*number-ring*)
by (*simp add*: *numeral-m1-eq-minus-1*)

lemma *mult-minus1-right* [*simp*]: *z* ∗ −*1* = −(*z*::'*a*::*number-ring*)
by (*simp add*: *numeral-m1-eq-minus-1*)

lemma *minus-number-of-mult* [*simp*]:
− (*number-of w*) ∗ *z* = *number-of*(*bin-minus w*) ∗ (*z*::'*a*::*number-ring*)
by (*simp add*: *number-of-minus*)

Subtraction

lemma *diff-number-of-eq*:
number-of v − *number-of w* =
(*number-of*(*bin-add v* (*bin-minus w*))::'*a*::*number-ring*)
by (*simp add*: *diff-minus number-of-add number-of-minus*)

lemma *number-of-Pls*: *number-of Numeral.Pls* = (*0*::'*a*::*number-ring*)
by (*simp add*: *number-of-eq Bin-simps*)

lemma *number-of-Min*: *number-of Numeral.Min* = (− *1*::'*a*::*number-ring*)

by (*simp add*: *number-of-eq Bin-simps*)

lemma *number-of-BIT*:
number-of (*w BIT x*) = (*case x of bit.B0 => 0 | bit.B1 => (1*::*'a*::*number-ring*))
+
(*number-of w*) + (*number-of w*)
by (*simp add*: *number-of-eq Bin-simps split*: *bit.split*)

25.4 Equality of Binary Numbers

First version by Norbert Voelker

lemma *eq-number-of-eq*:
((*number-of x*::*'a*::*number-ring*) = *number-of y*) =
iszero (*number-of* (*bin-add x* (*bin-minus y*)) :: *'a*)
by (*simp add*: *iszero-def compare-rls number-of-add number-of-minus*)

lemma *iszero-number-of-Pls*: *iszero* ((*number-of Numeral.Pls*)::*'a*::*number-ring*)
by (*simp add*: *iszero-def numeral-0-eq-0*)

lemma *nonzero-number-of-Min*: ~ *iszero* ((*number-of Numeral.Min*)::*'a*::*number-ring*)
by (*simp add*: *iszero-def numeral-m1-eq-minus-1 eq-commute*)

25.5 Comparisons, for Ordered Rings

lemma *double-eq-0-iff*: (*a* + *a* = *0*) = (*a* = (*0*::*'a*::*ordered-idom*))
proof −
have *a* + *a* = (*1*+*1*)∗*a* **by** (*simp add*: *left-distrib*)
with *zero-less-two* [**where** *'a* = *'a*]
show *?thesis* **by** *force*
qed

lemma *le-imp-0-less*:
assumes *le*: *0* ≤ *z* **shows** (*0*::*int*) < *1* + *z*
proof −
have *0* ≤ *z* **.**
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-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 < 1 + (int n + int 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
```

The premise involving $\mathbb{Z}$ prevents $a = (1::'a) \,/\, (2::'a)$.

```
lemma Ints-odd-nonzero: a ∈ Ints ==> 1 + a + a ≠ (0::'a::ordered-idom)
proof (unfold Ints-def)
  assume a ∈ range of-int
  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
by (simp add: number-of-eq Ints-def)


lemma iszero-number-of-BIT:
     iszero (number-of (w BIT x)::'a) =
      (x=bit.B0 & iszero (number-of w::'a::{ordered-idom,number-ring}))
by (simp add: iszero-def number-of-eq Bin-simps double-eq-0-iff
              Ints-odd-nonzero Ints-def split: bit.split)

lemma iszero-number-of-0:
     iszero (number-of (w BIT bit.B0) :: 'a::{ordered-idom,number-ring}) =
      iszero (number-of w :: 'a)
by (simp only: iszero-number-of-BIT simp-thms)

lemma iszero-number-of-1:
    ~ iszero (number-of (w BIT bit.B1)::'a::{ordered-idom,number-ring})
by (simp add: iszero-number-of-BIT)
```

25.6 The Less-Than Relation

```
lemma less-number-of-eq-neg:
   ((number-of x::'a::{ordered-idom,number-ring}) < number-of y)
```

```
    = neg (number-of (bin-add x (bin-minus y)) :: 'a)
apply (subst less-iff-diff-less-0)
apply (simp add: neg-def diff-minus number-of-add number-of-minus)
done
```

If *Numeral0* is rewritten to 0 then this rule can't be applied: *Numeral0* IS *Numeral0*

```
lemma not-neg-number-of-Pls:
    ~ neg (number-of Numeral.Pls ::'a::{ordered-idom,number-ring})
by (simp add: neg-def numeral-0-eq-0)

lemma neg-number-of-Min:
    neg (number-of Numeral.Min ::'a::{ordered-idom,number-ring})
by (simp add: neg-def zero-less-one numeral-m1-eq-minus-1)

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: int-Suc
                       add: int-Suc0-eq-1 [symmetric] zadd-int compare-rls)
qed
```

The premise involving $\mathbb{Z}$ prevents $a = (1::'a) \,/\, (2::'a)$.

```
lemma Ints-odd-less-0:
    a ∈ Ints ==> (1 + a + a < 0) = (a < (0::'a::ordered-idom))
proof (unfold Ints-def)
  assume a ∈ range of-int
  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

lemma neg-number-of-BIT:
```

neg (number-of (w BIT x)::'a) =
neg (number-of w :: 'a::{ordered-idom,number-ring})
by (*simp add*: *neg-def number-of-eq Bin-simps double-less-0-iff*
Ints-odd-less-0 Ints-def split: *bit.split*)

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]

lemma *le-number-of-eq*:
((number-of x::'a::{ordered-idom,number-ring}) ≤ number-of y)
= (~ (neg (number-of (bin-add y (bin-minus x)) :: 'a)))
by (*simp add*: *le-number-of-eq-not-less less-number-of-eq-neg*)

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*

25.7 Simplification of arithmetic operations on integer constants.

lemmas *bin-arith-extra-simps =*
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 *bin-arith-simps =*
Numeral.bit.distinct
Pls-0-eq Min-1-eq
bin-pred-Pls bin-pred-Min bin-pred-1 bin-pred-0
bin-succ-Pls bin-succ-Min bin-succ-1 bin-succ-0
bin-add-Pls bin-add-Min bin-add-BIT-0 bin-add-BIT-10 bin-add-BIT-11
bin-minus-Pls bin-minus-Min bin-minus-1 bin-minus-0
bin-mult-Pls bin-mult-Min bin-mult-1 bin-mult-0
bin-add-Pls-right bin-add-Min-right
abs-zero abs-one bin-arith-extra-simps

Simplification of relational operations

lemmas *bin-rel-simps* =
eq-number-of-eq iszero-number-of-Pls nonzero-number-of-Min
iszero-number-of-0 iszero-number-of-1
less-number-of-eq-neg
not-neg-number-of-Pls not-neg-0 not-neg-1 not-iszero-1
neg-number-of-Min neg-number-of-BIT
le-number-of-eq

declare *bin-arith-extra-simps* [*simp*]
declare *bin-rel-simps* [*simp*]

25.8 Simplification of arithmetic when nested to the right

lemma *add-number-of-left* [*simp*]:
number-of v + (*number-of w* + *z*) =
(*number-of*(*bin-add 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*(*bin-mult 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(*bin-add 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 (*bin-add v* (*bin-minus w*)) + (*c*::'*a*::*number-ring*)
apply (*subst diff-number-of-eq* [*symmetric*])
apply (*simp only*: *compare-rls*)
done

end

26 IntArith: Integer arithmetic

theory *IntArith*
imports *Numeral*
uses (*int-arith1.ML*)
begin

Duplicate: can't understand why it's necessary

declare *numeral-0-eq-0* [*simp*]

26.1 Instantiating Binary Arithmetic for the Integers

instance
int :: *number* **..**

defs (**overloaded**)
int-number-of-def: (*number-of w* :: *int*) == *of-int* (*Rep-Bin w*)
— the type constraint is essential!

instance *int* :: *number-ring*
by (*intro-classes*, *simp add*: *int-number-of-def*)

26.2 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*

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

26.3 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-eq-neg numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op* <, *OF less-number-of-eq-neg numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op* <, *OF less-number-of-eq-neg refl numeral-0-eq-0*, *standard*]
binop-eq [*of op* <, *OF less-number-of-eq-neg 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-eq numeral-0-eq-0 refl*, *standard*]
binop-eq [*of op* ≤, *OF le-number-of-eq numeral-1-eq-1 refl*, *standard*]
binop-eq [*of op* ≤, *OF le-number-of-eq refl numeral-0-eq-0*, *standard*]
binop-eq [*of op* ≤, *OF le-number-of-eq refl numeral-1-eq-1*, *standard*]

lemmas *arith-special* =
add-special diff-special eq-special less-special le-special

use *int-arith1.ML*
setup *int-arith-setup*

26.4 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*:
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-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-ring*)
by (*subst mult-commute*, *rule mult-2*)

26.5 More Inequality Reasoning

lemma *zless-add1-eq*: (*w* < *z* + (1::*int*)) = (*w*<*z* | *w*=*z*)
by *arith*

lemma *add1-zle-eq*: (*w* + (1::*int*) ≤ *z*) = (*w*<*z*)
by *arith*

lemma *zle-diff1-eq* [*simp*]: (*w* ≤ *z* − (1::*int*)) = (*w*<*z*)
by *arith*

lemma *zle-add1-eq-le* [*simp*]: (*w* < *z* + (1::*int*)) = (*w*≤*z*)
by *arith*

lemma *int-one-le-iff-zero-less*: ((1::*int*) ≤ *z*) = (0 < *z*)
by *arith*

26.6 The Functions *nat* and *int*

Simplify the terms *int 0*, *int* (*Suc 0*) and *w* + − *z*

declare *Zero-int-def* [*symmetric*, *simp*]
declare *One-int-def* [*symmetric*, *simp*]

cooper.ML refers to this theorem

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 = int-eq-iff* [*of - number-of v*, *standard*]
declare *int-eq-iff-number-of* [*simp*]

lemma *split-nat* [*arith-split*]:
  P(nat(i::int)) = ((∀ n. i = int n ⟶ P n) & (i < 0 ⟶ P 0))
  (**is** *?P = (?L & ?R)*)
proof (*cases i < 0*)
  case *True* **thus** *?thesis* **by** *simp*
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

lemma *zdiff-int*: *n ≤ m ==> int m − int n = int (m−n)*
by (*induct m n rule*: *diff-induct*, *simp-all*)

lemma *nat-mult-distrib*: *(0::int) ≤ z ==> nat (z*z′) = nat z * nat z′*
apply (*case-tac 0 ≤ z′*)
apply (*rule inj-int* [*THEN injD*])
apply (*simp add*: *int-mult zero-le-mult-iff*)
apply (*simp add*: *mult-le-0-iff*)
done

lemma *nat-mult-distrib-neg*: *z ≤ (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 (*case-tac z=0* | *w=0*)
apply (*auto simp add*: *abs-if nat-mult-distrib* [*symmetric*]
nat-mult-distrib-neg [*symmetric*] *mult-less-0-iff*)
done

26.7 Induction principles for int

theorem *int-ge-induct*[*case-names base step*,*induct set*:*int*]:
assumes *ge*: *k* ≤ (*i*::*int*) **and**
base: *P*(*k*) **and**
step: ⋀*i*. ⟦*k* ≤ *i*; *P i*⟧ ⟹ *P*(*i*+*1*)
shows *P i*
proof −
{ fix *n* **have** ⋀*i*::*int*. *n* = *nat*(*i*−*k*) ⟹ *k* ≤ *i* ⟹ *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*((*i* − *1*) − *k*) **by** *arith*
moreover
have *ki1*: *k* ≤ *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: ⋀*i*. ⟦*k* < *i*; *P i*⟧ ⟹ *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* ≤ (*k*::*int*) **and**
base: *P*(*k*) **and**
step: ⋀*i*. ⟦*i* ≤ *k*; *P i*⟧ ⟹ *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
```

26.8 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))
apply (induct-tac n, simp)
apply (intro strip)
apply (erule impE, simp)
apply (erule-tac x = n in allE, simp)
apply (case-tac k = f (n+1) )
 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*)
apply *simp*
apply (*erule impE*)
apply (*intro strip*)
apply (*erule-tac x* = *i*+*m* **in** *allE*, *arith*)
apply (*erule exE*)
apply (*rule-tac x* = *i*+*m* **in** *exI*, *arith*)
done

26.9 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 (*case-tac* |*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

ML
⟪
val zle-diff1-eq = *thm zle-diff1-eq*;

val zle-add1-eq-le = thm zle-add1-eq-le;
val nonneg-eq-int = thm nonneg-eq-int;
val abs-minus-one = thm abs-minus-one;
val of-int-number-of-eq = thmof-int-number-of-eq;
val nat-eq-iff = thm nat-eq-iff;
val nat-eq-iff2 = thm nat-eq-iff2;
val nat-less-iff = thm nat-less-iff;
val int-eq-iff = thm int-eq-iff;
val nat-0 = thm nat-0;
val nat-1 = thm nat-1;
val nat-2 = thm nat-2;
val nat-less-eq-zless = thm nat-less-eq-zless;
val nat-le-eq-zle = thm nat-le-eq-zle;

val nat-intermed-int-val = thm nat-intermed-int-val;
val pos-zmult-eq-1-iff = thm pos-zmult-eq-1-iff;
val zmult-eq-1-iff = thm zmult-eq-1-iff;
val nat-add-distrib = thm nat-add-distrib;
val nat-diff-distrib = thm nat-diff-distrib;
val nat-mult-distrib = thm nat-mult-distrib;
val nat-mult-distrib-neg = thm nat-mult-distrib-neg;
val nat-abs-mult-distrib = thm nat-abs-mult-distrib;
⟫

end

27 SetInterval: Set intervals

theory *SetInterval*
imports *IntArith*
begin

constdefs
lessThan :: ('*a*::*ord*) => '*a set* ((*1*{..<-}))
{..<*u*} == {*x*. *x*<*u*}

atMost :: ('*a*::*ord*) => '*a set* ((*1*{..-}))
{..*u*} == {*x*. *x*<=*u*}

greaterThan :: ('*a*::*ord*) => '*a set* ((*1*{-<..}))
{*l*<..} == {*x*. *l*<*x*}

atLeast :: ('*a*::*ord*) => '*a set* ((*1*{-..}))
{*l*..} == {*x*. *l*<=*x*}

greaterThanLessThan :: ['*a*::*ord*, '*a*] => '*a set* ((*1*{-<..<-}))
{*l*<..<*u*} == {*l*<..} *Int* {..<*u*}

atLeastLessThan :: [*'a*::*ord*, *'a*] => *'a set* ((*1*{-..<-}))
{*l*..<*u*} == {*l*..} *Int* {..<*u*}

greaterThanAtMost :: [*'a*::*ord*, *'a*] => *'a set* ((*1*{-<..-}))
{*l*<..*u*} == {*l*<..} *Int* {..*u*}

atLeastAtMost :: [*'a*::*ord*, *'a*] => *'a set* ((*1*{-..-}))
{*l*..*u*} == {*l*..} *Int* {..*u*}

syntax
-lessThan :: (*'a*::*ord*) => *'a set* ((*1*{..-'(}))
-greaterThan :: (*'a*::*ord*) => *'a set* ((*1*{')-..}))
-greaterThanLessThan :: [*'a*::*ord*, *'a*] => *'a set* ((*1*{')-..-'(}))
-atLeastLessThan :: [*'a*::*ord*, *'a*] => *'a set* ((*1*{-..-'(}))
-greaterThanAtMost :: [*'a*::*ord*, *'a*] => *'a set* ((*1*{')-..-}))
translations
{..*m*(} => {..<*m*}
{)*m*..} => {*m*<..}
{)*m*..*n*(} => {*m*<..<*n*}
{*m*..*n*(} => {*m*..<*n*}
{)*m*..*n*} => {*m*<..*n*}

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* :: *nat* => *nat* => *'b set* => *'b set* ((*3UN* -<=-./ -) *10*)
@*UNION-less* :: *nat* => *nat* => *'b set* => *'b set* ((*3UN* -<-./ -) *10*)
@*INTER-le* :: *nat* => *nat* => *'b set* => *'b set* ((*3INT* -<=-./ -) *10*)
@*INTER-less* :: *nat* => *nat* => *'b set* => *'b set* ((*3INT* -<-./ -) *10*)

syntax (*input*)
@*UNION-le* :: *nat* => *nat* => *'b set* => *'b set* ((*3*⋃ -≤-./ -) *10*)
@*UNION-less* :: *nat* => *nat* => *'b set* => *'b set* ((*3*⋃ -<-./ -) *10*)
@*INTER-le* :: *nat* => *nat* => *'b set* => *'b set* ((*3*⋂ -≤-./ -) *10*)
@*INTER-less* :: *nat* => *nat* => *'b set* => *'b set* ((*3*⋂ -<-./ -) *10*)

syntax (*xsymbols*)
@*UNION-le* :: *nat* ⇒ *nat* => *'b set* => *'b set* ((*3*⋃(*00*_ ≤ _)/ -) *10*)
@*UNION-less* :: *nat* ⇒ *nat* => *'b set* => *'b set* ((*3*⋃(*00*_ < _)/ -) *10*)
@*INTER-le* :: *nat* ⇒ *nat* => *'b set* => *'b set* ((*3*⋂(*00*_ ≤ _)/ -) *10*)
@*INTER-less* :: *nat* ⇒ *nat* => *'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*

27.1 Various equivalences

lemma *lessThan-iff* [*iff*]: $(i\text{: } lessThan\ k) = (i<k)$
by (*simp add*: *lessThan-def*)

lemma *Compl-lessThan* [*simp*]:
$!!k\text{::}\ 'a\text{::}linorder.\ -lessThan\ k = atLeast\ k$
apply (*auto simp add*: *lessThan-def atLeast-def*)
done

lemma *single-Diff-lessThan* [*simp*]: $!!k\text{::}\ 'a\text{::}order.\ \{k\} - lessThan\ k = \{k\}$
by *auto*

lemma *greaterThan-iff* [*iff*]: $(i\text{: } greaterThan\ k) = (k<i)$
by (*simp add*: *greaterThan-def*)

lemma *Compl-greaterThan* [*simp*]:
$!!k\text{::}\ 'a\text{::}linorder.\ -greaterThan\ k = atMost\ k$
apply (*simp add*: *greaterThan-def atMost-def le-def*, *auto*)
done

lemma *Compl-atMost* [*simp*]: $!!k\text{::}\ 'a\text{::}linorder.\ -atMost\ k = greaterThan\ k$
apply (*subst Compl-greaterThan* [*symmetric*])
apply (*rule double-complement*)
done

lemma *atLeast-iff* [*iff*]: $(i\text{: } atLeast\ k) = (k<=i)$
by (*simp add*: *atLeast-def*)

lemma *Compl-atLeast* [*simp*]:
$!!k\text{::}\ 'a\text{::}linorder.\ -atLeast\ k = lessThan\ k$
apply (*simp add*: *lessThan-def atLeast-def le-def*, *auto*)
done

lemma *atMost-iff* [*iff*]: $(i\text{: } atMost\ k) = (i<=k)$
by (*simp add*: *atMost-def*)

lemma *atMost-Int-atLeast*: $!!n\text{::}\ 'a\text{::}order.\ atMost\ n\ Int\ atLeast\ n = \{n\}$
by (*blast intro*: *order-antisym*)

27.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-subset-iff* [*iff*]:
$(atLeast\ x \subseteq atLeast\ y) = (y \leq (x\text{::}'a\text{::}order))$
by (*blast intro*: *order-trans*)

lemma *atLeast-eq-iff* [*iff*]:
$(atLeast\ x = atLeast\ y) = (x = (y\text{::}'a\text{::}linorder))$
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 add*: *greaterThan-subset-iff*)
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 add*: *lessThan-subset-iff*)
done

27.3 Two-sided intervals

lemma *greaterThanLessThan-iff* [*simp*]:
(*i* : {*l*<..<*u*}) = (*l* < *i* & *i* < *u*)
by (*simp add*: *greaterThanLessThan-def*)

lemma *atLeastLessThan-iff* [*simp*]:
(*i* : {*l*..<*u*}) = (*l* <= *i* & *i* < *u*)
by (*simp add*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-iff* [*simp*]:
(*i* : {*l*<..*u*}) = (*l* < *i* & *i* <= *u*)
by (*simp add*: *greaterThanAtMost-def*)

lemma *atLeastAtMost-iff* [*simp*]:
(*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).

27.3.1 Emptyness and singletons

lemma *atLeastAtMost-empty* [*simp*]: $n < m ==> \{m::'a::order..n\} = \{\}$
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

lemma *atLeastLessThan-empty*[*simp*]: $n \leq m ==> \{m..<n::'a::order\} = \{\}$
by (*auto simp add*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-empty*[*simp*]:$l \leq k ==> \{k<..(l::'a::order)\} = \{\}$
by(*auto simp*:*greaterThanAtMost-def greaterThan-def atMost-def*)

lemma *greaterThanLessThan-empty*[*simp*]:$l \leq k ==> \{k<..(l::'a::order)\} = \{\}$
by(*auto simp*:*greaterThanLessThan-def greaterThan-def lessThan-def*)

lemma *atLeastAtMost-singleton* [*simp*]: $\{a::'a::order..a\} = \{a\}$
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

27.4 Intervals of natural numbers

27.4.1 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*

27.4.2 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*

27.4.3 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*

27.4.4 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*

27.4.5 The Constant *atLeastLessThan*

But not a simprule because some concepts are better left in terms of *atLeastLessThan*

lemma *atLeast0LessThan*: {*0*::*nat*..<*n*} = {..<*n*}
by(*simp* *add*:*lessThan-def* *atLeastLessThan-def*)

27.4.6 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:
{*m*..<*Suc* *n*} = (*if* *m* ≤ *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*)

27.4.7 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 arith+*
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 arith+*
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=1*] **by** *simp*

corollary *image-Suc-atLeastLessThan*[*simp*]:
Suc ' {i..<j} = {Suc i..<Suc j}
using *image-add-atLeastLessThan*[**where** *k=1*] **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

27.4.8 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*)

lemma *bounded-nat-set-is-finite*:
(ALL i:N. i < (n::nat)) ==> finite N
— A bounded set of natural numbers is finite.
apply (*rule finite-subset*)
apply (*rule-tac* [*2*] *finite-lessThan*, *auto*)
done

27.4.9 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 *arith*
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*)

27.5 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*)

27.5.1 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*)

27.5.2 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*: *compare-rls*)
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*)

27.6 Lemmas useful with the summation operator setsum

For examples, see Algebra/poly/UnivPoly2.thy

27.6.1 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*

27.6.2 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*

27.6.3 Some Differences

lemma *ivl-diff*[*simp*]:
$i \leq n \Longrightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$
by(*auto*)

27.6.4 Some Subset Conditions

lemma *ivl-subset*[*simp*]:
$(\{i..<j\} \subseteq \{m..<n\}) = (j \leq i \mid m \leq i \;\&\; j \leq (n::'a::linorder))$
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

27.7 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*∑⁻ᶜ *- = -*) [*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* == *setsum* (%*x*. *t*) {*a*..*b*}
∑ *x*=*a*..<*b*. *t* == *setsum* (%*x*. *t*) {*a*..<*b*}
∑ *i*≤*n*. *t* == *setsum* (λ*i*. *t*) {..*n*}
∑ *i*<*n*. *t* == *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 $\sum x = 0..<n.\ e$ rather than $\sum x{<}n.\ e$: *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-add-nat-ivl*: $[\![m \le n;\ n \le p]\!] \Longrightarrow$
$setsum\ f\ \{m..<n\} + setsum\ f\ \{n..<p\} = setsum\ f\ \{m..<p::nat\}$
by (*simp add*:*setsum-Un-disjoint*[*symmetric*] *ivl-disj-int ivl-disj-un*)

lemma *setsum-diff-nat-ivl*:
fixes $f :: nat \Rightarrow 'a::ab\text{-}group\text{-}add$
shows $[\![m \le n;\ n \le p]\!] \Longrightarrow$
$setsum\ f\ \{m..<p\} - setsum\ f\ \{m..<n\} = setsum\ f\ \{n..<p\}$
using *setsum-add-nat-ivl* [*of m n p f*,*symmetric*]
apply (*simp add*: *add-ac*)
done

27.8 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=1*,*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=1*,*simplified*])

27.9 The formula for geometric sums

lemma *geometric-sum*:
x ~= 1 ==> (∑ i=0..<n. x ^ i) =
(x ^ n − 1) / (x − 1::'a::{field, recpower, division-by-zero})
apply (*induct n*, *auto*)
apply (*rule-tac c = x − 1* **in** *field-mult-cancel-right-lemma*)
apply (*auto simp add*: *mult-assoc left-distrib*)
apply (*simp add*: *right-distrib diff-minus mult-commute power-Suc*)
done

ML
⟪
val Compl-atLeast = thm Compl-atLeast;
val Compl-atMost = thm Compl-atMost;
val Compl-greaterThan = thm Compl-greaterThan;
val Compl-lessThan = thm Compl-lessThan;
val INT-greaterThan-UNIV = thm INT-greaterThan-UNIV;
val UN-atLeast-UNIV = thm UN-atLeast-UNIV;
val UN-atMost-UNIV = thm UN-atMost-UNIV;
val UN-lessThan-UNIV = thm UN-lessThan-UNIV;
val atLeastAtMost-def = thm atLeastAtMost-def;
val atLeastAtMost-iff = thm atLeastAtMost-iff;
val atLeastLessThan-def = thm atLeastLessThan-def;
val atLeastLessThan-iff = thm atLeastLessThan-iff;
val atLeast-0 = thm atLeast-0;
val atLeast-Suc = thm atLeast-Suc;
val atLeast-def = thm atLeast-def;
val atLeast-iff = thm atLeast-iff;
val atMost-0 = thm atMost-0;
val atMost-Int-atLeast = thm atMost-Int-atLeast;
val atMost-Suc = thm atMost-Suc;
val atMost-def = thm atMost-def;

val atMost-iff = thm atMost-iff;
val greaterThanAtMost-def = thm greaterThanAtMost-def;
val greaterThanAtMost-iff = thm greaterThanAtMost-iff;
val greaterThanLessThan-def = thm greaterThanLessThan-def;
val greaterThanLessThan-iff = thm greaterThanLessThan-iff;
val greaterThan-0 = thm greaterThan-0;
val greaterThan-Suc = thm greaterThan-Suc;
val greaterThan-def = thm greaterThan-def;
val greaterThan-iff = thm greaterThan-iff;
val ivl-disj-int = thms ivl-disj-int;
val ivl-disj-int-one = thms ivl-disj-int-one;
val ivl-disj-int-singleton = thms ivl-disj-int-singleton;
val ivl-disj-int-two = thms ivl-disj-int-two;
val ivl-disj-un = thms ivl-disj-un;
val ivl-disj-un-one = thms ivl-disj-un-one;
val ivl-disj-un-singleton = thms ivl-disj-un-singleton;
val ivl-disj-un-two = thms ivl-disj-un-two;
val lessThan-0 = thm lessThan-0;
val lessThan-Suc = thm lessThan-Suc;
val lessThan-Suc-atMost = thm lessThan-Suc-atMost;
val lessThan-def = thm lessThan-def;
val lessThan-iff = thm lessThan-iff;
val single-Diff-lessThan = thm single-Diff-lessThan;

val bounded-nat-set-is-finite = thm bounded-nat-set-is-finite;
val finite-atMost = thm finite-atMost;
val finite-lessThan = thm finite-lessThan;
⟩⟩

end

28 Recdef: TFL: recursive function definitions

theory *Recdef*
imports *Wellfounded-Relations Datatype*
uses
(*../TFL/casesplit.ML*)
(*../TFL/utils.ML*)
(*../TFL/usyntax.ML*)
(*../TFL/dcterm.ML*)
(*../TFL/thms.ML*)
(*../TFL/rules.ML*)
(*../TFL/thry.ML*)
(*../TFL/tfl.ML*)
(*../TFL/post.ML*)
(*Tools/recdef-package.ML*)
begin

lemma *tfl-eq-True*: $(x = True) \dashrightarrow x$
by *blast*

lemma *tfl-rev-eq-mp*: $(x = y) \dashrightarrow y \dashrightarrow x$
by *blast*

lemma *tfl-simp-thm*: $(x \dashrightarrow y) \dashrightarrow (x = x') \dashrightarrow (x' \dashrightarrow y)$
by *blast*

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = True$
by *blast*

lemma *tfl-imp-trans*: $(A \dashrightarrow B) \Longrightarrow (B \dashrightarrow C) \Longrightarrow (A \dashrightarrow C)$
by *blast*

lemma *tfl-disj-assoc*: $(a \vee b) \vee c == a \vee (b \vee c)$
by *simp*

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \dashrightarrow R \Longrightarrow Q \dashrightarrow R \Longrightarrow R$
by *blast*

lemma *tfl-exE*: $\exists x.\ P\ x \Longrightarrow \forall x.\ P\ x \dashrightarrow Q \Longrightarrow Q$
by *blast*

use *../TFL/casesplit.ML*
use *../TFL/utils.ML*
use *../TFL/usyntax.ML*
use *../TFL/dcterm.ML*
use *../TFL/thms.ML*
use *../TFL/rules.ML*
use *../TFL/thry.ML*
use *../TFL/tfl.ML*
use *../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 image-cong*

lemma *let-cong* [*recdef-cong*]:
$M = N \Longrightarrow (!!x.\ x = N \Longrightarrow f\ x = g\ x) \Longrightarrow Let\ M\ f = Let\ N\ g$
by (*unfold Let-def*) *blast*

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-pred-nat
wf-same-fst
wf-empty

lemma *insert-None-conv-UNIV*: *insert None* (*range Some*) = *UNIV*
by (*rule set-ext*, *case-tac x*, *auto*)

instance *option* :: (*finite*) *finite*
proof
have *finite* (*UNIV* :: *'a set*) **by** (*rule finite*)
hence *finite* (*insert None* (*Some* ‘ (*UNIV* :: *'a set*))) **by** *simp*
also have *insert None* (*Some* ‘ (*UNIV* :: *'a set*)) = *UNIV*
by (*rule insert-None-conv-UNIV*)
finally show *finite* (*UNIV* :: *'a option set*) .
qed

end

29 IntDiv: The Division Operators div and mod; the Divides Relation dvd

theory *IntDiv*
imports *SetInterval Recdef*
uses (*IntDiv-setup.ML*)
begin

declare *zless-nat-conj* [*simp*]

constdefs
quorem :: (*int***int*) * (*int***int*) => *bool*
— definition of quotient and remainder
quorem == %((*a*,*b*), (*q*,*r*)).
a = *b***q* + *r* &
(*if* 0 < *b then* 0≤*r* & *r*<*b else b*<*r* & *r* ≤ 0)

adjust :: [*int*, *int***int*] => *int***int*
— for the division algorithm
adjust b == %(*q*,*r*). *if* 0 ≤ *r*−*b then* (2**q* + 1, *r*−*b*)
else (2**q*, *r*)

algorithm for the case $a \geq 0,\ b > 0$

consts *posDivAlg* :: *int***int* => *int***int*
recdef *posDivAlg measure* (%(*a*,*b*). *nat*(*a* − *b* + *1*))
  posDivAlg (*a*,*b*) =
    (*if* (*a*<*b* | *b*≤*0*) *then* (*0*,*a*)
     else adjust b (*posDivAlg*(*a*, *2***b*)))

algorithm for the case $a < 0,\ b > 0$

consts *negDivAlg* :: *int***int* => *int***int*
recdef *negDivAlg measure* (%(*a*,*b*). *nat*(− *a* − *b*))
  negDivAlg (*a*,*b*) =
    (*if* (*0*≤*a*+*b* | *b*≤*0*) *then* (−*1*,*a*+*b*)
     else adjust b (*negDivAlg*(*a*, *2***b*)))

algorithm for the general case $b \neq (0::'a)$

constdefs
  negateSnd :: *int***int* => *int***int*
  negateSnd == %(*q*,*r*). (*q*,−*r*)

  divAlg :: *int***int* => *int***int*
  — 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*).
  divAlg ==
    %(*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*))

instance
  int :: *Divides.div* **..** — avoid clash with 'div' token

The operators are defined with reference to the algorithm, which is proved to satisfy the specification.

defs
  div-def: *a div b* == *fst* (*divAlg* (*a*,*b*))
  mod-def: *a mod b* == *snd* (*divAlg* (*a*,*b*))

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 divAlg (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));
```

29.1 Uniqueness and Monotonicity of Quotients and Remainders

lemma *unique-quotient-lemma*:
$[\!|\ b*q' + r' \le b*q + r;\ \ 0 \le r';\ \ r' < b;\ \ r < b\ |\!]$
$\Longrightarrow q' \le (q::int)$
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*:
$[\!|\ b*q' + r' \le b*q + r;\ \ r \le 0;\ \ b < r;\ \ b < r'\ |\!]$
$\Longrightarrow q \le (q'::int)$
by (*rule-tac* $b = -b$ **and** $r = -r'$ **and** $r' = -r$ **in** *unique-quotient-lemma*, *auto*)

lemma *unique-quotient*:
$[\!|\ quorem\ ((a,b), (q,r));\ \ quorem\ ((a,b), (q',r'));\ \ b \neq 0\ |\!]$
$\Longrightarrow q = q'$
apply (*simp add*: *quorem-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*:
[| *quorem* ((*a*,*b*), (*q*,*r*)); *quorem* ((*a*,*b*), (*q'*,*r'*)); *b* ≠ *0* |]
==> *r* = *r'*
apply (*subgoal-tac q* = *q'*)
apply (*simp add*: *quorem-def*)
apply (*blast intro*: *unique-quotient*)
done

29.2 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 ≤ *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* [*rule-format*]:
0 ≤ *a* −−> *0* < *b* −−> *quorem* ((*a*, *b*), *posDivAlg* (*a*, *b*))
apply (*induct-tac a b rule*: *posDivAlg.induct*, *auto*)
apply (*simp-all add*: *quorem-def*)

apply (*simp add*: *posDivAlg-eqn*)

apply (*subst posDivAlg-eqn*, *simp-all*)
apply (*erule splitE*)
apply (*auto simp add*: *right-distrib Let-def*)
done

29.3 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* [*rule-format*]:
  a < 0 −−> 0 < b −−> quorem ((a, b), negDivAlg (a, b))
apply (*induct-tac a b rule*: *negDivAlg.induct*, *auto*)
apply (*simp-all add*: *quorem-def*)

apply (*simp add*: *negDivAlg-eqn*)

apply (*subst negDivAlg-eqn*, *assumption*)
apply (*erule splitE*)
apply (*auto simp add*: *right-distrib Let-def*)
done

29.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma *quorem-0*: *b ≠ 0 ==> quorem ((0,b), (0,0))*
by (*auto simp add*: *quorem-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 *quorem-neg*: *quorem ((−a,−b), qr) ==> quorem ((a,b), negateSnd qr)*
by (*auto simp add*: *split-ifs quorem-def*)

lemma *divAlg-correct*: *b ≠ 0 ==> quorem ((a,b), divAlg(a,b))*
by (*force simp add*: *linorder-neq-iff quorem-0 divAlg-def quorem-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 divAlg-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** *divAlg-correct*)
apply (*auto simp add*: *quorem-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*])

use *IntDiv-setup.ML*

lemma *pos-mod-conj* : $(0::int) < b \Longrightarrow 0 \leq a\ mod\ b\ \&\ a\ mod\ b < b$
apply (*cut-tac* $a = a$ **and** $b = b$ **in** *divAlg-correct*)
apply (*auto simp add*: *quorem-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* : $b < (0::int) \Longrightarrow a\ mod\ b \leq 0\ \&\ b < a\ mod\ b$
apply (*cut-tac* $a = a$ **and** $b = b$ **in** *divAlg-correct*)
apply (*auto simp add*: *quorem-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*]

29.5 General Properties of div and mod

lemma *quorem-div-mod*: $b \neq 0 \Longrightarrow quorem\ ((a, b), (a\ div\ b, a\ mod\ b))$
apply (*cut-tac* $a = a$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*force simp add*: *quorem-def linorder-neq-iff*)
done

lemma *quorem-div*: $[|\ quorem((a,b),(q,r));\ b \neq 0\ |] \Longrightarrow a\ div\ b = q$
by (*simp add*: *quorem-div-mod* [*THEN unique-quotient*])

lemma *quorem-mod*: $[|\ quorem((a,b),(q,r));\ b \neq 0\ |] \Longrightarrow a\ mod\ b = r$
by (*simp add*: *quorem-div-mod* [*THEN unique-remainder*])

lemma *div-pos-pos-trivial*: $[|\ (0::int) \leq a;\ a < b\ |] \Longrightarrow a\ div\ b = 0$
apply (*rule quorem-div*)
apply (*auto simp add*: *quorem-def*)
done

lemma *div-neg-neg-trivial*: [| $a \leq (0::int)$; $b < a$ |] ==> a *div* $b = 0$
apply (*rule quorem-div*)
apply (*auto simp add*: *quorem-def*)
done

lemma *div-pos-neg-trivial*: [| $(0::int) < a$; $a+b \leq 0$ |] ==> a *div* $b = -1$
apply (*rule quorem-div*)
apply (*auto simp add*: *quorem-def*)
done

lemma *mod-pos-pos-trivial*: [| $(0::int) \leq a$; $a < b$ |] ==> a *mod* $b = a$
apply (*rule-tac q = 0* **in** *quorem-mod*)
apply (*auto simp add*: *quorem-def*)
done

lemma *mod-neg-neg-trivial*: [| $a \leq (0::int)$; $b < a$ |] ==> a *mod* $b = a$
apply (*rule-tac q = 0* **in** *quorem-mod*)
apply (*auto simp add*: *quorem-def*)
done

lemma *mod-pos-neg-trivial*: [| $(0::int) < a$; $a+b \leq 0$ |] ==> a *mod* $b = a+b$
apply (*rule-tac q = -1* **in** *quorem-mod*)
apply (*auto simp add*: *quorem-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*: *quorem-div-mod* [*THEN quorem-neg*, *simplified*,
THEN quorem-div, *THEN sym*])

done

lemma *zmod-zminus-zminus* [*simp*]: $(-a)$ *mod* $(-b) = -$ $(a$ *mod* $(b::int))$
apply (*case-tac b = 0*, *simp*)
apply (*subst quorem-div-mod* [*THEN quorem-neg*, *simplified*, *THEN quorem-mod*],
auto)
done

29.6 Laws for div and mod with Unary Minus

lemma *zminus1-lemma*:
quorem((*a*,*b*),(*q*,*r*))
==> *quorem* ((−*a*,*b*), (*if r=0 then* −*q else* −*q* − *1*),
(*if r=0 then 0 else b*−*r*))
by (*force simp add*: *split-ifs quorem-def linorder-neq-iff right-diff-distrib*)

lemma *zdiv-zminus1-eq-if*:
    $b \neq (0::int)$
    ==> $(-a)$ *div* $b =$
        (*if* a *mod* $b = 0$ *then* $-(a$ *div* $b)$ *else* $-(a$ *div* $b) - 1)$
by (*blast intro*: *quorem-div-mod* [*THEN zminus1-lemma*, *THEN quorem-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*: *quorem-div-mod* [*THEN zminus1-lemma*, *THEN quorem-mod*])
done

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*:
    $b \neq (0::int)$
    ==> 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*)

29.7 Division of a Number by Itself

lemma *self-quotient-aux1*: [| $(0::int) < a$; $a = r + a*q$; $r < a$ |] ==> $1 \leq q$
apply (*subgoal-tac* $0 < a*q$)
apply (*simp add*: *zero-less-mult-iff*, *arith*)
done

lemma *self-quotient-aux2*: [| $(0::int) < a$; $a = r + a*q$; $0 \leq r$ |] ==> $q \leq 1$
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*: [| *quorem*((a,a),(q,r)); $a \neq (0::int)$ |] ==> $q = 1$
apply (*simp add*: *split-ifs quorem-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*: [| *quorem*((*a*,*a*),(*q*,*r*)); *a* ≠ (*0*::*int*) |] ==> *r* = *0*
apply (*frule self-quotient*, *assumption*)
apply (*simp add*: *quorem-def*)
done

lemma *zdiv-self* [*simp*]: *a* ≠ *0* ==> *a div a* = (*1*::*int*)
by (*simp add*: *quorem-div-mod* [*THEN self-quotient*])

lemma *zmod-self* [*simp*]: *a mod a* = (*0*::*int*)
apply (*case-tac a* = *0*, *simp*)
apply (*simp add*: *quorem-div-mod* [*THEN self-remainder*])
done

29.8 Computation of Division and Remainder

lemma *zdiv-zero* [*simp*]: (*0*::*int*) *div b* = *0*
by (*simp add*: *div-def divAlg-def*)

lemma *div-eq-minus1*: (*0*::*int*) < *b* ==> −*1 div b* = −*1*
by (*simp add*: *div-def divAlg-def*)

lemma *zmod-zero* [*simp*]: (*0*::*int*) *mod b* = *0*
by (*simp add*: *mod-def divAlg-def*)

lemma *zdiv-minus1*: (*0*::*int*) < *b* ==> −*1 div b* = −*1*
by (*simp add*: *div-def divAlg-def*)

lemma *zmod-minus1*: (*0*::*int*) < *b* ==> −*1 mod b* = *b* − *1*
by (*simp add*: *mod-def divAlg-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 divAlg-def*)

lemma *mod-pos-pos*: [| *0* < *a*; *0* ≤ *b* |] ==> *a mod b* = *snd* (*posDivAlg*(*a*,*b*))
by (*simp add*: *mod-def divAlg-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 divAlg-def*)

lemma *mod-neg-pos*: [| *a* < *0*; *0* < *b* |] ==> *a mod b* = *snd* (*negDivAlg*(*a*,*b*))
by (*simp add*: *mod-def divAlg-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 divAlg-def*)

lemma *mod-pos-neg*:
[| $0 < a$; $b < 0$ |] ==> $a\ mod\ b = snd\ (negateSnd(negDivAlg(-a,-b)))$
by (*simp add*: *mod-def divAlg-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 divAlg-def*)

lemma *mod-neg-neg*:
[| $a < 0$; $b \leq 0$ |] ==> $a\ mod\ b = snd\ (negateSnd(posDivAlg(-a,-b)))$
by (*simp add*: *mod-def divAlg-def*)

Simplify expresions in which div and mod combine numerical constants

lemmas *div-pos-pos-number-of* =
div-pos-pos [*of number-of v number-of w*, *standard*]
declare *div-pos-pos-number-of* [*simp*]

lemmas *div-neg-pos-number-of* =
div-neg-pos [*of number-of v number-of w*, *standard*]
declare *div-neg-pos-number-of* [*simp*]

lemmas *div-pos-neg-number-of* =
div-pos-neg [*of number-of v number-of w*, *standard*]
declare *div-pos-neg-number-of* [*simp*]

lemmas *div-neg-neg-number-of* =
div-neg-neg [*of number-of v number-of w*, *standard*]
declare *div-neg-neg-number-of* [*simp*]

lemmas *mod-pos-pos-number-of* =
mod-pos-pos [*of number-of v number-of w*, *standard*]
declare *mod-pos-pos-number-of* [*simp*]

lemmas *mod-neg-pos-number-of* =
mod-neg-pos [*of number-of v number-of w*, *standard*]
declare *mod-neg-pos-number-of* [*simp*]

lemmas *mod-pos-neg-number-of* =
mod-pos-neg [*of number-of v number-of w*, *standard*]
declare *mod-pos-neg-number-of* [*simp*]

lemmas *mod-neg-neg-number-of* =
mod-neg-neg [*of number-of v number-of w*, *standard*]
declare *mod-neg-neg-number-of* [*simp*]

lemmas *posDivAlg-eqn-number-of* =
posDivAlg-eqn [*of number-of v number-of w*, *standard*]
declare *posDivAlg-eqn-number-of* [*simp*]

lemmas *negDivAlg-eqn-number-of* =
negDivAlg-eqn [*of number-of v number-of w*, *standard*]
declare *negDivAlg-eqn-number-of* [*simp*]

Special-case simplification

lemma *zmod-1* [*simp*]: *a mod* (*1*::*int*) = *0*
apply (*cut-tac a* = *a* **and** *b* = *1* **in** *pos-mod-sign*)
apply (*cut-tac* [*2*] *a* = *a* **and** *b* = *1* **in** *pos-mod-bound*)
apply (*auto simp del*:*pos-mod-bound pos-mod-sign*)
done

lemma *zdiv-1* [*simp*]: *a div* (*1*::*int*) = *a*
by (*cut-tac a* = *a* **and** *b* = *1* **in** *zmod-zdiv-equality*, *auto*)

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* =
div-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]
declare *div-pos-pos-1-number-of* [*simp*]

lemmas *div-pos-neg-1-number-of* =
div-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]
declare *div-pos-neg-1-number-of* [*simp*]

lemmas *mod-pos-pos-1-number-of* =
mod-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]
declare *mod-pos-pos-1-number-of* [*simp*]

lemmas *mod-pos-neg-1-number-of* =
mod-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]
declare *mod-pos-neg-1-number-of* [*simp*]

lemmas *posDivAlg-eqn-1-number-of* =

posDivAlg-eqn [*of* **concl**: *1 number-of w*, *standard*]
declare *posDivAlg-eqn-1-number-of* [*simp*]

lemmas *negDivAlg-eqn-1-number-of* =
negDivAlg-eqn [*of* **concl**: *1 number-of w*, *standard*]
declare *negDivAlg-eqn-1-number-of* [*simp*]

29.9 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

29.10 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*:
[| $(0::int) \leq a$; $0 < b'$; $b' \leq b$ |] ==> a *div* $b \leq a$ *div* b'

apply (*subgoal-tac* $b \neq 0$)
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*:
[| $b'*q' + r' < 0$; $0 \leq r'$; $0 < b'$ |] ==> $q' \leq (0::int)$
apply (*subgoal-tac* $b'*q' < 0$)
apply (*simp add*: *mult-less-0-iff*, *arith*)
done

lemma *zdiv-mono2-neg-lemma*:
[| $b*q + r = b'*q' + r'$; $b'*q' + r' < 0$;
$r < b$; $0 \leq r'$; $0 < b'$; $b' \leq b$ |]
==> $q' \leq (q::int)$
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* $b*q' \leq b'*q'$)
prefer *2* **apply** (*simp add*: *mult-right-mono-neg*, *arith*)
done

lemma *zdiv-mono2-neg*:
[| $a < (0::int)$; $0 < b'$; $b' \leq b$ |] ==> 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 zdiv-mono2-neg-lemma*)
apply (*erule subst*)
apply (*erule subst*, *simp-all*)
done

29.11 More Algebraic Laws for div and mod

proving (a*b) div c = a * (b div c) + a * (b mod c)

lemma *zmult1-lemma*:
[| *quorem*((*b*,*c*),(*q*,*r*)); $c \neq 0$ |]
==> *quorem* ((*a**b*, *c*), (*a***q* + *a***r* *div* *c*, *a***r* *mod* *c*))
by (*force simp add*: *split-ifs quorem-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*: *quorem-div-mod* [*THEN zmult1-lemma*, *THEN quorem-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*: *quorem-div-mod* [*THEN zmult1-lemma*, *THEN quorem-mod*])
done

lemma *zmod-zmult1-eq'*: $(a*b)\ mod\ (c::int) = ((a\ mod\ c) * b)\ mod\ c$
apply (*rule trans*)
apply (*rule-tac* $s = b*a\ mod\ c$ **in** *trans*)
apply (*rule-tac* [2] *zmod-zmult1-eq*)
apply (*simp-all add*: *mult-commute*)
done

lemma *zmod-zmult-distrib*: $(a*b)\ mod\ (c::int) = ((a\ mod\ c) * (b\ mod\ c))\ mod\ c$
apply (*rule zmod-zmult1-eq'* [*THEN trans*])
apply (*rule zmod-zmult1-eq*)
done

lemma *zdiv-zmult-self1* [*simp*]: $b \neq (0::int) ==> (a*b)\ div\ b = a$
by (*simp add*: *zdiv-zmult1-eq*)

lemma *zdiv-zmult-self2* [*simp*]: $b \neq (0::int) ==> (b*a)\ div\ b = a$
by (*subst mult-commute*, *erule zdiv-zmult-self1*)

lemma *zmod-zmult-self1* [*simp*]: $(a*b)\ mod\ b = (0::int)$
by (*simp add*: *zmod-zmult1-eq*)

lemma *zmod-zmult-self2* [*simp*]: $(b*a)\ mod\ b = (0::int)$
by (*simp add*: *mult-commute zmod-zmult1-eq*)

lemma *zmod-eq-0-iff*: $(m\ mod\ d = 0) = (EX\ q::int.\ m = d*q)$
proof
  assume $m\ mod\ d = 0$
  with *zmod-zdiv-equality*[*of m d*] **show** $EX\ q::int.\ m = d*q$ **by** *auto*
next
  assume $EX\ q::int.\ m = d*q$
  thus $m\ mod\ d = 0$ **by** *auto*
qed

lemmas *zmod-eq-0D* = *zmod-eq-0-iff* [*THEN iffD1*]
declare *zmod-eq-0D* [*dest*!]

proving (a+b) div c = a div c + b div c + ((a mod c + b mod c) div c)

lemma *zadd1-lemma*:
    [| $quorem((a,c),(aq,ar))$; $quorem((b,c),(bq,br))$; $c \neq 0$ |]
     ==> $quorem\ ((a+b,\ c),\ (aq\ +\ bq\ +\ (ar+br)\ div\ c,\ (ar+br)\ mod\ c))$
by (*force simp add*: *split-ifs quorem-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 quorem-div-mod quorem-div-mod*] *quorem-div*)
done

lemma *zmod-zadd1-eq*: $(a+b)\ mod\ (c::int) = (a\ mod\ c + b\ mod\ c)\ mod\ c$
apply (*case-tac* $c = 0$, *simp*)
apply (*blast intro*: *zadd1-lemma* [*OF quorem-div-mod quorem-div-mod*] *quorem-mod*)
done

lemma *mod-div-trivial* [*simp*]: $(a\ mod\ b)\ div\ b = (0::int)$
apply (*case-tac* $b = 0$, *simp*)
apply (*auto simp add*: *linorder-neq-iff div-pos-pos-trivial div-neg-neg-trivial*)
done

lemma *mod-mod-trivial* [*simp*]: $(a\ mod\ b)\ mod\ b = a\ mod\ (b::int)$
apply (*case-tac* $b = 0$, *simp*)
apply (*force simp add*: *linorder-neq-iff mod-pos-pos-trivial mod-neg-neg-trivial*)
done

lemma *zmod-zadd-left-eq*: $(a+b)\ mod\ (c::int) = ((a\ mod\ c) + b)\ mod\ c$
apply (*rule trans* [*symmetric*])
apply (*rule zmod-zadd1-eq*, *simp*)
apply (*rule zmod-zadd1-eq* [*symmetric*])
done

lemma *zmod-zadd-right-eq*: $(a+b)\ mod\ (c::int) = (a + (b\ mod\ c))\ mod\ c$
apply (*rule trans* [*symmetric*])
apply (*rule zmod-zadd1-eq*, *simp*)
apply (*rule zmod-zadd1-eq* [*symmetric*])
done

lemma *zdiv-zadd-self1*[*simp*]: $a \neq (0::int) ==> (a+b)\ div\ a = b\ div\ a + 1$
by (*simp add*: *zdiv-zadd1-eq*)

lemma *zdiv-zadd-self2*[*simp*]: $a \neq (0::int) ==> (b+a)\ div\ a = b\ div\ a + 1$
by (*simp add*: *zdiv-zadd1-eq*)

lemma *zmod-zadd-self1*[*simp*]: $(a+b)\ mod\ a = b\ mod\ (a::int)$
apply (*case-tac* $a = 0$, *simp*)
apply (*simp add*: *zmod-zadd1-eq*)
done

lemma *zmod-zadd-self2*[*simp*]: $(b+a)\ mod\ a = b\ mod\ (a::int)$
apply (*case-tac* $a = 0$, *simp*)
apply (*simp add*: *zmod-zadd1-eq*)
done

29.12 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*: [| $(0::int) < c$; $b < r$; $r \leq 0$ |] ==> $b*c < b*(q\ mod\ c) + r$
apply (*subgoal-tac* $b * (c - q\ mod\ c) < r * 1$)
apply (*simp add*: *right-diff-distrib*)
apply (*rule order-le-less-trans*)
apply (*erule-tac* [2] *mult-strict-right-mono*)
apply (*rule mult-left-mono-neg*)
apply (*auto simp add*: *compare-rls add-commute* [*of 1*]
add1-zle-eq pos-mod-bound)
done

lemma *zmult2-lemma-aux2*:
[| $(0::int) < c$; $b < r$; $r \leq 0$ |] ==> $b * (q\ mod\ c) + r \leq 0$
apply (*subgoal-tac* $b * (q\ mod\ c) \leq 0$)
apply *arith*
apply (*simp add*: *mult-le-0-iff*)
done

lemma *zmult2-lemma-aux3*: [| $(0::int) < c$; $0 \leq r$; $r < b$ |] ==> $0 \leq b * (q\ mod\ c) + r$
apply (*subgoal-tac* $0 \leq b * (q\ mod\ c)$)
apply *arith*
apply (*simp add*: *zero-le-mult-iff*)
done

lemma *zmult2-lemma-aux4*: [| $(0::int) < c$; $0 \leq r$; $r < b$ |] ==> $b * (q\ mod\ c) + r < b * c$
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 (*auto simp add*: *compare-rls add-commute* [*of 1*]
add1-zle-eq pos-mod-bound)
done

lemma *zmult2-lemma*: [| *quorem* $((a,b), (q,r))$; $b \neq 0$; $0 < c$ |]
==> *quorem* $((a, b*c), (q\ div\ c, b*(q\ mod\ c) + r))$
by (*auto simp add*: *mult-ac quorem-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*: *quorem-div-mod* [*THEN zmult2-lemma*, *THEN quorem-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*: *quorem-div-mod* [*THEN zmult2-lemma*, *THEN quorem-mod*])
done

29.13 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-zmult2*: *c* ≠ (*0*::*int*) ==> (*a*∗*c*) *div* (*b*∗*c*) = *a div b*
apply (*drule zdiv-zmult-zmult1*)
apply (*auto simp add*: *mult-commute*)
done

29.14 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-neq-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

29.15 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 zmod-zadd1-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 zmod-zadd1-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 --> P 0) &
   (0<k --> (∀ i j. 0≤j & j<k & n = k∗i + j --> P i)) &
   (k<0 --> (∀ i j. k<j & j≤0 & n = k∗i + j --> 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 --> P n) &

```
   (0<k --> (∀ i j. 0≤j & j<k & n = k*i + j --> P j)) &
   (k<0 --> (∀ i j. k<j & j≤0 & n = k*i + j --> 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]
```

29.16 Speeding up the Division Algorithm with Shifting

computing div by shifting

```
lemma pos-zdiv-mult-2: (0::int) ≤ a ==> (1 + 2*b) div (2*a) = b div a
proof cases
  assume a=0
    thus ?thesis by simp
next
  assume a≠0 and le-a: 0≤a
  hence a-pos: 1 ≤ a by arith
  hence one-less-a2: 1 < 2*a by arith
  hence le-2a: 2 * (1 + b mod a) ≤ 2 * a
    by (simp add: mult-le-cancel-left add-commute [of 1] add1-zle-eq)
  with a-pos have 0 ≤ b mod a by simp
  hence le-addm: 0 ≤ 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 *not-0-le-lemma*: ~ *0* ≤ *x* ==> *x* ≤ (*0*::*int*)
by *auto*

lemma *zdiv-number-of-BIT*[*simp*]:
    number-of (*v BIT b*) *div number-of* (*w BIT bit.B0*) =
        (*if b*=*bit.B0* | (*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 Bin-simps UNIV-I split*: *split-if*)
apply (*simp add*: *zdiv-zmult-zmult1 pos-zdiv-mult-2 neg-zdiv-mult-2 add-ac*
        split: *bit.split*)
done

29.17 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 zmod-zadd1-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 left-distrib*)
apply (*subgoal-tac 0* ≤ *b mod a*, *arith*, *simp*)
done

lemma *neg-zmod-mult-2*:
    a ≤ (*0*::*int*) ==> (*1* + *2***b*) *mod* (*2***a*) = *2* * ((*b*+*1*) *mod a*) − *1*
apply (*subgoal-tac* (*1* + *2** (−*b* − *1*)) *mod* (*2** (−*a*)) =
                1 + *2** ((−*b* − *1*) *mod* (−*a*)))
apply (*rule-tac* [*2*] *pos-zmod-mult-2*)
apply (*auto simp add*: *minus-mult-right* [*symmetric*] *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-BIT* [*simp*]:
    number-of (*v BIT b*) *mod number-of* (*w BIT bit.B0*) =
     (*case b of*
        bit.B0 => *2* * (*number-of v mod number-of w*)
      | *bit.B1* => *if* (*0*::*int*) ≤ *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 Bin-simps UNIV-I split*: *bit.split*)
apply (*simp add*: *zmod-zmult-zmult1 pos-zmod-mult-2*
not-0-le-lemma neg-zmod-mult-2 add-ac)
done

29.18 Quotients of Signs

lemma *div-neg-pos-less0*: [| *a* < (*0*::*int*); *0* < *b* |] ==> *a div b* < *0*
apply (*subgoal-tac a div b* ≤ *-1*, *force*)
apply (*rule order-trans*)
apply (*rule-tac a'* = *-1* **in** *zdiv-mono1*)
apply (*auto simp add*: *zdiv-minus1*)
done

lemma *div-nonneg-neg-le0*: [| (*0*::*int*) ≤ *a*; *b* < *0* |] ==> *a div b* ≤ *0*
by (*drule zdiv-mono1-neg*, *auto*)

lemma *pos-imp-zdiv-nonneg-iff*: (*0*::*int*) < *b* ==> (*0* ≤ *a div b*) = (*0* ≤ *a*)
apply *auto*
apply (*drule-tac* [*2*] *zdiv-mono1*)
apply (*auto simp add*: *linorder-neg-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*) ==> (*0* ≤ *a div b*) = (*a* ≤ (*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* ==> (*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*) ==> (*a div b* < *0*) = (*0* < *a*)
by (*simp add*: *linorder-not-le* [*symmetric*] *neg-imp-zdiv-nonneg-iff*)

29.19 The Divides Relation

lemma *zdvd-iff-zmod-eq-0*: (*m dvd n*) = (*n mod m* = (*0*::*int*))
by(*simp add*:*dvd-def zmod-eq-0-iff*)

lemma *zdvd-0-right* [*iff*]: (*m*::*int*) *dvd 0*
by (*simp add*: *dvd-def*)

lemma *zdvd-0-left* [*iff*]: (*0 dvd* (*m*::*int*)) = (*m* = *0*)

by (*simp add*: *dvd-def*)

lemma *zdvd-1-left* [*iff*]: *1 dvd* (*m*::*int*)
by (*simp add*: *dvd-def*)

lemma *zdvd-refl* [*simp*]: *m dvd* (*m*::*int*)
by (*auto simp add*: *dvd-def intro*: *zmult-1-right* [*symmetric*])

lemma *zdvd-trans*: *m dvd n* ==> *n dvd k* ==> *m dvd* (*k*::*int*)
by (*auto simp add*: *dvd-def intro*: *mult-assoc*)

lemma *zdvd-zminus-iff*: (*m dvd* −*n*) = (*m dvd* (*n*::*int*))
apply (*simp add*: *dvd-def*, *auto*)
apply (*rule-tac* [!] *x* = −*k* **in** *exI*, *auto*)
done

lemma *zdvd-zminus2-iff*: (−*m dvd n*) = (*m dvd* (*n*::*int*))
apply (*simp add*: *dvd-def*, *auto*)
apply (*rule-tac* [!] *x* = −*k* **in** *exI*, *auto*)
done

lemma *zdvd-anti-sym*:
0 < *m* ==> *0* < *n* ==> *m dvd n* ==> *n dvd m* ==> *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-zadd*: *k dvd m* ==> *k dvd n* ==> *k dvd* (*m* + *n* :: *int*)
apply (*simp add*: *dvd-def*)
apply (*blast intro*: *right-distrib* [*symmetric*])
done

lemma *zdvd-zdiff*: *k dvd m* ==> *k dvd n* ==> *k dvd* (*m* − *n* :: *int*)
apply (*simp add*: *dvd-def*)
apply (*blast intro*: *right-diff-distrib* [*symmetric*])
done

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*: *zdvd-zadd*, *simp*)
done

lemma *zdvd-zmult*: *k dvd* (*n*::*int*) ==> *k dvd m* ∗ *n*
apply (*simp add*: *dvd-def*)
apply (*blast intro*: *mult-left-commute*)
done

lemma *zdvd-zmult2*: *k dvd* (*m*::*int*) ==> *k dvd m* ∗ *n*

apply (*subst mult-commute*)
apply (*erule zdvd-zmult*)
done

lemma *zdvd-triv-right* [*iff*]: (*k*::*int*) *dvd m* ∗ *k*
apply (*rule zdvd-zmult*)
apply (*rule zdvd-refl*)
done

lemma *zdvd-triv-left* [*iff*]: (*k*::*int*) *dvd k* ∗ *m*
apply (*rule zdvd-zmult2*)
apply (*rule zdvd-refl*)
done

lemma *zdvd-zmultD2*: *j* ∗ *k dvd n* ==> *j dvd* (*n*::*int*)
apply (*simp add*: *dvd-def*)
apply (*simp add*: *mult-assoc*, *blast*)
done

lemma *zdvd-zmultD*: *j* ∗ *k dvd n* ==> *k dvd* (*n*::*int*)
apply (*rule zdvd-zmultD2*)
apply (*subst mult-commute*, *assumption*)
done

lemma *zdvd-zmult-mono*: *i dvd m* ==> *j dvd* (*n*::*int*) ==> *i* ∗ *j dvd m* ∗ *n*
apply (*simp add*: *dvd-def*, *clarify*)
apply (*rule-tac x* = *k* ∗ *ka* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma *zdvd-reduce*: (*k dvd n* + *k* ∗ *m*) = (*k dvd* (*n*::*int*))
apply (*rule iffI*)
apply (*erule-tac* [2] *zdvd-zadd*)
apply (*subgoal-tac n* = (*n* + *k* ∗ *m*) − *k* ∗ *m*)
apply (*erule ssubst*)
apply (*erule zdvd-zdiff*, *simp-all*)
done

lemma *zdvd-zmod*: *f dvd m* ==> *f dvd* (*n*::*int*) ==> *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* ==> *k dvd n* ==> *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*: *zdvd-zadd zdvd-zmult2*)
done

lemma *zdvd-not-zless*: $0 < m \Longrightarrow m < n \Longrightarrow \neg\ n\ dvd\ (m::int)$
apply (*simp add*: *dvd-def*, *auto*)
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 *int-dvd-iff*: $(int\ m\ dvd\ z) = (m\ dvd\ nat\ (abs\ z))$
apply (*auto simp add*: *dvd-def nat-abs-mult-distrib*)
apply (*auto simp add*: *nat-eq-iff abs-if split add*: *split-if-asm*)
apply (*rule-tac* $x = -(int\ k)$ **in** *exI*)
apply (*auto simp add*: *int-mult*)
done

lemma *dvd-int-iff*: $(z\ dvd\ int\ m) = (nat\ (abs\ z)\ dvd\ m)$
apply (*auto simp add*: *dvd-def abs-if int-mult*)
apply (*rule-tac* [*3*] $x = nat\ k$ **in** *exI*)
apply (*rule-tac* [*2*] $x = -(int\ k)$ **in** *exI*)
apply (*rule-tac* $x = nat\ (-k)$ **in** *exI*)
apply (*cut-tac* [*3*] $k = m$ **in** *int-less-0-conv*)
apply (*cut-tac* $k = m$ **in** *int-less-0-conv*)
apply (*auto simp add*: *zero-le-mult-iff mult-less-0-iff*
nat-mult-distrib [*symmetric*] *nat-eq-iff2*)
done

lemma *nat-dvd-iff*: $(nat\ z\ dvd\ m) = (if\ 0 \le z\ then\ (z\ dvd\ int\ m)\ else\ m = 0)$
apply (*auto simp add*: *dvd-def int-mult*)
apply (*rule-tac* $x = nat\ k$ **in** *exI*)
apply (*cut-tac* $k = m$ **in** *int-less-0-conv*)
apply (*auto simp add*: *zero-le-mult-iff mult-less-0-iff*
nat-mult-distrib [*symmetric*] *nat-eq-iff2*)
done

lemma *zminus-dvd-iff* [*iff*]: $(-z\ dvd\ w) = (z\ dvd\ (w::int))$
apply (*auto simp add*: *dvd-def*)
apply (*rule-tac* [!] $x = -k$ **in** *exI*, *auto*)
done

lemma *dvd-zminus-iff* [*iff*]: $(z\ dvd\ -w) = (z\ dvd\ (w::int))$
apply (*auto simp add*: *dvd-def*)
apply (*drule minus-equation-iff* [*THEN iffD1*])
apply (*rule-tac* [!] $x = -k$ **in** *exI*, *auto*)
done

lemma *zdvd-imp-le*: $[|\ z\ dvd\ n;\ 0 < n\ |] \Longrightarrow z \le (n::int)$
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

29.20 Integer Powers

instance *int* :: *power* **..**

primrec
p ^ *0* = *1*
p ^ (*Suc n*) = (*p*::*int*) * (*p* ^ *n*)

instance *int* :: *recpower*
proof
fix *z* :: *int*
fix *n* :: *nat*
show *z*^*0* = *1* **by** *simp*
show *z*^(*Suc n*) = *z* * (*z*^*n*) **by** *simp*
qed

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 zmod-zmult-distrib* [*symmetric*])
done

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*)
apply (*induct n*)
apply (*auto simp add*: *zero-less-mult-iff*)
done

lemma *zero-le-zpower-abs* [*simp*]: (*0*::*int*) <= (*abs x*) ^*n*
apply (*induct n*)
apply (*auto simp add*: *zero-le-mult-iff*)
done

lemma *int-power*: *int* (*m*^*n*) = (*int m*) ^ *n*
by (*induct n*, *simp-all add*: *int-mult*)

Compatibility binding

lemmas *zpower-int* = *int-power* [*symmetric*]

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.quorem-def int-eq-of-nat*)
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.quorem-def int-eq-of-nat*)
done

Suggested by Matthias Daum

lemma *int-power-div-base*:
⟦0 < *m*; 0 < *k*⟧ ⟹ *k* ^ *m div k* = (*k*::*int*) ^ (*m* − *Suc* 0)
apply (*subgoal-tac k* ^ *m* = *k* ^ ((*m* − 1) + 1))
apply (*erule ssubst*)
apply (*simp only*: *power-add*)
apply *simp-all*
done

ML
⟪
val quorem-def = *thm quorem-def*;

val unique-quotient = *thm unique-quotient*;
val unique-remainder = *thm unique-remainder*;
val adjust-eq = *thm adjust-eq*;
val posDivAlg-eqn = *thm posDivAlg-eqn*;
val posDivAlg-correct = *thm posDivAlg-correct*;
val negDivAlg-eqn = *thm negDivAlg-eqn*;
val negDivAlg-correct = *thm negDivAlg-correct*;
val quorem-0 = *thm quorem-0*;
val posDivAlg-0 = *thm posDivAlg-0*;
val negDivAlg-minus1 = *thm negDivAlg-minus1*;
val negateSnd-eq = *thm negateSnd-eq*;
val quorem-neg = *thm quorem-neg*;
val divAlg-correct = *thm divAlg-correct*;
val DIVISION-BY-ZERO = *thm DIVISION-BY-ZERO*;
val zmod-zdiv-equality = *thm zmod-zdiv-equality*;
val pos-mod-conj = *thm pos-mod-conj*;
val pos-mod-sign = *thm pos-mod-sign*;

val neg-mod-conj = thm neg-mod-conj;
val neg-mod-sign = thm neg-mod-sign;
val quorem-div-mod = thm quorem-div-mod;
val quorem-div = thm quorem-div;
val quorem-mod = thm quorem-mod;
val div-pos-pos-trivial = thm div-pos-pos-trivial;
val div-neg-neg-trivial = thm div-neg-neg-trivial;
val div-pos-neg-trivial = thm div-pos-neg-trivial;
val mod-pos-pos-trivial = thm mod-pos-pos-trivial;
val mod-neg-neg-trivial = thm mod-neg-neg-trivial;
val mod-pos-neg-trivial = thm mod-pos-neg-trivial;
val zdiv-zminus-zminus = thm zdiv-zminus-zminus;
val zmod-zminus-zminus = thm zmod-zminus-zminus;
val zdiv-zminus1-eq-if = thm zdiv-zminus1-eq-if;
val zmod-zminus1-eq-if = thm zmod-zminus1-eq-if;
val zdiv-zminus2 = thm zdiv-zminus2;
val zmod-zminus2 = thm zmod-zminus2;
val zdiv-zminus2-eq-if = thm zdiv-zminus2-eq-if;
val zmod-zminus2-eq-if = thm zmod-zminus2-eq-if;
val self-quotient = thm self-quotient;
val self-remainder = thm self-remainder;
val zdiv-self = thm zdiv-self;
val zmod-self = thm zmod-self;
val zdiv-zero = thm zdiv-zero;
val div-eq-minus1 = thm div-eq-minus1;
val zmod-zero = thm zmod-zero;
val zdiv-minus1 = thm zdiv-minus1;
val zmod-minus1 = thm zmod-minus1;
val div-pos-pos = thm div-pos-pos;
val mod-pos-pos = thm mod-pos-pos;
val div-neg-pos = thm div-neg-pos;
val mod-neg-pos = thm mod-neg-pos;
val div-pos-neg = thm div-pos-neg;
val mod-pos-neg = thm mod-pos-neg;
val div-neg-neg = thm div-neg-neg;
val mod-neg-neg = thm mod-neg-neg;
val zmod-1 = thm zmod-1;
val zdiv-1 = thm zdiv-1;
val zmod-minus1-right = thm zmod-minus1-right;
val zdiv-minus1-right = thm zdiv-minus1-right;
val zdiv-mono1 = thm zdiv-mono1;
val zdiv-mono1-neg = thm zdiv-mono1-neg;
val zdiv-mono2 = thm zdiv-mono2;
val zdiv-mono2-neg = thm zdiv-mono2-neg;
val zdiv-zmult1-eq = thm zdiv-zmult1-eq;
val zmod-zmult1-eq = thm zmod-zmult1-eq;
val zmod-zmult1-eq′ = thm zmod-zmult1-eq′;
val zmod-zmult-distrib = thm zmod-zmult-distrib;
val zdiv-zmult-self1 = thm zdiv-zmult-self1;

val zdiv-zmult-self2 = thm zdiv-zmult-self2;
val zmod-zmult-self1 = thm zmod-zmult-self1;
val zmod-zmult-self2 = thm zmod-zmult-self2;
val zmod-eq-0-iff = thm zmod-eq-0-iff;
val zdiv-zadd1-eq = thm zdiv-zadd1-eq;
val zmod-zadd1-eq = thm zmod-zadd1-eq;
val mod-div-trivial = thm mod-div-trivial;
val mod-mod-trivial = thm mod-mod-trivial;
val zmod-zadd-left-eq = thm zmod-zadd-left-eq;
val zmod-zadd-right-eq = thm zmod-zadd-right-eq;
val zdiv-zadd-self1 = thm zdiv-zadd-self1;
val zdiv-zadd-self2 = thm zdiv-zadd-self2;
val zmod-zadd-self1 = thm zmod-zadd-self1;
val zmod-zadd-self2 = thm zmod-zadd-self2;
val zdiv-zmult2-eq = thm zdiv-zmult2-eq;
val zmod-zmult2-eq = thm zmod-zmult2-eq;
val zdiv-zmult-zmult1 = thm zdiv-zmult-zmult1;
val zdiv-zmult-zmult2 = thm zdiv-zmult-zmult2;
val zmod-zmult-zmult1 = thm zmod-zmult-zmult1;
val zmod-zmult-zmult2 = thm zmod-zmult-zmult2;
val pos-zdiv-mult-2 = thm pos-zdiv-mult-2;
val neg-zdiv-mult-2 = thm neg-zdiv-mult-2;
val zdiv-number-of-BIT = thm zdiv-number-of-BIT;
val pos-zmod-mult-2 = thm pos-zmod-mult-2;
val neg-zmod-mult-2 = thm neg-zmod-mult-2;
val zmod-number-of-BIT = thm zmod-number-of-BIT;
val div-neg-pos-less0 = thm div-neg-pos-less0;
val div-nonneg-neg-le0 = thm div-nonneg-neg-le0;
val pos-imp-zdiv-nonneg-iff = thm pos-imp-zdiv-nonneg-iff;
val neg-imp-zdiv-nonneg-iff = thm neg-imp-zdiv-nonneg-iff;
val pos-imp-zdiv-neg-iff = thm pos-imp-zdiv-neg-iff;
val neg-imp-zdiv-neg-iff = thm neg-imp-zdiv-neg-iff;

val zpower-zmod = thm zpower-zmod;
val zpower-zadd-distrib = thm zpower-zadd-distrib;
val zpower-zpower = thm zpower-zpower;
val zero-less-zpower-abs-iff = thm zero-less-zpower-abs-iff;
val zero-le-zpower-abs = thm zero-le-zpower-abs;
val zpower-int = thm zpower-int;
⟩⟩

end

30 NatBin: Binary arithmetic for the natural numbers

theory *NatBin*

imports *IntDiv*
begin

Arithmetic for naturals is reduced to that for the non-negative integers.

instance *nat* :: *number* **..**

defs (**overloaded**)
nat-number-of-def:
(*number-of*::*bin* => *nat*) *v* == *nat* ((*number-of* :: *bin* => *int*) *v*)

30.1 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

Distributive laws for type *nat*. The others are in theory *IntArith*, but these require div and mod to be defined for type "int". They also need some of the lemmas proved above.

lemma *nat-div-distrib*: (*0*::*int*) <= *z* ==> *nat* (*z div z'*) = *nat z div nat z'*
apply (*case-tac 0* <= *z'*)
apply (*auto simp add*: *div-nonneg-neg-le0 DIVISION-BY-ZERO-DIV*)
apply (*case-tac z'* = *0*, *simp add*: *DIVISION-BY-ZERO*)
apply (*auto elim*!: *nonneg-eq-int*)
apply (*rename-tac m m'*)
apply (*subgoal-tac 0* <= *int m div int m'*)
prefer *2* **apply** (*simp add*: *nat-numeral-0-eq-0 pos-imp-zdiv-nonneg-iff*)
apply (*rule inj-int* [*THEN injD*], *simp*)
apply (*rule-tac r* = *int* (*m mod m'*) **in** *quorem-div*)
prefer *2* **apply** *force*
apply (*simp add*: *nat-less-iff* [*symmetric*] *quorem-def nat-numeral-0-eq-0 zadd-int*

zmult-int)
done

lemma *nat-mod-distrib*:
[| (0::*int*) <= *z*; 0 <= *z'* |] ==> *nat* (*z mod z'*) = *nat z mod nat z'*
apply (*case-tac z' = 0*, *simp add*: *DIVISION-BY-ZERO*)
apply (*auto elim*!: *nonneg-eq-int*)
apply (*rename-tac m m'*)
apply (*subgoal-tac 0 <= int m mod int m'*)
prefer *2* **apply** (*simp add*: *nat-less-iff nat-numeral-0-eq-0 pos-mod-sign*)
apply (*rule inj-int* [*THEN injD*], *simp*)
apply (*rule-tac q = int* (*m div m'*) **in** *quorem-mod*)
prefer *2* **apply** *force*
apply (*simp add*: *nat-less-iff* [*symmetric*] *quorem-def nat-numeral-0-eq-0 zadd-int zmult-int*)
done

Suggested by Matthias Daum

lemma *int-div-less-self*: ⟦*0 < x*; *1 < k*⟧ ⟹ *x div k* < (*x*::*int*)
apply (*subgoal-tac nat x div nat k < nat x*)
apply (*simp add*: *nat-div-distrib* [*symmetric*])
apply (*rule Divides.div-less-dividend*, *simp-all*)
done

30.2 Function *int*: Coercion from Type *nat* to *int*

lemma *int-nat-number-of* [*simp*]:
int (*number-of v* :: *nat*) =
(*if neg* (*number-of v* :: *int*) *then 0*
else (*number-of v* :: *int*))
by (*simp del*: *nat-number-of*
add: *neg-nat nat-number-of-def not-neg-nat add-assoc*)

30.2.1 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* (*bin-succ v*) + *n*)
by (*simp del*: *nat-number-of*
add: *nat-number-of-def neg-nat*
Suc-nat-eq-nat-zadd1 number-of-succ)

lemma *Suc-nat-number-of* [*simp*]:
Suc (*number-of v*) =
(*if neg* (*number-of v* :: *int*) *then 1 else number-of* (*bin-succ v*))

apply (*cut-tac n = 0* **in** *Suc-nat-number-of-add*)
apply (*simp cong del*: *if-weak-cong*)
done

30.2.2 Addition

lemma *add-nat-number-of* [*simp*]:
(*number-of v* :: *nat*) + *number-of v'* =
(*if neg* (*number-of v* :: *int*) *then number-of v'*
else if neg (*number-of v'* :: *int*) *then number-of v*
else number-of (*bin-add v v'*))
by (*force dest*!: *neg-nat*
simp del: *nat-number-of*
simp add: *nat-number-of-def nat-add-distrib* [*symmetric*])

30.2.3 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)
apply (*simp add*: *Let-def nat-diff-distrib* [*symmetric*] *neg-eq-less-0 not-neg-eq-ge-0*)
apply (*simp add*: *diff-is-0-eq nat-le-eq-zle*)
done

lemma *diff-nat-number-of* [*simp*]:
(*number-of v* :: *nat*) − *number-of v'* =
(*if neg* (*number-of v'* :: *int*) *then number-of v*
else let d = *number-of* (*bin-add v* (*bin-minus v'*)) *in*
if neg d then 0 else nat d)
by (*simp del*: *nat-number-of add*: *diff-nat-eq-if nat-number-of-def*)

30.2.4 Multiplication

lemma *mult-nat-number-of* [*simp*]:
(*number-of v* :: *nat*) ∗ *number-of v'* =
(*if neg* (*number-of v* :: *int*) *then 0 else number-of* (*bin-mult v v'*))
by (*force dest*!: *neg-nat*
simp del: *nat-number-of*
simp add: *nat-number-of-def nat-mult-distrib* [*symmetric*])

30.2.5 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'*))
by (*force dest*!: *neg-nat*
simp del: *nat-number-of*

simp add: *nat-number-of-def nat-div-distrib* [*symmetric*])

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*])

30.2.6 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'*))
by (*force dest*!: *neg-nat*
simp del: *nat-number-of*
simp add: *nat-number-of-def nat-mod-distrib* [*symmetric*])

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*])

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*;
⟫

30.3 Comparisons

30.3.1 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-of v* :: *nat*) = *number-of v'*) =
(*if neg* (*number-of v* :: *int*) *then* (*iszero* (*number-of v'* :: *int*) | *neg* (*number-of v'* :: *int*))
else if neg (*number-of v'* :: *int*) *then iszero* (*number-of v* :: *int*)
else iszero (*number-of* (*bin-add v* (*bin-minus v'*)) :: *int*))
apply (*simp only*: *simp-thms neg-nat not-neg-eq-ge-0 nat-number-of-def*
eq-nat-nat-iff eq-number-of-eq nat-0 iszero-def
split add: *split-if cong add*: *imp-cong*)
apply (*simp only*: *nat-eq-iff nat-eq-iff2*)
apply (*simp add*: *not-neg-eq-ge-0* [*symmetric*])
done

30.3.2 Less-than (¡)

lemma *less-nat-number-of* [*simp*]:
((*number-of v* :: *nat*) < *number-of v'*) =
(*if neg* (*number-of v* :: *int*) *then neg* (*number-of* (*bin-minus v'*) :: *int*)
else neg (*number-of* (*bin-add v* (*bin-minus v'*)) :: *int*))
by (*simp only*: *simp-thms neg-nat not-neg-eq-ge-0 nat-number-of-def*
nat-less-eq-zless less-number-of-eq-neg zless-nat-eq-int-zless
cong add: *imp-cong*, *simp*)

lemmas *numerals* = *nat-numeral-0-eq-0 nat-numeral-1-eq-1 numeral-2-eq-2*

30.4 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*::{*comm-semiring-1-cancel*,*recpower*})^2 = *a* * *a*
by (*simp add*: *numeral-2-eq-2 Power.power-Suc*)

lemma *zero-power2* [*simp*]: (*0*::*'a*::{*comm-semiring-1-cancel*,*recpower*})^2 = *0*
by (*simp add*: *power2-eq-square*)

lemma *one-power2* [*simp*]: $(1::'a::\{comm\text{-}semiring\text{-}1\text{-}cancel,recpower\})^2 = 1$
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*: $0 \leq (a^2::'a::\{ordered\text{-}idom,recpower\})$
by (*simp add*: *power2-eq-square zero-le-square*)

lemma *zero-less-power2*:
$(0 < a^2) = (a \neq (0::'a::\{ordered\text{-}idom,recpower\}))$
by (*force simp add*: *power2-eq-square zero-less-mult-iff linorder-neq-iff*)

lemma *power2-less-0*:
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*:
$(a^2 = 0) = (a = (0::'a::\{ordered\text{-}idom,recpower\}))$
by (*force simp add*: *power2-eq-square mult-eq-0-iff*)

lemma *abs-power2*:
$abs(a^2) = (a^2::'a::\{ordered\text{-}idom,recpower\})$
by (*simp add*: *power2-eq-square abs-mult abs-mult-self*)

lemma *power2-abs*:
$(abs\ a)^2 = (a^2::'a::\{ordered\text{-}idom,recpower\})$
by (*simp add*: *power2-eq-square abs-mult-self*)

lemma *power2-minus*:
$(-\ a)^2 = (a^2::'a::\{comm\text{-}ring\text{-}1,recpower\})$
by (*simp add*: *power2-eq-square*)

lemma *power-minus1-even*: $(-\ 1)\ \hat{}\ (2*n) = (1::'a::\{comm\text{-}ring\text{-}1,recpower\})$
apply (*induct n*)
apply (*auto simp add*: *power-Suc power-add power2-minus*)

done

lemma *power-even-eq*: $(a::'a::recpower)\ \hat{}\ (2*n) = (a\hat{}n)\hat{}2$
by (*simp add*: *power-mult power-mult-distrib power2-eq-square*)

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'*:
$0 \leq (a::'a::\{ordered\text{-}idom,recpower\})\ \hat{}\ (2*n)$
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-square 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*
show *?case* **by** (*simp add*: *Power.power-Suc*)
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 Power.power-Suc*)
thus *?case*
by (*simp 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

30.4.1 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*]

30.4.2 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*: $(m::nat) + n = (if\ m=0\ then\ n\ else\ Suc\ ((m - 1) + n))$
apply (*case-tac m*)
apply (*simp-all add*: *numerals*)
done

lemma *mult-eq-if*: $(m::nat) * n = (if\ m=0\ then\ 0\ else\ n + ((m - 1) * n))$
apply (*case-tac m*)
apply (*simp-all add*: *numerals*)
done

lemma *power-eq-if*: $(p\ \hat{}\ m :: nat) = (if\ m=0\ then\ 1\ else\ p * (p\ \hat{}\ (m - 1)))$
apply (*case-tac m*)
apply (*simp-all add*: *numerals*)
done

30.5 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*]:
$(number\text{-}of\ v = (0::nat)) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ True\ else\ iszero\ (number\text{-}of\ v :: int))$
by (*simp del*: *nat-numeral-0-eq-0 add*: *nat-numeral-0-eq-0* [*symmetric*] *iszero-0*)

lemma *eq-0-number-of* [*simp*]:
$((0::nat) = number\text{-}of\ v) =$
$(if\ neg\ (number\text{-}of\ v :: int)\ then\ True\ else\ iszero\ (number\text{-}of\ v :: int))$
by (*rule trans* [*OF eq-sym-conv eq-number-of-0*])

lemma *less-0-number-of* [*simp*]:
((0::*nat*) < *number-of v*) = *neg* (*number-of* (*bin-minus v*) :: *int*)
by (*simp del*: *nat-numeral-0-eq-0 add*: *nat-numeral-0-eq-0* [*symmetric*])

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*] *iszero-0*)

30.6 Comparisons involving Suc

lemma *eq-number-of-Suc* [*simp*]:
(*number-of v* = *Suc n*) =
(*let pv* = *number-of* (*bin-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* (*bin-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* (*bin-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* (*bin-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* (*bin-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* (*bin-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*])

declare *zadd-int* [*symmetric*, *simp*]

lemma *lemma1*: (*m*+*m* = *n*+*n*) = (*m* = (*n*::*int*))
by *auto*

lemma *lemma2*: *m*+*m* ~= (*1*::*int*) + (*n* + *n*)
apply *auto*
apply (*drule-tac f* = %*x*. *x mod 2* **in** *arg-cong*)
apply (*simp add*: *zmod-zadd1-eq*)
done

lemma *eq-number-of-BIT-BIT*:
((*number-of* (*v BIT x*) ::*int*) = *number-of* (*w BIT y*)) =
(*x*=*y* & (((*number-of v*) ::*int*) = *number-of w*))
apply (*simp only*: *number-of-BIT lemma1 lemma2 eq-commute*
OrderedGroup.add-left-cancel add-assoc OrderedGroup.add-0
split add: *bit.split*)
apply *simp*
done

lemma *eq-number-of-BIT-Pls*:
((*number-of* (*v BIT x*) ::*int*) = *Numeral0*) =
(*x*=*bit.B0* & (((*number-of v*) ::*int*) = *Numeral0*))
apply (*simp only*: *simp-thms add*: *number-of-BIT number-of-Pls eq-commute*
split add: *bit.split cong*: *imp-cong*)
apply (*rule-tac x* = *number-of v* **in** *spec*, *safe*)
apply (*simp-all* (*no-asm-use*))
apply (*drule-tac f* = %*x*. *x mod 2* **in** *arg-cong*)
apply (*simp add*: *zmod-zadd1-eq*)
done

lemma *eq-number-of-BIT-Min*:
((*number-of* (*v BIT x*) ::*int*) = *number-of Numeral.Min*) =

(*x*=*bit.B1* & (((*number-of v*) ::*int*) = *number-of Numeral.Min*))
apply (*simp only*: *simp-thms add*: *number-of-BIT number-of-Min eq-commute*
split add: *bit.split cong*: *imp-cong*)
apply (*rule-tac x* = *number-of v* **in** *spec*, *auto*)
apply (*drule-tac f* = %*x*. *x mod 2* **in** *arg-cong*, *auto*)
done

lemma *eq-number-of-Pls-Min*: (*Numeral0* ::*int*) ~= *number-of Numeral.Min*
by *auto*

30.7 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 the integers

lemma *zpower-number-of-even*:
(*z*::*int*) ^ *number-of* (*w BIT bit.B0*) =
(*let w* = *z* ^ (*number-of w*) *in* *w**w*)
apply (*simp del*: *nat-number-of add*: *nat-number-of-def number-of-BIT Let-def*)
apply (*simp only*: *number-of-add*)
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 *zpower-number-of-odd*:
(*z*::*int*) ^ *number-of* (*w BIT bit.B1*) =
(*if* (*0*::*int*) <= *number-of w*
then (*let w* = *z* ^ (*number-of w*) *in* *z**w**w*)
else 1)
apply (*simp del*: *nat-number-of add*: *nat-number-of-def number-of-BIT Let-def*)
apply (*simp only*: *number-of-add nat-numeral-1-eq-1 not-neg-eq-ge-0 neg-eq-less-0*)

apply (*rule-tac x* = *number-of w* **in** *spec*, *clarify*)
apply (*auto simp add*: *nat-add-distrib nat-mult-distrib power-even-eq power2-eq-square*

neg-nat)
done

lemmas *zpower-number-of-even-number-of* =
zpower-number-of-even [*of number-of v*, *standard*]
declare *zpower-number-of-even-number-of* [*simp*]

lemmas *zpower-number-of-odd-number-of* =
zpower-number-of-odd [*of number-of v*, *standard*]
declare *zpower-number-of-odd-number-of* [*simp*]

ML
⟪
val numerals = *thmsnumerals*;
val numeral-ss = *simpset*() *addsimps numerals*;

val nat-bin-arith-setup =
[*Fast-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* [*Suc-nat-number-of*, *int-nat-number-of*,
not-neg-number-of-Pls,
neg-number-of-Min,*neg-number-of-BIT*]})]
⟫

setup *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 Numeral.Min* = (*0*::*nat*)
apply (*simp only*: *number-of-Min nat-number-of-def nat-zminus-int*)
apply (*simp add*: *neg-nat*)
done

lemma *nat-number-of-BIT-1*:
number-of (*w BIT bit.B1*) =
(*if neg* (*number-of w* :: *int*) *then 0*
else let n = *number-of w in Suc* (*n* + *n*))
apply (*simp only*: *nat-number-of-def Let-def split*: *split-if*)

apply (*intro conjI impI*)
apply (*simp add*: *neg-nat neg-number-of-BIT*)
apply (*rule int-int-eq* [*THEN iffD1*])
apply (*simp only*: *not-neg-nat neg-number-of-BIT int-Suc zadd-int* [*symmetric*] *simp-thms*)
apply (*simp only*: *number-of-BIT zadd-assoc split*: *bit.split*)
apply *simp*
done

lemma *nat-number-of-BIT-0*:
number-of (*w BIT bit.B0*) = (*let n*::*nat* = *number-of w in n* + *n*)
apply (*simp only*: *nat-number-of-def Let-def*)
apply (*cases neg* (*number-of w* :: *int*))
apply (*simp add*: *neg-nat neg-number-of-BIT*)
apply (*rule int-int-eq* [*THEN iffD1*])
apply (*simp only*: *not-neg-nat neg-number-of-BIT int-Suc zadd-int* [*symmetric*] *simp-thms*)
apply (*simp only*: *number-of-BIT zadd-assoc*)
apply *simp*
done

lemmas *nat-number* =
nat-number-of-Pls nat-number-of-Min
nat-number-of-BIT-1 nat-number-of-BIT-0

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*)

lemma *power-m1-odd*: (−1) ^ *Suc*(2∗*n*) = (−1::′*a*::{*number-ring*,*recpower*})
by (*simp add*: *power-mult power-Suc*)

30.8 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*, *simp*)

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*)

30.9 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 (*bin-add v v'*) + *k*)
by *simp*

lemma *nat-number-of-mult-left*:
number-of v ∗ (*number-of v'* ∗ (*k*::*nat*)) =
(*if neg* (*number-of v* :: *int*) *then 0*
else number-of (*bin-mult v v'*) ∗ *k*)
by *simp*

30.9.1 For *combine-numerals*

lemma *left-add-mult-distrib*: *i*∗*u* + (*j*∗*u* + *k*) = (*i*+*j*)∗*u* + (*k*::*nat*)
by (*simp add*: *add-mult-distrib*)

30.9.2 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*)

30.9.3 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*

30.9.4 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*:
$(k*m)\ div\ (k*n) = (if\ k = (0::nat)\ then\ 0\ else\ m\ div\ n)$
by (*simp add*: *nat-mult-div-cancel1*)

ML
⟪
val eq-nat-nat-iff = thmeq-nat-nat-iff;
val eq-nat-number-of = thmeq-nat-number-of;
val less-nat-number-of = thmless-nat-number-of;
val power2-eq-square = thm power2-eq-square;
val zero-le-power2 = thm zero-le-power2;
val zero-less-power2 = thm zero-less-power2;

```
val zero-eq-power2 = thm zero-eq-power2;
val abs-power2 = thm abs-power2;
val power2-abs = thm power2-abs;
val power2-minus = thm power2-minus;
val power-minus1-even = thm power-minus1-even;
val power-minus-even = thm power-minus-even;
(* val zero-le-even-power = thm zero-le-even-power; *)
val odd-power-less-zero = thm odd-power-less-zero;
val odd-0-le-power-imp-0-le = thm odd-0-le-power-imp-0-le;

val Suc-pred' = thmSuc-pred';
val expand-Suc = thmexpand-Suc;
val Suc-eq-add-numeral-1 = thmSuc-eq-add-numeral-1;
val Suc-eq-add-numeral-1-left = thmSuc-eq-add-numeral-1-left;
val add-eq-if = thmadd-eq-if;
val mult-eq-if = thmmult-eq-if;
val power-eq-if = thmpower-eq-if;
val eq-number-of-0 = thmeq-number-of-0;
val eq-0-number-of = thmeq-0-number-of;
val less-0-number-of = thmless-0-number-of;
val neg-imp-number-of-eq-0 = thmneg-imp-number-of-eq-0;
val eq-number-of-Suc = thmeq-number-of-Suc;
val Suc-eq-number-of = thmSuc-eq-number-of;
val less-number-of-Suc = thmless-number-of-Suc;
val less-Suc-number-of = thmless-Suc-number-of;
val le-number-of-Suc = thmle-number-of-Suc;
val le-Suc-number-of = thmle-Suc-number-of;
val eq-number-of-BIT-BIT = thmeq-number-of-BIT-BIT;
val eq-number-of-BIT-Pls = thmeq-number-of-BIT-Pls;
val eq-number-of-BIT-Min = thmeq-number-of-BIT-Min;
val eq-number-of-Pls-Min = thmeq-number-of-Pls-Min;
val of-nat-number-of-eq = thmof-nat-number-of-eq;
val nat-power-eq = thmnat-power-eq;
val power-nat-number-of = thmpower-nat-number-of;
val zpower-number-of-even = thmzpower-number-of-even;
val zpower-number-of-odd = thmzpower-number-of-odd;
val nat-number-of-Pls = thmnat-number-of-Pls;
val nat-number-of-Min = thmnat-number-of-Min;
val Let-Suc = thmLet-Suc;

val nat-number = thmsnat-number;

val nat-number-of-add-left = thmnat-number-of-add-left;
val nat-number-of-mult-left = thmnat-number-of-mult-left;
val left-add-mult-distrib = thmleft-add-mult-distrib;
val nat-diff-add-eq1 = thmnat-diff-add-eq1;
val nat-diff-add-eq2 = thmnat-diff-add-eq2;
val nat-eq-add-iff1 = thmnat-eq-add-iff1;
val nat-eq-add-iff2 = thmnat-eq-add-iff2;
```

val nat-less-add-iff1 = thmnat-less-add-iff1;
val nat-less-add-iff2 = thmnat-less-add-iff2;
val nat-le-add-iff1 = thmnat-le-add-iff1;
val nat-le-add-iff2 = thmnat-le-add-iff2;
val nat-mult-le-cancel1 = thmnat-mult-le-cancel1;
val nat-mult-less-cancel1 = thmnat-mult-less-cancel1;
val nat-mult-eq-cancel1 = thmnat-mult-eq-cancel1;
val nat-mult-div-cancel1 = thmnat-mult-div-cancel1;
val nat-mult-le-cancel-disj = thmnat-mult-le-cancel-disj;
val nat-mult-less-cancel-disj = thmnat-mult-less-cancel-disj;
val nat-mult-eq-cancel-disj = thmnat-mult-eq-cancel-disj;
val nat-mult-div-cancel-disj = thmnat-mult-div-cancel-disj;

val power-minus-even = thmpower-minus-even;
(∗ *val zero-le-even-power = thmzero-le-even-power*; ∗)
⟩⟩

end

31 NatSimprocs: Simprocs for the Naturals

theory *NatSimprocs*
imports *NatBin*
uses *int-factor-simprocs.ML nat-simprocs.ML*
begin

setup *nat-simprocs-setup*

31.1 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-bin-pred-iff-0*:
    neg (number-of (bin-pred v)::int) = (number-of v = (0::nat))
apply (*subgoal-tac neg (number-of (bin-pred v)) = (number-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*:
    neg (number-of (bin-minus v)::int) ==>

```
    Suc m − (number-of v) = m − (number-of (bin-pred v))
apply (subst Suc-diff-eq-diff-pred)
apply (simp add: )
apply (simp del: nat-numeral-1-eq-1)
apply (auto simp only: diff-nat-number-of less-0-number-of [symmetric]
                       neg-number-of-bin-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)
```

31.2 For *nat-case* and *nat-rec*

```
lemma nat-case-number-of [simp]:
    nat-case a f (number-of v) =
      (let pv = number-of (bin-pred v) in
       if neg pv then a else f (nat pv))
by (simp split add: nat.split add: Let-def neg-number-of-bin-pred-iff-0)

lemma nat-case-add-eq-if [simp]:
    nat-case a f ((number-of v) + n) =
      (let pv = number-of (bin-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: numeral-1-eq-Suc-0 [symmetric] Let-def
               neg-imp-number-of-eq-0 neg-number-of-bin-pred-iff-0)
done

lemma nat-rec-number-of [simp]:
    nat-rec a f (number-of v) =
      (let pv = number-of (bin-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-bin-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 (bin-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: numeral-1-eq-Suc-0 [symmetric] Let-def neg-imp-number-of-eq-0
               neg-number-of-bin-pred-iff-0)
```

done

31.3 Various Other Lemmas

31.3.1 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*: $(n::nat) < 2 ==> n = 0 \mid n = Suc\ 0$
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

31.3.2 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*]

31.4 Special Simplification for Constants

These belong here, late in the development of HOL, to prevent their interfering with proofs of abstract properties of instances of the function *number-of*

These distributive laws move literals inside sums and differences.

lemmas *left-distrib-number-of* = *left-distrib* [*of* - - *number-of v*, *standard*]
declare *left-distrib-number-of* [*simp*]

lemmas *right-distrib-number-of* = *right-distrib* [*of number-of v*, *standard*]
declare *right-distrib-number-of* [*simp*]

lemmas *left-diff-distrib-number-of* =
left-diff-distrib [*of* - - *number-of v*, *standard*]
declare *left-diff-distrib-number-of* [*simp*]

lemmas *right-diff-distrib-number-of* =
right-diff-distrib [*of number-of v*, *standard*]
declare *right-diff-distrib-number-of* [*simp*]

These are actually for fields, like real: but where else to put them?

lemmas *zero-less-divide-iff-number-of* =

zero-less-divide-iff [*of number-of w*, *standard*]
declare *zero-less-divide-iff-number-of* [*simp*]

lemmas *divide-less-0-iff-number-of* =
divide-less-0-iff [*of number-of w*, *standard*]
declare *divide-less-0-iff-number-of* [*simp*]

lemmas *zero-le-divide-iff-number-of* =
zero-le-divide-iff [*of number-of w*, *standard*]
declare *zero-le-divide-iff-number-of* [*simp*]

lemmas *divide-le-0-iff-number-of* =
divide-le-0-iff [*of number-of w*, *standard*]
declare *divide-le-0-iff-number-of* [*simp*]

Replaces *inverse #nn* by *1/#nn*. It looks strange, but then other simprocs simplify the quotient.

lemmas *inverse-eq-divide-number-of* =
inverse-eq-divide [*of number-of w*, *standard*]
declare *inverse-eq-divide-number-of* [*simp*]

These laws simplify inequalities, moving unary minus from a term into the literal.

lemmas *less-minus-iff-number-of* =
less-minus-iff [*of number-of v*, *standard*]
declare *less-minus-iff-number-of* [*simp*]

lemmas *le-minus-iff-number-of* =
le-minus-iff [*of number-of v*, *standard*]
declare *le-minus-iff-number-of* [*simp*]

lemmas *equation-minus-iff-number-of* =
equation-minus-iff [*of number-of v*, *standard*]
declare *equation-minus-iff-number-of* [*simp*]

lemmas *minus-less-iff-number-of* =
minus-less-iff [*of - number-of v*, *standard*]
declare *minus-less-iff-number-of* [*simp*]

lemmas *minus-le-iff-number-of* =
minus-le-iff [*of - number-of v*, *standard*]
declare *minus-le-iff-number-of* [*simp*]

lemmas *minus-equation-iff-number-of* =
minus-equation-iff [*of - number-of v*, *standard*]
declare *minus-equation-iff-number-of* [*simp*]

These simplify inequalities where one side is the constant 1.

lemmas *less-minus-iff-1 = less-minus-iff [of 1, simplified]*
declare *less-minus-iff-1 [simp]*

lemmas *le-minus-iff-1 = le-minus-iff [of 1, simplified]*
declare *le-minus-iff-1 [simp]*

lemmas *equation-minus-iff-1 = equation-minus-iff [of 1, simplified]*
declare *equation-minus-iff-1 [simp]*

lemmas *minus-less-iff-1 = minus-less-iff [of - 1, simplified]*
declare *minus-less-iff-1 [simp]*

lemmas *minus-le-iff-1 = minus-le-iff [of - 1, simplified]*
declare *minus-le-iff-1 [simp]*

lemmas *minus-equation-iff-1 = minus-equation-iff [of - 1, simplified]*
declare *minus-equation-iff-1 [simp]*

Cancellation of constant factors in comparisons ($<$ and $\leq$)

lemmas *mult-less-cancel-left-number-of =*
mult-less-cancel-left [of number-of v, standard]
declare *mult-less-cancel-left-number-of [simp]*

lemmas *mult-less-cancel-right-number-of =*
mult-less-cancel-right [of - number-of v, standard]
declare *mult-less-cancel-right-number-of [simp]*

lemmas *mult-le-cancel-left-number-of =*
mult-le-cancel-left [of number-of v, standard]
declare *mult-le-cancel-left-number-of [simp]*

lemmas *mult-le-cancel-right-number-of =*
mult-le-cancel-right [of - number-of v, standard]
declare *mult-le-cancel-right-number-of [simp]*

Multiplying out constant divisors in comparisons ($<$, $\leq$ and $=$)

lemmas *le-divide-eq-number-of = le-divide-eq [of - - number-of w, standard]*
declare *le-divide-eq-number-of [simp]*

lemmas *divide-le-eq-number-of = divide-le-eq [of - number-of w, standard]*
declare *divide-le-eq-number-of [simp]*

lemmas *less-divide-eq-number-of = less-divide-eq [of - - number-of w, standard]*
declare *less-divide-eq-number-of [simp]*

lemmas *divide-less-eq-number-of = divide-less-eq [of - number-of w, standard]*
declare *divide-less-eq-number-of [simp]*

lemmas *eq-divide-eq-number-of = eq-divide-eq [of - - number-of w, standard]*

declare *eq-divide-eq-number-of* [*simp*]

lemmas *divide-eq-eq-number-of* = *divide-eq-eq* [*of - number-of w*, *standard*]
declare *divide-eq-eq-number-of* [*simp*]

31.5 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

31.5.1 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* = *half-gt-zero-iff* [*THEN iffD2*, *simp*]

lemma *nat-dvd-not-less*:
[| $0 < m$; $m < n$ |] ==> $\neg$ n *dvd* $(m::nat)$
by (*unfold dvd-def*) *auto*

ML
⟪
val divide-minus1 = *thm divide-minus1*;
val minus1-divide = *thm minus1-divide*;
⟫

end

32 Presburger: Presburger Arithmetic: Cooper's Algorithm

theory *Presburger*
imports *NatSimprocs SetInterval*
uses (*cooper-dec.ML*) (*cooper-proof.ML*) (*qelim.ML*)
(*reflected-presburger.ML*) (*reflected-cooper.ML*) (*presburger.ML*)
begin

Theorem for unitifying the coeffitients of x in an existential formula

theorem *unity-coeff-ex*: $(\exists x{::}int.\ P\ (l * x)) = (\exists x.\ l\ dvd\ (1{*}x{+}0) \wedge P\ x)$
apply (*rule iffI*)
apply (*erule exE*)
apply (*rule-tac* $x = l * x$ **in** *exI*)
apply *simp*
apply (*erule exE*)
apply (*erule conjE*)
apply (*erule dvdE*)
apply (*rule-tac* $x = k$ **in** *exI*)
apply *simp*
done

lemma *uminus-dvd-conv*: $(d\ dvd\ (t{::}int)) = (-d\ dvd\ t)$
apply(*unfold dvd-def*)
apply(*rule iffI*)
apply(*clarsimp*)
apply(*rename-tac k*)
apply(*rule-tac* $x = -k$ **in** *exI*)
apply *simp*
apply(*clarsimp*)
apply(*rename-tac k*)
apply(*rule-tac* $x = -k$ **in** *exI*)
apply *simp*
done

lemma *uminus-dvd-conv'*: $(d\ dvd\ (t{::}int)) = (d\ dvd\ -t)$
apply(*unfold dvd-def*)
apply(*rule iffI*)
apply(*clarsimp*)
apply(*rule-tac* $x = -k$ **in** *exI*)
apply *simp*
apply(*clarsimp*)
apply(*rule-tac* $x = -k$ **in** *exI*)
apply *simp*
done

Theorems for the combination of proofs of the equality of P and P-m for integers x less than some integer z.

theorem *eq-minf-conjI*: $\exists\, z1{::}int.\ \forall x.\ x < z1 \longrightarrow (A1\ x = A2\ x) \Longrightarrow$
$\exists\, z2{::}int.\ \forall x.\ x < z2 \longrightarrow (B1\ x = B2\ x) \Longrightarrow$
$\exists\, z{::}int.\ \forall x.\ x < z \longrightarrow ((A1\ x \wedge B1\ x) = (A2\ x \wedge B2\ x))$
apply (*erule exE*)+
apply (*rule-tac* $x = min\ z1\ z2$ **in** *exI*)
apply *simp*
done

theorem *eq-minf-disjI*: $\exists\, z1{::}int.\ \forall x.\ x < z1 \longrightarrow (A1\ x = A2\ x) \Longrightarrow$
$\exists\, z2{::}int.\ \forall x.\ x < z2 \longrightarrow (B1\ x = B2\ x) \Longrightarrow$
$\exists\, z{::}int.\ \forall x.\ x < z \longrightarrow ((A1\ x \vee B1\ x) = (A2\ x \vee B2\ x))$

apply (*erule exE*)+
apply (*rule-tac* $x = min\ z1\ z2$ **in** *exI*)
apply *simp*
done

Theorems for the combination of proofs of the equality of P and P-m for integers x greather than some integer z.

theorem *eq-pinf-conjI*: $\exists\, z1{::}int.\ \forall x.\ z1 < x \longrightarrow (A1\ x = A2\ x) \Longrightarrow$
$\exists\, z2{::}int.\ \forall x.\ z2 < x \longrightarrow (B1\ x = B2\ x) \Longrightarrow$
$\exists\, z{::}int.\ \forall x.\ z < x \longrightarrow ((A1\ x \wedge B1\ x) = (A2\ x \wedge B2\ x))$
apply (*erule exE*)+
apply (*rule-tac* $x = max\ z1\ z2$ **in** *exI*)
apply *simp*
done

theorem *eq-pinf-disjI*: $\exists\, z1{::}int.\ \forall x.\ z1 < x \longrightarrow (A1\ x = A2\ x) \Longrightarrow$
$\exists\, z2{::}int.\ \forall x.\ z2 < x \longrightarrow (B1\ x = B2\ x) \Longrightarrow$
$\exists\, z{::}int.\ \forall x.\ z < x \longrightarrow ((A1\ x \vee B1\ x) = (A2\ x \vee B2\ x))$
apply (*erule exE*)+
apply (*rule-tac* $x = max\ z1\ z2$ **in** *exI*)
apply *simp*
done

Theorems for the combination of proofs of the modulo D property for P *plusinfinity*

FIXME: This is THE SAME theorem as for the *minusinf* version, but with $+k..$ instead of $-k..$ In the future replace these both with only one.

theorem *modd-pinf-conjI*: $\forall (x{::}int)\ k.\ A\ x = A\ (x{+}k{*}d) \Longrightarrow$
$\forall (x{::}int)\ k.\ B\ x = B\ (x{+}k{*}d) \Longrightarrow$
$\forall (x{::}int)\ (k{::}int).\ (A\ x \wedge B\ x) = (A\ (x{+}k{*}d) \wedge B\ (x{+}k{*}d))$
by *simp*

theorem *modd-pinf-disjI*: $\forall(x::int)\ k.\ A\ x = A\ (x+k*d) \Longrightarrow$
$\forall(x::int)\ k.\ B\ x = B\ (x+k*d) \Longrightarrow$
$\forall(x::int)\ (k::int).\ (A\ x \vee B\ x) = (A\ (x+k*d) \vee B\ (x+k*d))$
by *simp*

This is one of the cases where the simplifed formula is prooved to habe some property (in relation to *P-m*) but we need to prove the property for the original formula (*P-m*)

FIXME: This is exaclty the same thm as for *minusinf*.

lemma *pinf-simp-eq*: *ALL x. P(x) = Q(x) ==> (EX (x::int). P(x)) --> (EX (x::int). F(x)) ==> (EX (x::int). Q(x)) --> (EX (x::int). F(x))*
by *blast*

Theorems for the combination of proofs of the modulo *D* property for *P minusinfinity*

theorem *modd-minf-conjI*: $\forall(x::int)\ k.\ A\ x = A\ (x-k*d) \Longrightarrow$
$\forall(x::int)\ k.\ B\ x = B\ (x-k*d) \Longrightarrow$
$\forall(x::int)\ (k::int).\ (A\ x \wedge B\ x) = (A\ (x-k*d) \wedge B\ (x-k*d))$
by *simp*

theorem *modd-minf-disjI*: $\forall(x::int)\ k.\ A\ x = A\ (x-k*d) \Longrightarrow$
$\forall(x::int)\ k.\ B\ x = B\ (x-k*d) \Longrightarrow$
$\forall(x::int)\ (k::int).\ (A\ x \vee B\ x) = (A\ (x-k*d) \vee B\ (x-k*d))$
by *simp*

This is one of the cases where the simplifed formula is prooved to have some property (in relation to *P-m*) but we need to prove the property for the original formula (*P-m*).

lemma *minf-simp-eq*: *ALL x. P(x) = Q(x) ==> (EX (x::int). P(x)) --> (EX (x::int). F(x)) ==> (EX (x::int). Q(x)) --> (EX (x::int). F(x))*
by *blast*

Theorem needed for proving at runtime divide properties using the arithmetic tactic (which knows only about modulo = 0).

lemma *zdvd-iff-zmod-eq-0*: $(m\ dvd\ n) = (n\ mod\ m = (0::int))$
by(*simp add:dvd-def zmod-eq-0-iff*)

Theorems used for the combination of proof for the backwards direction of Cooper's Theorem. They rely exclusively on Predicate calculus.

lemma *not-ast-p-disjI*: *(ALL x. Q(x::int) ∧ ~(EX (j::int) : {1..d}. EX (a::int) : A. Q(a − j)) --> P1(x) --> P1(x + d))*
==>
(ALL x. Q(x::int) ∧ ~(EX (j::int) : {1..d}. EX (a::int) : A. Q(a − j)) --> P2(x) --> P2(x + d))
==>

$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j)) --> (P1(x) \vee P2(x)) --> (P1(x+d) \vee P2(x+d)))$
by *blast*

lemma *not-ast-p-conjI*: $(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j)) --> P1(x) --> P1(x+d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j)) --> P2(x) --> P2(x+d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j)) --> (P1(x) \wedge P2(x)) --> (P1(x+d) \wedge P2(x+d)))$
by *blast*

lemma *not-ast-p-Q-elim*:
$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j)) --> P(x) --> P(x+d))$
$==>$ $(\ P = Q\ )$
$==> (ALL\ x.\ \sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ P(a-j)) --> P(x) --> P(x+d))$
by *blast*

Theorems used for the combination of proof for the backwards direction of Cooper's Theorem. They rely exclusively on Predicate calculus.

lemma *not-bst-p-disjI*: $(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> P1(x) --> P1(x-d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> P2(x) --> P2(x-d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> (P1(x) \vee P2(x)) --> (P1(x-d) \vee P2(x-d)))$
by *blast*

lemma *not-bst-p-conjI*: $(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> P1(x) --> P1(x-d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> P2(x) --> P2(x-d))$
$==>$
$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> (P1(x) \wedge P2(x)) --> (P1(x-d) \wedge P2(x-d)))$
by *blast*

lemma *not-bst-p-Q-elim*:

$(ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> P(x) --> P(x-d))$
$==>\ (\ P = Q\ )$
$==>\ (ALL\ x.\ \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ P(b+j)) --> P(x) --> P(x-d))$
by *blast*

This is the first direction of Cooper's Theorem.

lemma *cooper-thm*: $(R --> (EX\ x::int.\ P\ x)) ==> (Q --> (EX\ x::int.\ P\ x)) ==> ((R|Q) --> (EX\ x::int.\ P\ x))$
by *blast*

The full Cooper's Theorem in its equivalence Form. Given the premises it is trivial too, it relies exclusively on prediacte calculus.

lemma *cooper-eq-thm*: $(R --> (EX\ x::int.\ P\ x)) ==> (Q --> (EX\ x::int.\ P\ x)) ==> ((\sim Q) --> (EX\ x::int.\ P\ x) --> R) ==> (EX\ x::int.\ P\ x) = R|Q$
by *blast*

Some of the atomic theorems generated each time the atom does not depend on x, they are trivial.

lemma *fm-eq-minf*: $EX\ z::int.\ ALL\ x.\ x < z --> (P = P)$
by *blast*

lemma *fm-modd-minf*: $ALL\ (x::int).\ ALL\ (k::int).\ (P = P)$
by *blast*

lemma *not-bst-p-fm*: $ALL\ (x::int).\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> fm --> fm$
by *blast*

lemma *fm-eq-pinf*: $EX\ z::int.\ ALL\ x.\ z < x --> (P = P)$
by *blast*

The next two thms are the same as the *minusinf* version.

lemma *fm-modd-pinf*: $ALL\ (x::int).\ ALL\ (k::int).\ (P = P)$
by *blast*

lemma *not-ast-p-fm*: $ALL\ (x::int).\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (a::int) : A.\ Q(a-j)) --> fm --> fm$
by *blast*

Theorems to be deleted from simpset when proving simplified formulaes.

lemma *P-eqtrue*: $(P=True) = P$
by *iprover*

lemma *P-eqfalse*: $(P{=}False) = (\sim P)$
by *iprover*

Theorems for the generation of the bachwards direction of Cooper's Theorem.

These are the 6 interesting atomic cases which have to be proved relying on the properties of B-set and the arithmetic and contradiction proofs.

lemma *not-bst-p-lt*: $0 < (d::int) ==>$
$ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> (\ 0 < -x + a) --> (0 < -(x - d) + a\ )$
by *arith*

lemma *not-bst-p-gt*: $[\![\ (g::int) \in B;\ g = -a\ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> (0 < (x) + a) --> (\ 0 < (x - d) + a)$
apply *clarsimp*
apply(*rule ccontr*)
apply(*drule-tac* $x = x+a$ **in** *bspec*)
apply(*simp add*:*atLeastAtMost-iff*)
apply(*drule-tac* $x = -a$ **in** *bspec*)
apply *assumption*
apply(*simp*)
done

lemma *not-bst-p-eq*: $[\![\ 0 < d;\ (g::int) \in B;\ g = -a - 1\ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> (0 = x + a) --> (0 = (x - d) + a\ )$
apply *clarsimp*
apply(*subgoal-tac* $x = -a$)
prefer *2* **apply** *arith*
apply(*drule-tac* $x = 1$ **in** *bspec*)
apply(*simp add*:*atLeastAtMost-iff*)
apply(*drule-tac* $x = -a- 1$ **in** *bspec*)
apply *assumption*
apply(*simp*)
done

lemma *not-bst-p-ne*: $[\![\ 0 < d;\ (g::int) \in B;\ g = -a\ ]\!] \Longrightarrow$
$ALL\ x.\ Q(x::int) \wedge \sim(EX\ (j::int) : \{1..d\}.\ EX\ (b::int) : B.\ Q(b+j)) --> \sim(0 = x + a) --> \sim(0 = (x - d) + a)$
apply *clarsimp*
apply(*subgoal-tac* $x = -a+d$)
prefer *2* **apply** *arith*
apply(*drule-tac* $x = d$ **in** *bspec*)
apply(*simp add*:*atLeastAtMost-iff*)
apply(*drule-tac* $x = -a$ **in** *bspec*)
apply *assumption*

apply(*simp*)
done

lemma *not-bst-p-dvd*: (*d1*::*int*) *dvd d* ==>
ALL x. *Q*(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1*..*d*}. *EX* (*b*::*int*) : *B*. *Q*(*b*+*j*)) −−> *d1 dvd* (*x* + *a*) −−> *d1 dvd* ((*x* − *d*) + *a*)
apply(*clarsimp simp add*:*dvd-def*)
apply(*rename-tac m*)
apply(*rule-tac x* = *m* − *k* **in** *exI*)
apply(*simp add*:*int-distrib*)
done

lemma *not-bst-p-ndvd*: (*d1*::*int*) *dvd d* ==>
ALL x. *Q*(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1*..*d*}. *EX* (*b*::*int*) : *B*. *Q*(*b*+*j*)) −−> ~(*d1 dvd* (*x* + *a*)) −−> ~(*d1 dvd* ((*x* − *d*) + *a*))
apply(*clarsimp simp add*:*dvd-def*)
apply(*rename-tac m*)
apply(*erule-tac x* = *m* + *k* **in** *allE*)
apply(*simp add*:*int-distrib*)
done

Theorems for the generation of the bachwards direction of Cooper's Theorem.

These are the 6 interesting atomic cases which have to be proved relying on the properties of A-set ant the arithmetic and contradiction proofs.

lemma *not-ast-p-gt*: *0* < (*d*::*int*) ==>
ALL x. *Q*(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1*..*d*}. *EX* (*a*::*int*) : *A*. *Q*(*a* − *j*)) −−> (*0* < *x* + *t*) −−> (*0* < (*x* + *d*) + *t*)
by *arith*

lemma *not-ast-p-lt*: ⟦*0* < *d* ;(*t*::*int*) ∈ *A* ⟧ ⟹
ALL x. *Q*(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1*..*d*}. *EX* (*a*::*int*) : *A*. *Q*(*a* − *j*)) −−> (*0* < −*x* + *t*) −−> (*0* < −(*x* + *d*) + *t*)
apply *clarsimp*
apply (*rule ccontr*)
apply (*drule-tac x* = *t*−*x* **in** *bspec*)
apply *simp*
apply (*drule-tac x* = *t* **in** *bspec*)
apply *assumption*
apply *simp*
done

lemma *not-ast-p-eq*: ⟦ *0* < *d*; (*g*::*int*) ∈ *A*; *g* = −*t* + *1* ⟧ ⟹
ALL x. *Q*(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1*..*d*}. *EX* (*a*::*int*) : *A*. *Q*(*a* − *j*)) −−> (*0* = *x* + *t*) −−> (*0* = (*x* + *d*) + *t*)
apply *clarsimp*
apply (*drule-tac x=1* **in** *bspec*)

apply *simp*
apply (*drule-tac x=− t + 1* **in** *bspec*)
apply *assumption*
apply(*subgoal-tac x = −t*)
prefer *2* **apply** *arith*
apply *simp*
done

lemma *not-ast-p-ne*: ⟦ *0 < d*; (*g*::*int*) ∈ *A*; *g = −t* ⟧ ⟹
ALL x. Q(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1..d*}. *EX* (*a*::*int*) : *A. Q*(*a − j*)) −−> ~(*0 = x + t*) −−> ~(*0 = (x + d) + t*)
apply *clarsimp*
apply (*subgoal-tac x = −t−d*)
prefer *2* **apply** *arith*
apply (*drule-tac x = d* **in** *bspec*)
apply *simp*
apply (*drule-tac x = −t* **in** *bspec*)
apply *assumption*
apply *simp*
done

lemma *not-ast-p-dvd*: (*d1*::*int*) *dvd d* ==>
ALL x. Q(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1..d*}. *EX* (*a*::*int*) : *A. Q*(*a − j*)) −−> *d1 dvd* (*x + t*) −−> *d1 dvd* ((*x + d*) + *t*)
apply(*clarsimp simp add*:*dvd-def*)
apply(*rename-tac m*)
apply(*rule-tac x = m + k* **in** *exI*)
apply(*simp add*:*int-distrib*)
done

lemma *not-ast-p-ndvd*: (*d1*::*int*) *dvd d* ==>
ALL x. Q(*x*::*int*) ∧ ~(*EX* (*j*::*int*) : {*1..d*}. *EX* (*a*::*int*) : *A. Q*(*a − j*)) −−> ~(*d1 dvd* (*x + t*)) −−> ~(*d1 dvd* ((*x + d*) + *t*))
apply(*clarsimp simp add*:*dvd-def*)
apply(*rename-tac m*)
apply(*erule-tac x = m − k* **in** *allE*)
apply(*simp add*:*int-distrib*)
done

These are the atomic cases for the proof generation for the modulo *D* property for *P plusinfinity*

They are fully based on arithmetics.

lemma *dvd-modd-pinf*: ((*d*::*int*) *dvd d1*) ==>
(*ALL* (*x*::*int*). *ALL* (*k*::*int*). (((*d*::*int*) *dvd* (*x + t*)) = (*d dvd* (*x+k∗d1 + t*))))
apply(*clarsimp simp add*:*dvd-def*)
apply(*rule iffI*)
apply(*clarsimp*)
apply(*rename-tac n m*)

apply(*rule-tac* $x = m + n*k$ **in** *exI*)
apply(*simp add*:*int-distrib*)
apply(*clarsimp*)
apply(*rename-tac n m*)
apply(*rule-tac* $x = m - n*k$ **in** *exI*)
apply(*simp add*:*int-distrib mult-ac*)
done

lemma *not-dvd-modd-pinf*: $((d::int)\ dvd\ d1) ==>$
$(ALL\ (x::int).\ ALL\ k.\ (\sim((d::int)\ dvd\ (x + t))) = (\sim(d\ dvd\ (x+k*d1 + t))))$
apply(*clarsimp simp add*:*dvd-def*)
apply(*rule iffI*)
apply(*clarsimp*)
apply(*rename-tac n m*)
apply(*erule-tac* $x = m - n*k$ **in** *allE*)
apply(*simp add*:*int-distrib mult-ac*)
apply(*clarsimp*)
apply(*rename-tac n m*)
apply(*erule-tac* $x = m + n*k$ **in** *allE*)
apply(*simp add*:*int-distrib mult-ac*)
done

These are the atomic cases for the proof generation for the equivalence of P and P *plusinfinity* for integers x greater than some integer z.
They are fully based on arithmetics.

lemma *eq-eq-pinf*: $EX\ z::int.\ ALL\ x.\ z < x \longrightarrow ((\ 0 = x + t\ ) = False\ )$
apply(*rule-tac* $x = -t$ **in** *exI*)
apply *simp*
done

lemma *neq-eq-pinf*: $EX\ z::int.\ ALL\ x.\ \ z < x \longrightarrow ((\sim(\ 0 = x + t\ )) = True\ )$
apply(*rule-tac* $x = -t$ **in** *exI*)
apply *simp*
done

lemma *le-eq-pinf*: $EX\ z::int.\ ALL\ x.\ \ z < x \longrightarrow (\ 0 < x + t\ = True\ )$
apply(*rule-tac* $x = -t$ **in** *exI*)
apply *simp*
done

lemma *len-eq-pinf*: $EX\ z::int.\ ALL\ x.\ z < x\ \longrightarrow (0 < -x + t\ = False\ )$
apply(*rule-tac* $x = t$ **in** *exI*)
apply *simp*
done

lemma *dvd-eq-pinf*: $EX\ z::int.\ ALL\ x.\ \ z < x \longrightarrow ((d\ dvd\ (x + t)) = (d\ dvd\ (x + t)))$
by *simp*

lemma *not-dvd-eq-pinf*: *EX z::int. ALL x. z < x −−> ((~(d dvd (x + t))) = (~(d dvd (x + t))))*
by *simp*

These are the atomic cases for the proof generation for the modulo D property for P *minusinfinity*.

They are fully based on arithmetics.

lemma *dvd-modd-minf*: *((d::int) dvd d1) ==> (ALL (x::int). ALL (k::int). (((d::int) dvd (x + t)) = (d dvd (x−k∗d1 + t))))*
apply(*clarsimp simp add:dvd-def*)
apply(*rule iffI*)
apply(*clarsimp*)
apply(*rename-tac n m*)
apply(*rule-tac x = m − n∗k* **in** *exI*)
apply(*simp add:int-distrib*)
apply(*clarsimp*)
apply(*rename-tac n m*)
apply(*rule-tac x = m + n∗k* **in** *exI*)
apply(*simp add:int-distrib mult-ac*)
done

lemma *not-dvd-modd-minf*: *((d::int) dvd d1) ==> (ALL (x::int). ALL k. (~((d::int) dvd (x + t))) = (~(d dvd (x−k∗d1 + t))))*
apply(*clarsimp simp add:dvd-def*)
apply(*rule iffI*)
apply(*clarsimp*)
apply(*rename-tac n m*)
apply(*erule-tac x = m + n∗k* **in** *allE*)
apply(*simp add:int-distrib mult-ac*)
apply(*clarsimp*)
apply(*rename-tac n m*)
apply(*erule-tac x = m − n∗k* **in** *allE*)
apply(*simp add:int-distrib mult-ac*)
done

These are the atomic cases for the proof generation for the equivalence of P and P *minusinfinity* for integers x less than some integer z.

They are fully based on arithmetics.

lemma *eq-eq-minf*: *EX z::int. ALL x. x < z −−> ((0 = x +t) = False)*
apply(*rule-tac x = −t* **in** *exI*)
apply *simp*
done

lemma *neq-eq-minf*: *EX z::int. ALL x. x < z −−> ((~(0 = x +t)) = True)*
apply(*rule-tac x = −t* **in** *exI*)

apply *simp*
done

lemma *le-eq-minf*: *EX z::int. ALL x. x < z −−> (0 < x +t = False)*
apply(*rule-tac x = −t* **in** *exI*)
apply *simp*
done

lemma *len-eq-minf*: *EX z::int. ALL x. x < z −−> (0 < −x +t = True)*
apply(*rule-tac x = t* **in** *exI*)
apply *simp*
done

lemma *dvd-eq-minf*: *EX z::int. ALL x. x < z −−> ((d dvd (x + t)) = (d dvd (x + t)))*
by *simp*

lemma *not-dvd-eq-minf*: *EX z::int. ALL x. x < z −−> ((~(d dvd (x + t))) = (~(d dvd (x + t))))*
by *simp*

This Theorem combines whithnesses about *P minusinfinity* to show one component of the equivalence proof for Cooper's Theorem.

FIXME: remove once they are part of the distribution.

theorem *int-ge-induct*[*consumes 1*,*case-names base step*]:
assumes *ge*: $k \leq (i::int)$ **and**
base: $P(k)$ **and**
step: $\bigwedge i.\ [\![k \leq i;\ P\ i]\!] \Longrightarrow P(i+1)$
shows *P i*
proof −
{ **fix** *n* **have** $\bigwedge i::int.\ n = nat(i-k) \Longrightarrow k <= i \Longrightarrow P\ i$
proof (*induct n*)
case *0*
hence *i = k* **by** *arith*
thus *P i* **using** *base* **by** *simp*
next
case (*Suc n*)
hence *n = nat((i − 1) − k)* **by** *arith*
moreover
have *ki1*: *k ≤ 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
}
from *this ge* **show** *?thesis* **by** *fast*
qed

theorem *int-gr-induct*[*consumes 1*,*case-names base step*]:
  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*)
 apply *simp+*
done

lemma *decr-lemma*: $0 < (d::int) \Longrightarrow x - (abs(x-z)+1) * d < z$
apply(*induct rule*: *int-gr-induct*)
 apply *simp*
 apply *arith*
apply (*simp add*:*int-distrib*)
apply *arith*
done

lemma *incr-lemma*: $0 < (d::int) \Longrightarrow z < x + (abs(x-z)+1) * d$
apply(*induct rule*: *int-gr-induct*)
 apply *simp*
 apply *arith*
apply (*simp add*:*int-distrib*)
apply *arith*
done

lemma *minusinfinity*:
  assumes $0 < d$ **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 $?w = x - (abs(x-z)+1) * d$
  show $EX\ x.\ P\ x$
  proof
    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 show $P\ ?w$ **using** *P1* **by** *blast*
  qed
qed

This Theorem combines whithnesses about P *minusinfinity* to show one

component of the equivalence proof for Cooper's Theorem.

lemma *plusinfinity*:
assumes $0 < d$ **and**
$P1eqP1$: $ALL\ (x{::}int).\ (k{::}int).\ P1\ x = P1\ (x + k * d)$ **and**
$ePeqP1$: $EX\ z{::}int.\ ALL\ x.\ z < x\ \text{--}\text{>}\ (P\ x = P1\ x)$
shows $(EX\ x{::}int.\ P1\ x)\ \text{--}\text{>}\ (EX\ x{::}int.\ P\ x)$
proof
assume $eP1$: $EX\ x.\ P1\ x$
then obtain x **where** $P1$: $P1\ x$ **..**
from $ePeqP1$ **obtain** z **where** $P1eqP$: $ALL\ x.\ z < x \longrightarrow (P\ x = P1\ x)$ **..**
let $?w = x + (abs(x-z)+1) * d$
show $EX\ x.\ P\ x$
proof
have w: $z < ?w$ **by**(*rule incr-lemma*)
have $P1\ x = P1\ ?w$ **using** $P1eqP1$ **by** *blast*
also have $\ldots = P(?w)$ **using** $w\ P1eqP$ **by** *blast*
finally show $P\ ?w$ **using** $P1$ **by** *blast*
qed
qed

Theorem for periodic function on discrete sets.

lemma *minf-vee*:
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*: $x\ mod\ d \neq 0$
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 $-$
have $0 \leq x\ mod\ d$ **by**(*rule pos-mod-sign*)
moreover 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
next
  assume ?RHS thus ?LHS by blast
qed
```

Theorem for periodic function on discrete sets.

```
lemma pinf-vee:
  assumes dpos: 0 < (d::int) and modd: ALL (x::int) (k::int). P x = P (x+k*d)
  shows (EX x::int. P x) = (EX (j::int) : {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 only:)
  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 -
      have 0 ≤ x mod d by(rule pos-mod-sign)
      moreover 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
next
  assume ?RHS thus ?LHS by blast
qed

lemma decr-mult-lemma:
  assumes dpos: (0::int) < d and
          minus: 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)
  show ?case
  proof
    fix x
    have P x ⟶ P (x − i ∗ d) using step.hyps by blast
    also have ... ⟶ P(x − (i + 1) ∗ d)
      using minus[THEN spec, of x − i ∗ d]
      by (simp add:int-distrib OrderedGroup.diff-diff-eq[symmetric])
    ultimately show P x ⟶ P(x − (i + 1) ∗ d) by blast
  qed
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)
  show ?case
  proof
    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 show P x ⟶ P(x + (i + 1) ∗ d) by blast
  qed
qed

lemma cpmi-eq: 0 < D ⟹ (EX z::int. ALL x. x < z −−> (P x = P1 x))
==> ALL x.~(EX (j::int) : {1..D}. EX (b::int) : B. P(b+j)) −−> P (x) −−>
P (x − D)
==> (ALL (x::int). ALL (k::int). ((P1 x)= (P1 (x−k∗D))))
==> (EX (x::int). P(x)) = ((EX (j::int) : {1..D} . (P1(j))) | (EX (j::int) :
{1..D}. EX (b::int) : B. P (b+j)))
apply(rule iffI)
prefer 2
apply(drule minusinfinity)
apply assumption+
apply(fastsimp)
apply clarsimp
apply(subgoal-tac !!k. 0<=k ⟹ !x. P x ⟶ P (x − k∗D))
apply(frule-tac x = x and z=z in decr-lemma)
apply(subgoal-tac P1(x − (|x − z| + 1) ∗ D))
prefer 2
```

apply(*subgoal-tac 0 <= (|x − z| + 1)*)
prefer *2* **apply** *arith*
apply *fastsimp*
apply(*drule (1) minf-vee*)
apply *blast*
apply(*blast dest*:*decr-mult-lemma*)
done

Cooper Theorem, plus infinity version.

lemma *cppi-eq*: *0 < D ⟹ (EX z*::*int. ALL x. z < x −−> (P x = P1 x))*
==> ALL x.~*(EX (j*::*int) : {1..D}. EX (a*::*int) : A. P(a − j)) −−> P (x) −−> P (x + D)*
==> (ALL (x::*int). ALL (k*::*int). ((P1 x)= (P1 (x+k∗D))))*
==> (EX (x::*int). P(x)) = ((EX (j*::*int) : {1..D} . (P1(j))) | (EX (j*::*int) : {1..D}. EX (a*::*int) : A. P (a − j)))*
apply(*rule iffI*)
prefer *2*
apply(*drule plusinfinity*)
apply *assumption+*
apply(*fastsimp*)
apply *clarsimp*
apply(*subgoal-tac !!k. 0<=k ⟹ !x. P x ⟶ P (x + k∗D)*)
apply(*frule-tac x = x* **and** *z=z* **in** *incr-lemma*)
apply(*subgoal-tac P1(x + (|x − z| + 1) ∗ D)*)
prefer *2*
apply(*subgoal-tac 0 <= (|x − z| + 1)*)
prefer *2* **apply** *arith*
apply *fastsimp*
apply(*drule (1) pinf-vee*)
apply *blast*
apply(*blast dest*:*incr-mult-lemma*)
done

Theorems for the quantifier elmininations Functions.

lemma *qe-ex-conj*: *(EX (x*::*int). A x) = R*
==> (EX (x::*int). P x) = (Q & (EX x*::*int. A x))*
==> (EX (x::*int). P x) = (Q & R)*
by *blast*

lemma *qe-ex-nconj*: *(EX (x*::*int). P x) = (True & Q)*
==> (EX (x::*int). P x) = Q*
by *blast*

lemma *qe-conjI*: *P1 = P2 ==> Q1 = Q2 ==> (P1 & Q1) = (P2 & Q2)*
by *blast*

lemma *qe-disjI*: *P1 = P2 ==> Q1 = Q2 ==> (P1 | Q1) = (P2 | Q2)*
by *blast*

lemma *qe-impI*: *P1 = P2 ==> Q1 = Q2 ==> (P1 --> Q1) = (P2 --> Q2)*
by *blast*

lemma *qe-eqI*: *P1 = P2 ==> Q1 = Q2 ==> (P1 = Q1) = (P2 = Q2)*
by *blast*

lemma *qe-Not*: *P = Q ==> (~P) = (~Q)*
by *blast*

lemma *qe-ALL*: *(EX x. ~P x) = R ==> (ALL x. P x) = (~R)*
by *blast*

Theorems for proving NNF

lemma *nnf-im*: *((~P) = P1) ==> (Q=Q1) ==> ((P --> Q) = (P1 | Q1))*
by *blast*

lemma *nnf-eq*: *((P & Q) = (P1 & Q1)) ==> (((~P) & (~Q)) = (P2 & Q2)) ==> ((P = Q) = ((P1 & Q1)|(P2 & Q2)))*
by *blast*

lemma *nnf-nn*: *(P = Q) ==> ((~~P) = Q)*
by *blast*
lemma *nnf-ncj*: *((~P) = P1) ==> ((~Q) = Q1) ==> ((~(P & Q)) = (P1 | Q1))*
by *blast*

lemma *nnf-ndj*: *((~P) = P1) ==> ((~Q) = Q1) ==> ((~(P | Q)) = (P1 & Q1))*
by *blast*
lemma *nnf-nim*: *(P = P1) ==> ((~Q) = Q1) ==> ((~(P --> Q)) = (P1 & Q1))*
by *blast*
lemma *nnf-neq*: *((P & (~Q)) = (P1 & Q1)) ==> (((~P) & Q) = (P2 & Q2)) ==> ((~(P = Q)) = ((P1 & Q1)|(P2 & Q2)))*
by *blast*
lemma *nnf-sdj*: *((A & (~B)) = (A1 & B1)) ==> ((C & (~D)) = (C1 & D1)) ==> (A = (~C)) ==> ((~((A & B) | (C & D))) = ((A1 & B1) | (C1 & D1)))*
by *blast*

lemma *qe-exI2*: *A = B ==> (EX (x::int). A(x)) = (EX (x::int). B(x))*
by *simp*

lemma *qe-exI*: *(!!x::int. A x = B x) ==> (EX (x::int). A(x)) = (EX (x::int). B(x))*
by *iprover*

lemma *qe-ALLI*: (!!x::*int*. $A\ x = B\ x$) ==> (*ALL* (x::*int*). $A(x)$) = (*ALL* (x::*int*). $B(x)$)
by *iprover*

lemma *cp-expand*: (*EX* (x::*int*). $P\ (x)$) = (*EX* (j::*int*) : {*1*..d}. *EX* (b::*int*) : B. (*P1* (j) | $P(b+j)$)))
==>(*EX* (x::*int*). $P\ (x)$) = (*EX* (j::*int*) : {*1*..d}. *EX* (b::*int*) : B. (*P1* (j) | $P(b+j)$)))
by *blast*

lemma *cppi-expand*: (*EX* (x::*int*). $P\ (x)$) = (*EX* (j::*int*) : {*1*..d}. *EX* (a::*int*) : A. (*P1* (j) | $P(a - j)$)))
==>(*EX* (x::*int*). $P\ (x)$) = (*EX* (j::*int*) : {*1*..d}. *EX* (a::*int*) : A. (*P1* (j) | $P(a - j)$)))
by *blast*

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

Theorems required for the *adjustcoeffitienteq*

lemma *ac-dvd-eq*: **assumes** *not0*: *0* ~= (k::*int*)
shows ((m::*int*) *dvd* (c*n+t)) = (k*m *dvd* ((k*c)*n+(k*t))) (**is** *?P* = *?Q*)
proof
assume *?P*
thus *?Q*
apply(*simp add*:*dvd-def*)
apply *clarify*
apply(*rename-tac d*)
apply(*drule-tac f* = *op* * *k* **in** *arg-cong*)
apply(*simp only*:*int-distrib*)
apply(*rule-tac x* = *d* **in** *exI*)
apply(*simp only*:*mult-ac*)
done
next
assume *?Q*
then obtain *d* **where** $k * c * n + k * t = (k*m)*d$ **by**(*fastsimp simp*:*dvd-def*)
hence $(c * n + t) * k = (m*d) * k$ **by**(*simp add*:*int-distrib mult-ac*)
hence $((c * n + t) * k)$ *div* $k = ((m*d) * k)$ *div* k **by**(*rule arg-cong*[*of* - - %*t*. *t div k*])
hence $c*n+t = m*d$ **by**(*simp add*: *zdiv-zmult-self1*[*OF not0*[*symmetric*]])
thus *?P* **by**(*simp add*:*dvd-def*)
qed

lemma *ac-lt-eq*: **assumes** *gr0*: *0* < (k::*int*)

shows $((m::int) < (c*n+t)) = (k*m < ((k*c)*n+(k*t)))$ (**is** *?P = ?Q*)
proof
assume *P*: *?P*
show *?Q* **using** *zmult-zless-mono2*[*OF P gr0*] **by**(*simp add*: *int-distrib mult-ac*)
next
assume *?Q*
hence $0 < k*(c*n + t - m)$ **by**(*simp add*: *int-distrib mult-ac*)
with *gr0* **have** $0 < (c*n + t - m)$ **by**(*simp add*: *zero-less-mult-iff*)
thus *?P* **by**(*simp*)
qed

lemma *ac-eq-eq* : **assumes** *not0*: $0 \sim= (k::int)$ **shows** $((m::int) = (c*n+t)) = (k*m = ((k*c)*n+(k*t)))$ (**is** *?P = ?Q*)
proof
assume *?P*
thus *?Q*
apply(*drule-tac f = op * k* **in** *arg-cong*)
apply(*simp only*:*int-distrib*)
done
next
assume *?Q*
hence $m * k = (c*n + t) * k$ **by**(*simp add*:*int-distrib mult-ac*)
hence $((m) * k)\ div\ k = ((c*n + t) * k)\ div\ k$ **by**(*rule arg-cong*[*of - - %t. t div k*])
thus *?P* **by**(*simp add*: *zdiv-zmult-self1*[*OF not0*[*symmetric*]])
qed

lemma *ac-pi-eq*: **assumes** *gr0*: $0 < (k::int)$ **shows** $(\sim((0::int) < (c*n + t))) = (0 < ((-k)*c)*n + ((-k)*t + k))$
proof −
have $(\sim (0::int) < (c*n + t)) = (0<1-(c*n + t))$ **by** *arith*
also have $(1-(c*n + t)) = (-1*c)*n + (-t+1)$ **by**(*simp add*: *int-distrib mult-ac*)
also have $0<(-1*c)*n + (-t+1) = (0 < (k*(-1*c)*n) + (k*(-t+1)))$ **by**(*rule ac-lt-eq*[*of - 0*,*OF gr0*,*simplified*])
also have $(k*(-1*c)*n) + (k*(-t+1)) = ((-k)*c)*n + ((-k)*t + k)$ **by**(*simp add*: *int-distrib mult-ac*)
finally show *?thesis* **.**
qed

lemma *binminus-uminus-conv*: $(a::int) - b = a + (-b)$
by *arith*

lemma *linearize-dvd*: $(t::int) = t1 ==> (d\ dvd\ t) = (d\ dvd\ t1)$
by *simp*

lemma *lf-lt*: $(l::int) = ll ==> (r::int) = lr ==> (l < r) = (ll < lr)$
by *simp*

lemma *lf-eq*: $(l::int) = ll \Longrightarrow (r::int) = lr \Longrightarrow (l = r) = (ll = lr)$
by *simp*

lemma *lf-dvd*: $(l::int) = ll \Longrightarrow (r::int) = lr \Longrightarrow (l\ dvd\ r) = (ll\ dvd\ lr)$
by *simp*

Theorems for transforming predicates on nat to predicates on *int*

theorem *all-nat*: $(\forall x::nat.\ P\ x) = (\forall x::int.\ 0 <= x \longrightarrow P\ (nat\ x))$
by (*simp split add*: *split-nat*)

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 *zdiff-int-split*: $P\ (int\ (x - y)) =$
$((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
apply (*case-tac* $y \leq x$)
apply (*simp-all add*: *zdiff-int*)
done

theorem *zdvd-int*: $(x\ dvd\ y) = (int\ x\ dvd\ int\ y)$
apply (*simp only*: *dvd-def ex-nat int-int-eq* [*symmetric*] *zmult-int* [*symmetric*]
nat-0-le cong add: *conj-cong*)
apply (*rule iffI*)
apply *iprover*
apply (*erule exE*)
apply (*case-tac x=0*)
apply (*rule-tac x=0* **in** *exI*)
apply *simp*
apply (*case-tac* $0 \leq k$)
apply *iprover*
apply (*simp add*: *linorder-not-le*)
apply (*drule mult-strict-left-mono-neg* [*OF iffD2* [*OF zero-less-int-conv*]])
apply *assumption*
apply (*simp add*: *mult-ac*)
done

theorem *number-of1*: $(0::int) <= number\text{-}of\ n \Longrightarrow (0::int) <= number\text{-}of\ (n\ BIT\ b)$

by *simp*

theorem *number-of2*: $(0::int) <= Numeral0$ **by** *simp*

theorem *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*)

use *cooper-dec.ML*
use *reflected-presburger.ML*
use *reflected-cooper.ML*
oracle
presburger-oracle (*term*) = *ReflectedCooper.presburger-oracle*

use *cooper-proof.ML*
use *qelim.ML*
use *presburger.ML*

setup *Presburger.setup*

end

33 Relation-Power: Powers of Relations and Functions

theory *Relation-Power*
imports *Nat*
begin

instance
set :: (*type*) *power* **..**
— only type $('a \times 'a)$ *set* should be in class *power*!

primrec (*relpow*)
$R\hat{}0 = Id$
$R\hat{}(Suc\ n) = R\ O\ (R\hat{}n)$

instance

fun :: (*type*, *type*) *power* **..**
— only type $'a \Rightarrow 'a$ should be in class *power*!

primrec (*funpow*)
f^0 = id
f^(Suc n) = f o (f^n)

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.

lemma *funpow-add*: *f ^ (m+n) = f^m o f^n*
by(*induct m*) *simp-all*

lemma *rel-pow-1*: !!*R*:: $('a*'a)$*set*. *R^1 = R*
by *simp*
declare *rel-pow-1* [*simp*]

lemma *rel-pow-0-I*: *(x,x) : R^0*
by *simp*

lemma *rel-pow-Suc-I*: [| *(x,y) : R^n*; *(y,z):R* |] ==> *(x,z):R^(Suc n)*
apply (*auto*)
done

lemma *rel-pow-Suc-I2* [*rule-format*]:
∀ *z*. *(x,y) : R --> (y,z):R^n --> (x,z):R^(Suc n)*
apply (*induct-tac n*, *simp-all*)
apply *blast*
done

lemma *rel-pow-0-E*: [| *(x,y) : R^0*; *x=y ==> P* |] ==> *P*
by *simp*

lemma *rel-pow-Suc-E*:
[| *(x,z) : R^(Suc n)*; !!*y*. [| *(x,y) : R^n*; *(y,z) : R* |] ==> *P* |] ==> *P*
by *auto*

lemma *rel-pow-E*:
[| *(x,z) : R^n*; [| *n=0*; *x = z* |] ==> *P*;
!!*y m*. [| *n = Suc m*; *(x,y) : R^m*; *(y,z) : R* |] ==> *P*
|] ==> *P*
by (*case-tac n*, *auto*)

lemma *rel-pow-Suc-D2* [*rule-format*]:
∀ *x z*. *(x,z):R^(Suc n) --> (∃ y. (x,y):R & (y,z):R^n)*
apply (*induct-tac n*)
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'*:
  $\forall x\ y\ z.\ (x,y) : R \hat{}\ n\ \&\ (y,z) : R \longrightarrow (\exists w.\ (x,w) : R\ \&\ (w,z) : R \hat{}\ n)$
by (*induct-tac n*, *simp-all*, *blast*)

lemma *rel-pow-E2*:
  $[|\ (x,z) : R \hat{}\ n;\ [|\ n=0;\ x = z\ |] \Longrightarrow P;$
    $!!y\ m.\ [|\ n = Suc\ m;\ (x,y) : R;\ (y,z) : R \hat{}\ m\ |] \Longrightarrow P$
  $|] \Longrightarrow P$
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*: $!!p.\ p:R \hat{}\ * \Longrightarrow p : (UN\ n.\ R \hat{}\ n)$
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*: $!!p.\ p:R \hat{}\ n \Longrightarrow p:R \hat{}\ *$
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*: $R \hat{}\ * = (UN\ n.\ R \hat{}\ n)$
by (*blast intro*: *rtrancl-imp-UN-rel-pow rel-pow-imp-rtrancl*)

lemma *single-valued-rel-pow* [*rule-format*]:
  $!!r::('a * 'a)set.\ single\text{-}valued\ r \Longrightarrow single\text{-}valued\ (r \hat{}\ n)$
apply (*rule single-valuedI*)
apply (*induct-tac n*, *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

34 Parity: Even and Odd for ints and nats

theory *Parity*
imports *Divides IntDiv NatSimprocs*
begin

axclass *even-odd* < *type*

instance *int* :: *even-odd* **..**
instance *nat* :: *even-odd* **..**

consts
  even :: *'a*::*even-odd* => *bool*

syntax
  odd :: *'a*::*even-odd* => *bool*

translations
  odd x == ~*even x*

defs (**overloaded**)
  even-def: *even* (*x*::*int*) == *x mod 2 = 0*
  even-nat-def: *even* (*x*::*nat*) == *even* (*int x*)

34.1 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 *auto*

lemma *neq-one-mod-two* [*simp*]: ((*x*::*int*) *mod 2* ~= *0*) = (*x mod 2 = 1*)
  apply (*subgoal-tac x mod 2 = 0* | *x mod 2 = 1*, *force*)
  apply (*rule int-pos-lt-two-imp-zero-or-one*, *auto*)
  done

34.2 Behavior under integer arithmetic operations

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

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

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

lemma *even-product*: *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 (*simp add*: *even-def zmod-zadd1-eq*)

lemma *even-plus-odd*: *even* (*x*::*int*) ==> *odd y* ==> *odd* (*x* + *y*)
by (*simp add*: *even-def zmod-zadd1-eq*)

lemma *odd-plus-even*: *odd* (*x*::*int*) ==> *even y* ==> *odd* (*x* + *y*)
by (*simp add*: *even-def zmod-zadd1-eq*)

lemma *odd-plus-odd*: *odd* (*x*::*int*) ==> *odd y* ==> *even* (*x* + *y*)
by (*simp add*: *even-def zmod-zadd1-eq*)

lemma *even-sum*: *even* ((*x*::*int*) + *y*) = ((*even x* & *even y*) | (*odd x* & *odd y*))
apply (*auto intro*: *even-plus-even odd-plus-odd*)
apply (*rule ccontr*, *simp add*: *even-plus-odd*)
apply (*rule ccontr*, *simp add*: *odd-plus-even*)
done

lemma *even-neg*: *even* (−(*x*::*int*)) = *even x*
by (*auto simp add*: *even-def zmod-zminus1-eq-if*)

lemma *even-difference*:
even ((*x*::*int*) − *y*) = ((*even x* & *even y*) | (*odd x* & *odd y*))
by (*simp only*: *diff-minus even-sum even-neg*)

lemma *even-pow-gt-zero* [*rule-format*]:
even (*x*::*int*) ==> *0* < *n* −−> *even* (*x*^*n*)
apply (*induct n*)
apply (*auto simp add*: *even-product*)
done

lemma *odd-pow*: *odd x* ==> *odd*((*x*::*int*)^*n*)
apply (*induct n*)

apply (*simp add*: *even-def*)
apply (*simp add*: *even-product*)
done

lemma *even-power*: *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*: *even* (*0*::*int*)
by (*simp add*: *even-def*)

lemma *odd-one*: *odd* (*1*::*int*)
by (*simp add*: *even-def*)

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*

34.3 Equivalent definitions

lemma *two-times-even-div-two*: *even* (*x*::*int*) ==> *2* * (*x div 2*) = *x*
by (*auto simp add*: *even-def*)

lemma *two-times-odd-div-two-plus-one*: *odd* (*x*::*int*) ==>
2 * (*x div 2*) + *1* = *x*
apply (*insert zmod-zdiv-equality* [*of x 2*, *THEN sym*])
by (*simp add*: *even-def*)

lemma *even-equiv-def*: *even* (*x*::*int*) = (*EX y*. *x* = *2* * *y*)
apply *auto*
apply (*rule exI*)
by (*erule two-times-even-div-two* [*THEN sym*])

lemma *odd-equiv-def*: *odd* (*x*::*int*) = (*EX y*. *x* = *2* * *y* + *1*)
apply *auto*
apply (*rule exI*)
by (*erule two-times-odd-div-two-plus-one* [*THEN sym*])

34.4 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*: *even*((*x*::*nat*) * *y*) = (*even x* | *even y*)
by (*simp add*: *even-nat-def int-mult*)

lemma *even-nat-sum*: *even* ((*x*::*nat*) + *y*) =
((*even x* & *even y*) | (*odd x* & *odd y*))
by (*unfold even-nat-def*, *simp*)

lemma *even-nat-difference*:
  even ((*x*::*nat*) − *y*) = (*x* < *y* | (*even x* & *even y*) | (*odd x* & *odd y*))
 apply (*auto simp add*: *even-nat-def zdiff-int* [*THEN sym*])
 apply (*case-tac x* < *y*, *auto simp add*: *zdiff-int* [*THEN sym*])
 apply (*case-tac x* < *y*, *auto simp add*: *zdiff-int* [*THEN sym*])
 done

lemma *even-nat-Suc*: *even* (*Suc x*) = *odd x*
 by (*simp add*: *even-nat-def*)

lemma *even-nat-power*: *even* ((*x*::*nat*) ^ *y*) = (*even x* & 0 < *y*)
 by (*simp add*: *even-nat-def int-power*)

lemma *even-nat-zero*: *even* (*0*::*nat*)
 by (*simp add*: *even-nat-def*)

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

34.5 Equivalent definitions

lemma *nat-lt-two-imp-zero-or-one*: (*x*::*nat*) < *Suc* (*Suc 0*) ==>
  x = *0* | *x* = *Suc 0*
 by *auto*

lemma *even-nat-mod-two-eq-zero*: *even* (*x*::*nat*) ==> *x mod* (*Suc* (*Suc 0*)) = *0*
 apply (*insert mod-div-equality* [*of x Suc* (*Suc 0*), *THEN sym*])
 apply (*drule subst*, *assumption*)
 apply (*subgoal-tac x mod Suc* (*Suc 0*) = *0* | *x mod Suc* (*Suc 0*) = *Suc 0*)
 apply *force*
 apply (*subgoal-tac 0* < *Suc* (*Suc 0*))
 apply (*frule mod-less-divisor* [*of Suc* (*Suc 0*) *x*])
 apply (*erule nat-lt-two-imp-zero-or-one*, *auto*)
 done

lemma *odd-nat-mod-two-eq-one*: *odd* (*x*::*nat*) ==> *x mod* (*Suc* (*Suc 0*)) = *Suc 0*
 apply (*insert mod-div-equality* [*of x Suc* (*Suc 0*), *THEN sym*])
 apply (*drule subst*, *assumption*)
 apply (*subgoal-tac x mod Suc* (*Suc 0*) = *0* | *x mod Suc* (*Suc 0*) = *Suc 0*)
 apply *force*
 apply (*subgoal-tac 0* < *Suc* (*Suc 0*))
 apply (*frule mod-less-divisor* [*of Suc* (*Suc 0*) *x*])
 apply (*erule nat-lt-two-imp-zero-or-one*, *auto*)
 done

lemma *even-nat-equiv-def*: *even* (*x*::*nat*) = (*x mod Suc* (*Suc 0*) = *0*)
 apply (*rule iffI*)
 apply (*erule even-nat-mod-two-eq-zero*)

apply (*insert odd-nat-mod-two-eq-one* [*of x*], *auto*)
done

lemma *odd-nat-equiv-def*: *odd* (*x*::*nat*) = (*x mod Suc* (*Suc 0*) = *Suc 0*)
apply (*auto simp add*: *even-nat-equiv-def*)
apply (*subgoal-tac x mod* (*Suc* (*Suc 0*)) < *Suc* (*Suc 0*))
apply (*frule nat-lt-two-imp-zero-or-one*, *auto*)
done

lemma *even-nat-div-two-times-two*: *even* (*x*::*nat*) ==>
Suc (*Suc 0*) * (*x div Suc* (*Suc 0*)) = *x*
apply (*insert mod-div-equality* [*of x Suc* (*Suc 0*), *THEN sym*])
apply (*drule even-nat-mod-two-eq-zero*, *simp*)
done

lemma *odd-nat-div-two-times-two-plus-one*: *odd* (*x*::*nat*) ==>
Suc(*Suc* (*Suc 0*) * (*x div Suc* (*Suc 0*))) = *x*
apply (*insert mod-div-equality* [*of x Suc* (*Suc 0*), *THEN sym*])
apply (*drule odd-nat-mod-two-eq-one*, *simp*)
done

lemma *even-nat-equiv-def2*: *even* (*x*::*nat*) = (*EX y. x = Suc* (*Suc 0*) * *y*)
apply (*rule iffI*, *rule exI*)
apply (*erule even-nat-div-two-times-two* [*THEN sym*], *auto*)
done

lemma *odd-nat-equiv-def2*: *odd* (*x*::*nat*) = (*EX y. x = Suc*(*Suc* (*Suc 0*) * *y*))
apply (*rule iffI*, *rule exI*)
apply (*erule odd-nat-div-two-times-two-plus-one* [*THEN sym*], *auto*)
done

34.6 Parity and powers

lemma *minus-one-even-odd-power*:
(*even x* --> (− *1*::'*a*::{*comm-ring-1*,*recpower*}) ^*x* = *1*) &
(*odd x* --> (− *1*::'*a*) ^*x* = − *1*)
apply (*induct x*)
apply (*rule conjI*)
apply *simp*
apply (*insert even-nat-zero*, *blast*)
apply (*simp add*: *power-Suc*)
done

lemma *minus-one-even-power* [*simp*]:
even x ==> (− *1*::'*a*::{*comm-ring-1*,*recpower*}) ^*x* = *1*
by (*rule minus-one-even-odd-power* [*THEN conjunct1*, *THEN mp*], *assumption*)

lemma *minus-one-odd-power* [*simp*]:
odd x ==> (− *1*::'*a*::{*comm-ring-1*,*recpower*}) ^*x* = − *1*

by (*rule minus-one-even-odd-power* [*THEN conjunct2*, *THEN mp*], *assumption*)

lemma *neg-one-even-odd-power*:
(*even x* −−> (−*1*::'*a*::{*number-ring*,*recpower*}) ^*x* = *1*) &
(*odd x* −−> (−*1*::'*a*) ^*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*
by (*rule neg-one-even-odd-power* [*THEN conjunct1*, *THEN mp*], *assumption*)

lemma *neg-one-odd-power* [*simp*]:
odd x ==> (−*1*::'*a*::{*number-ring*,*recpower*}) ^*x* = −*1*
by (*rule neg-one-even-odd-power* [*THEN conjunct2*, *THEN mp*], *assumption*)

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

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 (*simp add*: *odd-nat-equiv-def2*)
apply (*erule exE*)
apply (*erule ssubst*)
apply (*subst power-Suc*)
apply (*subst power-add*)
apply (*subst zero-le-mult-iff*)
apply *auto*
apply (*subgoal-tac x* = *0* & *0* < *y*)
apply (*erule conjE*, *assumption*)
apply (*subst power-eq-0-iff* [*THEN sym*])
apply (*subgoal-tac 0* <= *x*^*y* * *x*^*y*)
apply *simp*
apply (*rule zero-le-square*)+
done

lemma *zero-le-power-eq*: (*0* <= (*x*::'*a*::{*recpower*,*ordered-idom*}) ^ *n*) =

```
    (even n | (odd n & 0 <= x))
  apply auto
  apply (subst zero-le-odd-power [THEN sym])
  apply assumption+
  apply (erule zero-le-even-power)
  apply (subst zero-le-odd-power)
  apply assumption+
done

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

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

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

lemma *zero-less-power-nat-eq*: *(0 < (x::nat) ^ n) = (n = 0 | 0 < x)*
apply (*induct n*)
apply *simp*
apply *auto*
done

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

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 =*
zero-le-power-eq [*of - number-of w, standard*]
declare *zero-le-power-eq-number-of* [*simp*]

lemmas *zero-less-power-eq-number-of =*
zero-less-power-eq [*of - number-of w, standard*]
declare *zero-less-power-eq-number-of* [*simp*]

lemmas *power-le-zero-eq-number-of =*
power-le-zero-eq [*of - number-of w, standard*]
declare *power-le-zero-eq-number-of* [*simp*]

lemmas *power-less-zero-eq-number-of =*
power-less-zero-eq [*of - number-of w, standard*]
declare *power-less-zero-eq-number-of* [*simp*]

lemmas *zero-less-power-nat-eq-number-of =*

zero-less-power-nat-eq [*of* - *number-of w*, *standard*]
declare *zero-less-power-nat-eq-number-of* [*simp*]

lemmas *power-eq-0-iff-number-of* = *power-eq-0-iff* [*of* - *number-of w*, *standard*]
declare *power-eq-0-iff-number-of* [*simp*]

lemmas *power-even-abs-number-of* = *power-even-abs* [*of number-of w* -, *standard*]
declare *power-even-abs-number-of* [*simp*]

34.7 An Equivalence for $0 \leq a\hat{}n$

lemma *even-power-le-0-imp-0*:
a ^ (*2***k*) ≤ (*0*::'*a*::{*ordered-idom*,*recpower*}) ==> *a*=*0*
apply (*induct k*)
apply (*auto simp add*: *zero-le-mult-iff mult-le-0-iff power-Suc*)
done

lemma *zero-le-power-iff*:
(*0* ≤ *a*^*n*) = (*0* ≤ (*a*::'*a*::{*ordered-idom*,*recpower*}) | *even n*)
(**is** *?P 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

34.8 Miscellaneous

lemma *even-plus-one-div-two*: *even* (*x*::*int*) ==> (*x* + *1*) *div 2* = *x div 2*
apply (*subst zdiv-zadd1-eq*)
apply (*simp add*: *even-def*)
done

lemma *odd-plus-one-div-two*: *odd* (*x*::*int*) ==> (*x* + *1*) *div 2* = *x div 2* + *1*
apply (*subst zdiv-zadd1-eq*)
apply (*simp add*: *even-def*)
done

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 *even-nat-plus-one-div-two*: *even* (*x*::*nat*) ==>
(*Suc x*) *div Suc* (*Suc 0*) = *x div Suc* (*Suc 0*)
apply (*subst div-Suc*)
apply (*simp add*: *even-nat-equiv-def*)
done

lemma *odd-nat-plus-one-div-two*: *odd* (*x*::*nat*) ==>
(*Suc x*) *div Suc* (*Suc 0*) = *Suc* (*x div Suc* (*Suc 0*))
apply (*subst div-Suc*)
apply (*simp add*: *odd-nat-equiv-def*)
done

end

35 GCD: The Greatest Common Divisor

theory *GCD*
imports *Parity*
begin

See [1].

consts
gcd :: *nat* × *nat* => *nat* — Euclid's algorithm

recdef *gcd* *measure* ((λ(*m*, *n*). *n*) :: *nat* × *nat* => *nat*)
gcd (*m*, *n*) = (*if n* = *0 then m else gcd* (*n*, *m mod n*))

constdefs
is-gcd :: *nat* => *nat* => *nat* => *bool* — *gcd* as a relation
is-gcd p m n == *p dvd m* ∧ *p dvd n* ∧
(∀ *d*. *d dvd m* ∧ *d dvd n* −−> *d dvd p*)

lemma *gcd-induct*:
(!!*m*. *P m* 0) ==>
(!!*m n*. 0 < *n* ==> *P n* (*m mod n*) ==> *P m n*)
==> *P* (*m*::*nat*) (*n*::*nat*)
apply (*induct m n rule*: *gcd.induct*)
apply (*case-tac n* = *0*)
apply *simp-all*
done

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

apply *simp*
done

lemma *gcd-non-0*: *0 < n ==> gcd (m, n) = gcd (n, m mod n)*
apply *simp*
done

declare *gcd.simps* [*simp del*]

lemma *gcd-1* [*simp*]: *gcd (m, Suc 0) = 1*
apply (*simp add*: *gcd-non-0*)
done

gcd (m, n) divides *m* and *n*. The conjunctions don't seem provable separately.

lemma *gcd-dvd1* [*iff*]: *gcd (m, n) dvd m*
and *gcd-dvd2* [*iff*]: *gcd (m, n) dvd n*
apply (*induct m n rule*: *gcd-induct*)
apply (*simp-all add*: *gcd-non-0*)
apply (*blast dest*: *dvd-mod-imp-dvd*)
done

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

lemma *gcd-greatest*: *k dvd m ==> k dvd n ==> k dvd gcd (m, n)*
apply (*induct m n rule*: *gcd-induct*)
apply (*simp-all add*: *gcd-non-0 dvd-mod*)
done

lemma *gcd-greatest-iff* [*iff*]: *(k dvd gcd (m, n)) = (k dvd m ∧ k dvd n)*
apply (*blast intro*!: *gcd-greatest intro*: *dvd-trans*)
done

lemma *gcd-zero*: *(gcd (m, n) = 0) = (m = 0 ∧ n = 0)*
by (*simp only*: *dvd-0-left-iff* [*THEN sym*] *gcd-greatest-iff*)

Function gcd yields the Greatest Common Divisor.

lemma *is-gcd*: *is-gcd (gcd (m, n)) m n*
apply (*simp add*: *is-gcd-def gcd-greatest*)
done

Uniqueness of GCDs.

lemma *is-gcd-unique*: *is-gcd m a b ==> is-gcd n a b ==> m = n*
apply (*simp add*: *is-gcd-def*)
apply (*blast intro*: *dvd-anti-sym*)
done

lemma *is-gcd-dvd*: *is-gcd m a b ==> k dvd a ==> k dvd b ==> k dvd m*
apply (*auto simp add*: *is-gcd-def*)
done

Commutativity

lemma *is-gcd-commute*: *is-gcd k m n = is-gcd k n m*
apply (*auto simp add*: *is-gcd-def*)
done

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-0-left* [*simp*]: *gcd* (*0*, *m*) = *m*
apply (*simp add*: *gcd-commute* [*of 0*])
done

lemma *gcd-1-left* [*simp*]: *gcd* (*Suc 0*, *m*) = *1*
apply (*simp add*: *gcd-commute* [*of Suc 0*])
done

Multiplication laws

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

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

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

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

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

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

Addition laws

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

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

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

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

end

36 Binomial: Binomial Coefficients

theory *Binomial*
imports *GCD*
begin

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

consts
  binomial :: *nat* ⇒ *nat* ⇒ *nat* (**infixl** *choose* 65)

primrec
  binomial-0: (*0* *choose k*) = (*if k* = *0 then 1 else 0*)

  binomial-Suc: (*Suc n choose k*) =
        (*if k* = *0 then 1 else* (*n choose* (*k* − *1*)) + (*n choose k*))

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

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

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

lemma *binomial-eq-0* [*rule-format*]: ∀ *k*. *n* < *k* −−> (*n choose k*) = *0*
apply (*induct n*, *auto*)
apply (*erule allE*)
apply (*erule mp*, *arith*)
done

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

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

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

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

lemma *zero-less-binomial* [*rule-format*]: *k* ≤ *n* −−> *0* < (*n choose k*)
by (*rule-tac m* = *n* **and** *n* = *k* **in** *diff-induct*, *simp-all*)

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

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

lemma *Suc-times-binomial-eq* [*rule-format*]:
$\forall k.\ k \leq n \longrightarrow Suc\ n * (n \text{ choose } k) = (Suc\ n \text{ choose } Suc\ k) * Suc\ k$
apply (*induct n*)
apply (*simp add*: *binomial-0*, *clarify*)
apply (*case-tac k*)
apply (*auto simp add*: *add-mult-distrib add-mult-distrib2 le-Suc-eq*
binomial-eq-0)
done

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

lemma *binomial-Suc-Suc-eq-times*:
$k \leq n \Longrightarrow (Suc\ n \text{ choose } Suc\ k) = (Suc\ n * (n \text{ choose } k))\ div\ Suc\ k$
by (*simp add*: *Suc-times-binomial-eq div-mult-self-is-m zero-less-Suc*
del: *mult-Suc mult-Suc-right*)

Another version, with -1 instead of Suc.

lemma *times-binomial-minus1-eq*:
$[|k \leq n;\ 0<k|] \Longrightarrow (n \text{ choose } k) * k = n * ((n - 1) \text{ choose } (k - 1))$
apply (*cut-tac* $n = n - 1$ **and** $k = k - 1$ **in** *Suc-times-binomial-eq*)
apply (*simp split add*: *nat-diff-split*, *auto*)
done

36.0.1 Theorems about *choose*

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

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

lemma *choose-deconstruct*: *finite M* $\Longrightarrow$ $x \notin M$
$\Longrightarrow$ $\{s.\ s <= insert\ x\ M\ \&\ card(s) = Suc\ k\}$
$= \{s.\ s <= M\ \&\ card(s) = Suc\ k\}\ Un$
$\{s.\ EX\ t.\ t <= M\ \&\ card(t) = k\ \&\ s = insert\ x\ t\}$
apply *safe*
apply (*auto intro*: *finite-subset* [*THEN card-insert-disjoint*])

apply (*drule-tac* $x = xa - \{x\}$ **in** *spec*)
apply (*subgoal-tac* $x \notin xa$, *auto*)
apply (*erule rev-mp*, *subst card-Diff-singleton*)
apply (*auto intro*: *finite-subset*)
done

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

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

finiteness of the two sets

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

Main theorem: combinatorial statement about number of subsets of a set.

lemma *n-sub-lemma*:
!!*A*. *finite A* ==> *card* $\{B.\ B <= A\ \&\ card\ B = k\} = (card\ A\ choose\ k)$
apply (*induct k*)
apply (*simp add*: *card-s-0-eq-empty*, *atomize*)
apply (*rotate-tac* -1, *erule finite-induct*)
apply (*simp-all* (*no-asm-simp*) *cong add*: *conj-cong*
add: *card-s-0-eq-empty choose-deconstruct*)
apply (*subst card-Un-disjoint*)
prefer *4* **apply** (*force simp add*: *constr-bij*)
prefer *3* **apply** *force*
prefer *2* **apply** (*blast intro*: *finite-Pow-iff* [*THEN iffD2*]
finite-subset [*of* - *Pow* (*insert x F*), *standard*])
apply (*blast intro*: *finite-Pow-iff* [*THEN iffD2*, *THEN* [*2*] *finite-subset*])
done

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

The binomial theorem (courtesy of Tobias Nipkow):

theorem *binomial*: $(a+b::nat)\hat{}n = (\sum k=0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k))$
proof (*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)

have *decomp*: $\{0..n+1\} = \{0\} \cup \{n+1\} \cup \{1..n\}$
by (*auto simp add*:*atLeastAtMost-def atLeast-def atMost-def*)
have *decomp2*: $\{0..n\} = \{0\} \cup \{1..n\}$
by (*auto simp add*:*atLeastAtMost-def atLeast-def atMost-def*)
have $(a+b::nat)\hat{}(n+1) = (a+b) * (\sum k=0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k))$
using *Suc* **by** *simp*
also have $\ldots = \ a*(\sum k=0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k)) +$
$b*(\sum k=0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k))$
by(*rule nat-distrib*)
also have $\ldots = (\sum k=0..n.\ (n\ choose\ k) * a\hat{}(k+1) * b\hat{}(n-k)) +$
$(\sum k=0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n-k+1))$
by(*simp add*: *setsum-mult mult-ac*)
also have $\ldots = (\sum k=0..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n+1-k)) +$
$(\sum k=1..n+1.\ (n\ choose\ (k\ -\ 1)) * a\hat{}k * b\hat{}(n+1-k))$
by (*simp add*:*setsum-shift-bounds-cl-Suc-ivl Suc-diff-le*
del:*setsum-cl-ivl-Suc*)
also have $\ldots = a\hat{}(n+1) + b\hat{}(n+1) +$
$(\sum k=1..n.\ (n\ choose\ (k\ -\ 1)) * a\hat{}k * b\hat{}(n+1-k)) +$
$(\sum k=1..n.\ (n\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
by(*simp add*: *decomp2*)
also have
$\ldots = a\hat{}(n+1) + b\hat{}(n+1) + (\sum k=1..n.\ (n+1\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
by(*simp add*: *nat-distrib setsum-addf binomial.simps*)
also have $\ldots = (\sum k=0..n+1.\ (n+1\ choose\ k) * a\hat{}k * b\hat{}(n+1-k))$
using *decomp* **by** *simp*
finally show *?case* **by** *simp*
qed

end

37 PreList: A Basis for Building the Theory of Lists

theory *PreList*
imports *Wellfounded-Relations Presburger Relation-Power Binomial*
begin

Is defined separately to serve as a basis for theory ToyList in the documentation.

end

38 List: The datatype of finite lists

theory *List*
imports *PreList*

begin

datatype *'a list* =
  Nil ([])
| *Cons 'a 'a list* (**infixr** # *65*)

38.1 Basic list processing functions

consts
  @ :: *'a list => 'a list => 'a list* (**infixr** *65*)
  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*
  list-all2 :: *('a => 'b => bool) => 'a list => 'b list => bool*
  map :: *('a=>'b) => ('a list => 'b list)*
  nth :: *'a list => nat => 'a* (**infixl** ! *100*)
  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*
  null:: *'a list => bool*
  distinct:: *'a list => bool*
  replicate :: *nat => 'a => 'a list*
  rotate1 :: *'a list ⇒ 'a list*
  rotate :: *nat ⇒ 'a list ⇒ 'a list*
  sublist :: *'a list => nat set => 'a list*

  mem :: *'a => 'a list => bool* (**infixl** *55*)
  list-inter :: *'a list ⇒ 'a list ⇒ 'a list*
  list-ex :: *('a ⇒ bool) ⇒ 'a list ⇒ bool*
  list-all:: *('a => bool) => ('a list => bool)*
  itrev :: *'a list ⇒ 'a list ⇒ 'a list*
  filtermap :: *('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list*
  map-filter :: *('a => 'b) => ('a => bool) => 'a list => 'b 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*)

upto:: *nat* => *nat* => *nat list* ((*1*[-../-]))

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*

[*i*..*j*] == [*i*..<(*Suc j*)]

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*.

syntax *length* :: *'a list* => *nat*
translations *length* => *size* :: - *list* => *nat*

typed-print-translation ⟪
let
fun size-tr' - (*Type* (*fun*, (*Type* (*list*, -) :: -))) [*t*] =
Syntax.const length $ *t*
| *size-tr'* - - - = *raise Match*;
in [(*size*, *size-tr'*)] *end*
⟫

primrec
hd(*x*#*xs*) = *x*

primrec
tl([]) = []
tl(*x*#*xs*) = *xs*

primrec
null([]) = *True*
null(*x*#*xs*) = *False*

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-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
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-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
replicate-0: *replicate 0 x = []*
replicate-Suc: *replicate (Suc n) x = x # replicate n x*

defs
rotate1-def: *rotate1 xs == (case xs of [] ⇒ [] | x#xs ⇒ xs @ [x])*
rotate-def: *rotate n == rotate1 ^ n*

list-all2-def:
list-all2 P xs ys ==
length xs = length ys ∧ (∀(x, y) ∈ set (zip xs ys). P x y)

sublist-def:
sublist xs A == map fst (filter (%p. snd p : A) (zip xs [0..<size xs]))

primrec
x mem [] = False
x mem (y#ys) = (if y=x then True else x mem ys)

primrec
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 P [] = True
list-all P (x#xs) = (P(x) ∧ list-all P xs)

primrec
list-ex P [] = False
list-ex P (x#xs) = (P x ∨ list-ex P xs)

primrec
filtermap f [] = []
filtermap f (x#xs) =
(case f x of None ⇒ filtermap f xs
| Some y ⇒ y # (filtermap f xs))

primrec
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
itrev [] ys = ys
itrev (x#xs) ys = itrev xs (x#ys)

lemma *not-Cons-self* [*simp*]: *xs ≠ x # xs*

by (*induct xs*) *auto*

lemmas *not-Cons-self2* [*simp*] = *not-Cons-self* [*symmetric*]

lemma *neq-Nil-conv*: (*xs* ≠ []) = (∃ *y ys*. *xs* = *y* # *ys*)
by (*induct xs*) *auto*

lemma *length-induct*:
(!!*xs*. ∀ *ys*. *length ys* < *length xs* −−> *P ys* ==> *P xs*) ==> *P xs*
by (*rule measure-induct* [*of length*]) *iprover*

38.1.1 *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-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* [*rule-format*]:
length xs <= *length ys* −−> *xs* = *x* # *ys* = *False*
apply (*induct xs*, *auto*)
done

lemma *list-induct2*[*consumes 1*]: ⋀*ys*.
⟦ *length xs* = *length ys*;

P [] []; ⋀*x xs y ys*. ⟦ *length xs* = *length ys*; *P xs ys* ⟧ ⟹ *P* (*x*#*xs*) (*y*#*ys*) ⟧ ⟹ *P xs ys*
apply(*induct xs*)
apply *simp*
apply(*case-tac ys*)
apply *simp*
apply(*simp*)
done

38.1.2 @ − 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*]:
!!*ys*. *length xs* = *length ys* ∨ *length us* = *length vs*
==> (*xs*@*us* = *ys*@*vs*) = (*xs*=*ys* ∧ *us*=*vs*)
apply (*induct xs*)
apply (*case-tac ys*, *simp*, *force*)
apply (*case-tac ys*, *force*, *simp*)
done

lemma *append-eq-append-conv2*: !!*ys zs ts*.
(*xs* @ *ys* = *zs* @ *ts*) =
(*EX us*. *xs* = *zs* @ *us* & *us* @ *ys* = *ts* | *xs* @ *us* = *zs* & *ys* = *us*@ *ts*)
apply (*induct xs*)
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*]: $(xs \mathbin{@} [x] = ys \mathbin{@} [y]) = (xs = ys \wedge x = y)$
by *simp*

lemma *append-same-eq* [*iff*]: $(ys \mathbin{@} xs = zs \mathbin{@} xs) = (ys = zs)$
by *simp*

lemma *append-self-conv2* [*iff*]: $(xs \mathbin{@} ys = ys) = (xs = [])$
using *append-same-eq* [*of* - - []] **by** *auto*

lemma *self-append-conv2* [*iff*]: $(ys = xs \mathbin{@} ys) = (xs = [])$
using *append-same-eq* [*of* []] **by** *auto*

lemma *hd-Cons-tl* [*simp*]: $xs \neq [] \Longrightarrow hd\ xs \# tl\ xs = xs$
by (*induct xs*) *auto*

lemma *hd-append*: $hd\ (xs \mathbin{@} ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$
by (*induct xs*) *auto*

lemma *hd-append2* [*simp*]: $xs \neq [] \Longrightarrow hd\ (xs \mathbin{@} ys) = hd\ xs$
by (*simp add*: *hd-append split*: *list.split*)

lemma *tl-append*: $tl\ (xs \mathbin{@} ys) = (case\ xs\ of\ [] \Rightarrow tl\ ys \mid z\#zs \Rightarrow zs \mathbin{@} ys)$
by (*simp split*: *list.split*)

lemma *tl-append2* [*simp*]: $xs \neq [] \Longrightarrow tl\ (xs \mathbin{@} ys) = tl\ xs \mathbin{@} ys$
by (*simp add*: *tl-append split*: *list.split*)

lemma *Cons-eq-append-conv*: $x\#xs = ys@zs =$
$(ys = [] \ \&\ x\#xs = zs \mid (EX\ ys'.\ x\#ys' = ys \ \&\ xs = ys'@zs))$
by(*cases ys*) *auto*

lemma *append-eq-Cons-conv*: $(ys@zs = x\#xs) =$
$(ys = [] \ \&\ zs = x\#xs \mid (EX\ ys'.\ ys = x\#ys' \ \&\ ys'@zs = xs))$
by(*cases ys*) *auto*

Trivial rules for solving @-equations automatically.

lemma *eq-Nil-appendI*: $xs = ys \Longrightarrow xs = [] \mathbin{@} ys$
by *simp*

lemma *Cons-eq-appendI*:
$[|\ x \# xs1 = ys;\ xs = xs1 \mathbin{@} zs\ |] \Longrightarrow x \# xs = ys \mathbin{@} zs$
by (*drule sym*) *simp*

lemma *append-eq-appendI*:
$[|\ xs \mathbin{@} xs1 = zs;\ ys = xs1 \mathbin{@} us\ |] \Longrightarrow xs \mathbin{@} ys = zs \mathbin{@} us$
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-setup ⟪
local

```
val append-assoc = thm append-assoc;
val append-Nil = thm append-Nil;
val append-Cons = thm append-Cons;
val append1-eq-conv = thm append1-eq-conv;
val append-same-eq = thm append-same-eq;

fun last (cons as Const(List.list.Cons,-) $ - $ xs) =
  (case xs of Const(List.list.Nil,-) => cons | - => last xs)
  | last (Const(List.op @,-) $ - $ ys) = last ys
  | last t = t;

fun list1 (Const(List.list.Cons,-) $ - $ Const(List.list.Nil,-)) = true
  | list1 - = false;

fun butlast ((cons as Const(List.list.Cons,-) $ x) $ xs) =
  (case xs of Const(List.list.Nil,-) => xs | - => cons $ butlast xs)
  | butlast ((app as Const(List.op @,-) $ xs) $ ys) = app $ butlast ys
  | butlast xs = Const(List.list.Nil,fastype-of xs);

val rearr-ss = HOL-basic-ss addsimps [append-assoc, append-Nil, append-Cons];

fun list-eq sg 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(List.op @,appT)
        val F2 = eq $ (app$lhs1$lastl) $ (app$rhs1$lastr)
        val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (F,F2));
        val thm = Tactic.prove sg [] [] eq
          (K (simp-tac (Simplifier.inherit-bounds ss rearr-ss) 1));
      in SOME ((conv RS (thm RS trans)) RS eq-reflection) end;

  in
    if list1 lastl andalso list1 lastr then rearr append1-eq-conv
    else if lastl aconv lastr then rearr append-same-eq
    else NONE
  end;

in
```

val list-eq-simproc =
Simplifier.simproc (Theory.sign-of (the-context ())) list-eq [(xs::'a list) = ys] list-eq;

end;

Addsimprocs [list-eq-simproc];
⟩⟩

38.1.3 *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* [*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*[*iff*]:
(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*[*iff*]:
(x#xs = map f ys) = (∃z zs. ys = z#zs ∧ x = f z ∧ xs = map f zs)
by (*cases ys*) *auto*

lemma *ex-map-conv*:
(*EX xs. ys = map f xs*) = (*ALL y : set ys. EX x. y = f x*)
by(*induct ys*, *auto*)

lemma *map-eq-imp-length-eq*:
!!*xs. map f xs = map f ys ==> length xs = length ys*
apply (*induct ys*)
apply *simp*
apply(*simp* (*no-asm-use*))
apply *clarify*
apply(*simp* (*no-asm-use*))
apply *fast*
done

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*:
!!*xs. map f xs = map f ys ==> inj f ==> xs = ys*
by (*induct ys*) (*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* $(\bigcup(set\ `\ A)) \Longrightarrow$ *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*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = x) \Longrightarrow map\ f\ xs = xs$
by (*induct xs*, *auto*)

lemma *map-fun-upd* [*simp*]: $y \notin set\ xs \Longrightarrow map\ (f(y{:=}v))\ xs = map\ f\ xs$
by (*induct xs*) *auto*

lemma *map-fst-zip*[*simp*]:
length xs = *length ys* $\Longrightarrow$ *map fst* (*zip xs ys*) = *xs*
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *map-snd-zip*[*simp*]:
length xs = *length ys* $\Longrightarrow$ *map snd* (*zip xs ys*) = *ys*
by (*induct rule*:*list-induct2*, *simp-all*)

38.1.4 *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*]: !!*ys*. (*rev xs* = *rev ys*) = (*xs* = *ys*)
apply (*induct xs*, *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

ML ⟪ *val rev-induct-tac* = *induct-thm-tac* (*thm rev-induct*) ⟫— compatibility

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*

38.1.5 *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*: *l* = *x*#*xs* ⟹ *x*∈*set l*
by (*case-tac l*, *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 *in-set-conv-decomp*: $(x : set\ xs) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs)$
proof (*induct xs*)
  case *Nil* **show** *?case* **by** *simp*
  case (*Cons a xs*)
  show *?case*
  proof
    assume $x \in set\ (a\ \#\ xs)$
    with *prems* **show** $\exists\, ys\ zs.\ a\ \#\ xs = ys\ @\ x\ \#\ zs$
      by (*simp*, *blast intro*: *Cons-eq-appendI*)
  next
    assume $\exists\, ys\ zs.\ a\ \#\ xs = ys\ @\ x\ \#\ zs$
    then obtain *ys zs* **where** *eq*: $a\ \#\ xs = ys\ @\ x\ \#\ zs$ **by** *blast*
    show $x \in set\ (a\ \#\ xs)$
      by (*cases ys*, *auto simp add*: *eq*)
  qed
qed

lemma *finite-list*: *finite A* ==> *EX l. set l = A*
apply (*erule finite-induct*, *auto*)
apply (*rule-tac x=x#l* **in** *exI*, *auto*)
done

lemma *card-length*: $card\ (set\ xs) \leq length\ xs$
by (*induct xs*) (*auto simp add*: *card-insert-if*)

38.1.6 *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*]: $filter\ P\ (filter\ Q\ xs) = filter\ (\lambda x.\ Q\ x \wedge P\ x)\ xs$
by (*induct xs*) *auto*

lemma *length-filter-le* [*simp*]: $length\ (filter\ P\ xs) \leq length\ xs$
by (*induct xs*) (*auto simp add*: *le-SucI*)

lemma *filter-True* [*simp*]: $\forall\, x \in set\ xs.\ P\ x$ ==> $filter\ P\ xs = xs$
by (*induct xs*) *auto*

lemma *filter-False* [*simp*]: $\forall\, x \in set\ xs.\ \neg\ P\ x$ ==> $filter\ P\ xs = []$
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 add*: *nth-Cons image-def split*:*nat.split elim*:*lessE*)
have *length* (*filter p* (*x* # *xs*)) = *Suc*(*card ?S*)
using *Cons* **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: nth-Cons image-def split:nat.split elim:lessE)
    have length (filter p (x # xs)) = card ?S
      using Cons 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
  (concl 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 xy: x = y
      show ?thesis
      proof from Py xy Cons(2) show ?Q [] by simp qed
    next
      assume x ≠ y with Py Cons(2) show ?thesis by simp
    qed
  next
    assume Py: ¬ P y
    with Cons obtain us vs where 1 : ?P (y#ys) (y#us) vs by fastsimp
    show ?thesis (is ? us. ?Q us)
    proof show ?Q (y#us) using 1 by simp qed
  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*:
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*

38.1.7 *concat*

lemma *concat-append* [*simp*]: *concat* (*xs* @ *ys*) = *concat xs* @ *concat ys*
by (*induct xs*) *auto*

lemma *concat-eq-Nil-conv* [*iff*]: (*concat xss* = []) = (∀ *xs* ∈ *set xss*. *xs* = [])
by (*induct xss*) *auto*

lemma *Nil-eq-concat-conv* [*iff*]: ([] = *concat xss*) = (∀ *xs* ∈ *set xss*. *xs* = [])
by (*induct xss*) *auto*

lemma *set-concat* [*simp*]: *set* (*concat xs*) = ⋃(*set* ‘ *set 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*

38.1.8 *nth*

lemma *nth-Cons-0* [*simp*]: (*x* # *xs*)!*0* = *x*
by *auto*

lemma *nth-Cons-Suc* [*simp*]: (*x* # *xs*)!(*Suc n*) = *xs*!*n*
by *auto*

declare *nth.simps* [*simp del*]

lemma *nth-append*:

!!n. $(xs$ @ $ys)!n = (if\ n < length\ xs\ then\ xs!n\ else\ ys!(n - length\ xs))$
apply (*induct xs*, *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. $n < length\ xs$ ==> $(map\ f\ xs)!n = f(xs!n)$
apply (*induct xs*, *simp*)
apply (*case-tac n*, *auto*)
done

lemma *set-conv-nth*: $set\ xs = \{xs!i \mid i.\ i < length\ xs\}$
apply (*induct xs*, *simp*, *simp*)
apply *safe*
apply (*rule-tac x = 0* **in** *exI*, *simp*)
apply (*rule-tac x = Suc i* **in** *exI*, *simp*)
apply (*case-tac i*, *simp*)
apply (*rename-tac j*)
apply (*rule-tac x = j* **in** *exI*, *simp*)
done

lemma *in-set-conv-nth*: $(x \in set\ xs) = (\exists i < length\ xs.\ xs!i = x)$
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*:
$(\forall x \in set\ xs.\ P\ x) = (\forall i.\ i < length\ xs$ --> $P\ (xs\ !\ i))$
by (*auto simp add*: *set-conv-nth*)

38.1.9 *list-update*

lemma *length-list-update* [*simp*]: !!i. $length(xs[i:=x]) = length\ xs$
by (*induct xs*) (*auto split*: *nat.split*)

lemma *nth-list-update*:

!!$i\ j.\ i < length\ xs ==> (xs[i:=x])!j = (if\ i = j\ then\ x\ else\ xs!j)$
by (*induct xs*) (*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*]: !!$i\ j.\ i \neq j ==> xs[i:=x]!j = xs!j$
by (*induct xs*) (*auto simp add*: *nth-Cons split*: *nat.split*)

lemma *list-update-overwrite* [*simp*]:
!!$i.\ i < size\ xs ==> xs[i:=x,\ i:=y] = xs[i:=y]$
by (*induct xs*) (*auto split*: *nat.split*)

lemma *list-update-id*[*simp*]: !!$i.\ i < length\ xs ==> xs[i := xs!i] = xs$
apply (*induct xs*, *simp*)
apply(*simp split*:*nat.splits*)
done

lemma *list-update-beyond*[*simp*]: $\bigwedge i.\ length\ xs \leq i \Longrightarrow xs[i:=x] = xs$
apply (*induct xs*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
done

lemma *list-update-same-conv*:
!!$i.\ i < length\ xs ==> (xs[i := x] = xs) = (xs!i = x)$
by (*induct xs*) (*auto split*: *nat.split*)

lemma *list-update-append1*:
!!$i.\ i < size\ xs \Longrightarrow (xs\ @\ ys)[i:=x] = xs[i:=x]\ @\ ys$
apply (*induct xs*, *simp*)
apply(*simp split*:*nat.split*)
done

lemma *list-update-append*:
!!$n.\ (xs\ @\ ys)\ [n:=\ x] =$
$(if\ n < length\ xs\ then\ xs[n:=\ x]\ @\ ys\ else\ xs\ @\ (ys\ [n-length\ xs:=\ x]))$
by (*induct xs*) (*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*:
!!$i\ xy\ xs.\ length\ xs = length\ ys ==>$
$(zip\ xs\ ys)[i:=xy] = zip\ (xs[i:=fst\ xy])\ (ys[i:=snd\ xy])$
by (*induct ys*) (*auto*, *case-tac xs*, *auto split*: *nat.split*)

lemma *set-update-subset-insert*: !!i. $set(xs[i:=x]) <= insert\ x\ (set\ xs)$
by (*induct xs*) (*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. $n < length\ xs \Longrightarrow x \in set\ (xs[n := x])$
by (*induct xs*) (*auto split*:*nat.splits*)

38.1.10 *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 *length-butlast* [*simp*]: $length\ (butlast\ xs) = length\ xs - 1$
by (*induct xs rule*: *rev-induct*) *auto*

lemma *butlast-append*:
!!ys. $butlast\ (xs\ @\ ys) = (if\ ys = []\ then\ butlast\ xs\ else\ xs\ @\ butlast\ ys)$
by (*induct xs*) *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*. *n* < *length xs* ⟹ *last* (*drop n xs*) = *last xs*
apply (*induct xs*)
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*)

38.1.11 *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-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 *drop-tl*: !!*n*. *drop n* (*tl xs*) = *tl*(*drop n xs*)
by(*induct xs*, *simp-all add*:*drop-Cons drop-Suc split*:*nat.split*)

lemma *nth-via-drop*: !!*n*. *drop n xs* = *y*#*ys* ⟹ *xs*!*n* = *y*
apply (*induct xs*, *simp*)
apply(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)
done

lemma *take-Suc-conv-app-nth*:
!!*i*. *i* < *length xs* ⟹ *take* (*Suc i*) *xs* = *take i xs* @ [*xs*!*i*]
apply (*induct xs*, *simp*)
apply (*case-tac i*, *auto*)
done

lemma *drop-Suc-conv-tl*:
!!*i*. *i* < *length xs* ⟹ (*xs*!*i*) # (*drop* (*Suc i*) *xs*) = *drop i xs*

apply (*induct xs*, *simp*)
apply (*case-tac i*, *auto*)
done

lemma *length-take* [*simp*]: !!*xs*. *length* (*take n xs*) = *min* (*length xs*) *n*
by (*induct n*) (*auto*, *case-tac xs*, *auto*)

lemma *length-drop* [*simp*]: !!*xs*. *length* (*drop n xs*) = (*length xs* − *n*)
by (*induct n*) (*auto*, *case-tac xs*, *auto*)

lemma *take-all* [*simp*]: !!*xs*. *length xs* <= *n* ==> *take n xs* = *xs*
by (*induct n*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-all* [*simp*]: !!*xs*. *length xs* <= *n* ==> *drop n xs* = []
by (*induct n*) (*auto*, *case-tac xs*, *auto*)

lemma *take-append* [*simp*]:
!!*xs*. *take n* (*xs* @ *ys*) = (*take n xs* @ *take* (*n* − *length xs*) *ys*)
by (*induct n*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-append* [*simp*]:
!!*xs*. *drop n* (*xs* @ *ys*) = *drop n xs* @ *drop* (*n* − *length xs*) *ys*
by (*induct n*) (*auto*, *case-tac xs*, *auto*)

lemma *take-take* [*simp*]: !!*xs n*. *take n* (*take m xs*) = *take* (*min n m*) *xs*
apply (*induct m*, *auto*)
apply (*case-tac xs*, *auto*)
apply (*case-tac n*, *auto*)
done

lemma *drop-drop* [*simp*]: !!*xs*. *drop n* (*drop m xs*) = *drop* (*n* + *m*) *xs*
apply (*induct m*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *take-drop*: !!*xs n*. *take n* (*drop m xs*) = *drop m* (*take* (*n* + *m*) *xs*)
apply (*induct m*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *drop-take*: !!*m n*. *drop n* (*take m xs*) = *take* (*m*−*n*) (*drop n xs*)
apply(*induct xs*)
apply *simp*
apply(*simp add*: *take-Cons drop-Cons split*:*nat.split*)
done

lemma *append-take-drop-id* [*simp*]: !!*xs*. *take n xs* @ *drop n xs* = *xs*
apply (*induct n*, *auto*)
apply (*case-tac xs*, *auto*)

done

lemma *take-eq-Nil*[*simp*]: !!*n*. (*take n xs* = []) = (*n* = *0* ∨ *xs* = [])
apply(*induct xs*)
apply *simp*
apply(*simp add*:*take-Cons split*:*nat.split*)
done

lemma *drop-eq-Nil*[*simp*]: !!*n*. (*drop n xs* = []) = (*length xs* <= *n*)
apply(*induct xs*)
apply *simp*
apply(*simp add*:*drop-Cons split*:*nat.split*)
done

lemma *take-map*: !!*xs*. *take n* (*map f xs*) = *map f* (*take n xs*)
apply (*induct n*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *drop-map*: !!*xs*. *drop n* (*map f xs*) = *map f* (*drop n xs*)
apply (*induct n*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *rev-take*: !!*i*. *rev* (*take i xs*) = *drop* (*length xs* − *i*) (*rev xs*)
apply (*induct xs*, *auto*)
apply (*case-tac i*, *auto*)
done

lemma *rev-drop*: !!*i*. *rev* (*drop i xs*) = *take* (*length xs* − *i*) (*rev xs*)
apply (*induct xs*, *auto*)
apply (*case-tac i*, *auto*)
done

lemma *nth-take* [*simp*]: !!*n i*. *i* < *n* ==> (*take n xs*)!*i* = *xs*!*i*
apply (*induct xs*, *auto*)
apply (*case-tac n*, *blast*)
apply (*case-tac i*, *auto*)
done

lemma *nth-drop* [*simp*]:
!!*xs i*. *n* + *i* <= *length xs* ==> (*drop n xs*)!*i* = *xs*!(*n* + *i*)
apply (*induct n*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *set-take-subset*: ⋀*n*. *set*(*take n xs*) ⊆ *set xs*
by(*induct xs*)(*auto simp*:*take-Cons split*:*nat.split*)

lemma *set-drop-subset*: $\bigwedge n.\ set(drop\ n\ xs) \subseteq set\ xs$
by(*induct xs*)(*auto simp*:*drop-Cons split*:*nat.split*)

lemma *in-set-takeD*: $x : set(take\ n\ xs) \Longrightarrow x : set\ xs$
using *set-take-subset* **by** *fast*

lemma *in-set-dropD*: $x : set(drop\ n\ xs) \Longrightarrow x : set\ xs$
using *set-drop-subset* **by** *fast*

lemma *append-eq-conv-conj*:
$!!zs.\ (xs\ @\ ys = zs) = (xs = take\ (length\ xs)\ zs \wedge ys = drop\ (length\ xs)\ zs)$
apply (*induct xs*, *simp*, *clarsimp*)
apply (*case-tac zs*, *auto*)
done

lemma *take-add* [*rule-format*]:
$\forall i.\ i+j \leq length(xs) \longrightarrow take\ (i+j)\ xs = take\ i\ xs\ @\ take\ j\ (drop\ i\ xs)$
apply (*induct xs*, *auto*)
apply (*case-tac i*, *simp-all*)
done

lemma *append-eq-append-conv-if*:
$!!\ ys_1.\ (xs_1\ @\ xs_2 = ys_1\ @\ ys_2) =$
(*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)
apply *simp*
apply(*case-tac* ys_1)
apply *simp-all*
done

lemma *take-hd-drop*:
$!!n.\ n < length\ xs \Longrightarrow take\ n\ xs\ @\ [hd\ (drop\ n\ xs)] = take\ (n+1)\ xs$
apply(*induct xs*)
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* ⟹ *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

38.1.12 *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-take-whileD*: *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

38.1.13 *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*]:
!!*xs*. *length* (*zip xs ys*) = *min* (*length xs*) (*length ys*)
apply (*induct ys*, *simp*)
apply (*case-tac xs*, *auto*)
done

lemma *zip-append1*:
!!*xs*. *zip* (*xs* @ *ys*) *zs* =
zip xs (*take* (*length xs*) *zs*) @ *zip ys* (*drop* (*length xs*) *zs*)
apply (*induct zs*, *simp*)
apply (*case-tac xs*, *simp-all*)
done

lemma *zip-append2*:
!!*ys*. *zip xs* (*ys* @ *zs*) =
zip (*take* (*length ys*) *xs*) *ys* @ *zip* (*drop* (*length ys*) *xs*) *zs*
apply (*induct xs*, *simp*)
apply (*case-tac ys*, *simp-all*)
done

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 *nth-zip* [*simp*]:
!!*i xs*. [| *i* < *length xs*; *i* < *length ys*|] ==> (*zip xs ys*)!*i* = (*xs*!*i*, *ys*!*i*)
apply (*induct ys*, *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*]:
!!*j*. *zip* (*replicate i x*) (*replicate j y*) = *replicate* (*min i j*) (*x*,*y*)
apply (*induct i*, *auto*)
apply (*case-tac j*, *auto*)
done

38.1.14 *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*]: *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?*]:
⋀*n ys*. *list-all2 P xs ys* ⟹ *list-all2 P* (*take n xs*) (*take n ys*)
apply (*induct xs*)
apply *simp*
apply (*clarsimp simp add*: *list-all2-Cons1*)
apply (*case-tac n*)
apply *auto*
done

lemma *list-all2-dropI* [*simp*,*intro?*]:
⋀*n bs*. *list-all2 P as bs* ⟹ *list-all2 P* (*drop n as*) (*drop n bs*)
apply (*induct as*, *simp*)
apply (*clarsimp simp add*: *list-all2-Cons1*)
apply (*case-tac n*, *simp*, *simp*)
done

lemma *list-all2-mono* [*intro?*]:
⋀*y*. *list-all2 P x y* ⟹ (⋀*x y*. *P x y* ⟹ *Q x y*) ⟹ *list-all2 Q x y*
apply (*induct x*, *simp*)
apply (*case-tac y*, *auto*)
done

38.1.15 *foldl* and *foldr*

lemma *foldl-append* [*simp*]:
!!*a*. *foldl f a* (*xs* @ *ys*) = *foldl f* (*foldl f a xs*) *ys*
by (*induct xs*) *auto*

lemma *foldr-append*[*simp*]: *foldr f* (*xs* @ *ys*) *a* = *foldr f xs* (*foldr f ys a*)
by (*induct xs*) *auto*

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*])

Note: *n* ≤ *foldl* (*op* +) *n ns* looks simpler, but is more difficult to use because it requires an additional transitivity step.

lemma *start-le-sum*: !!*n*::*nat*. *m* <= *n* ==> *m* <= *foldl* (*op* +) *n ns*
by (*induct ns*) *auto*

lemma *elem-le-sum*: !!*n*::*nat*. *n* : *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*]:
!!*m*::*nat*. (*foldl* (*op* +) *m ns* = *0*) = (*m* = *0* ∧ (∀ *n* ∈ *set ns*. *n* = *0*))
by (*induct ns*) *auto*

38.1.16 *upto*

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* ∨ *j* <= *i*)
by(*induct j*)*simp-all*

lemma *upt-eq-Cons-conv*:
!!*x xs*. ([*i*..<*j*] = *x*#*xs*) = (*i* < *j* & *i* = *x* & [*i*+*1*..<*j*] = *xs*)
apply(*induct j*)
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*]
apply(*rule trans*)
apply(*subst upt-rec*)
prefer *2* **apply** (*rule refl*, *simp*)
done

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 *take-upt* [*simp*]: !!*i*. *i*+*m* <= *n* ==> *take m* [*i*..<*n*] = [*i*..<*i*+*m*]
apply (*induct m*, *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*
apply *arith*
done

lemma *map-Suc-upt*: *map Suc* [*m*..<*n*] = [*Suc m*..*n*]
by (*induct n*) *auto*

lemma *nth-map-upt*: !!*i*. *i* < *n*−*m* ==> (*map f* [*m*..<*n*]) ! *i* = *f*(*m*+*i*)
apply (*induct n m 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*:
!!*xs ys*. *k* <= *length xs* ==> *k* <= *length ys* ==>
(!!*i*. *i* < *k* −−> *xs*!*i* = *ys*!*i*) ==> *take k xs* = *take k ys*
apply (*atomize*, *induct k*)
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 *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*: $(\forall i.\ take\ i\ xs = take\ i\ ys) ==> xs = ys$
— 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 [*simp*] = *take-Cons'*[*of number-of v*,*standard*]
drop-Cons'[*of number-of v*,*standard*]
nth-Cons'[*of - - number-of v*,*standard*]

38.1.17 *distinct* and *remdups*

lemma *distinct-append* [*simp*]:
$distinct\ (xs\ @\ ys) = (distinct\ xs \wedge distinct\ ys \wedge set\ xs \cap set\ ys = \{\})$
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 *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-filter [simp]: distinct xs ==> distinct (filter P xs)
by (induct xs) auto

lemma distinct-map-filterI:
 distinct(map f xs) ⟹ distinct(map f (filter P xs))
apply(induct xs)
 apply simp
apply force
done

lemma distinct-upt[simp]: distinct[i..<j]
by (induct j) auto

lemma distinct-take[simp]: ⋀i. distinct xs ⟹ distinct (take i xs)
apply(induct xs)
 apply simp
apply (case-tac i)
 apply simp-all
apply(blast dest:in-set-takeD)
done

lemma distinct-drop[simp]: ⋀i. distinct xs ⟹ distinct (drop i xs)
apply(induct xs)
 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 *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 *inj-on-setI*: *distinct*(*map f xs*) ==> *inj-on f* (*set xs*)
apply(*induct xs*)
apply *simp*
apply *fastsimp*
done

lemma *inj-on-set-conv*:
distinct xs ⟹ *inj-on f* (*set xs*) = *distinct*(*map f xs*)

apply(*induct xs*)
apply *simp*
apply *fastsimp*
done

38.1.18 *remove1*

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 *notin-set-remove1*[*simp*]: *x* ~: *set xs* ==> *x* ~: *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*

38.1.19 *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 ≠ 0 ==> tl* (*replicate n x*) = *replicate* (*n − 1*) *x*
by (*induct n*) *auto*

lemma *last-replicate* [*simp*]: *n ≠ 0 ==> last* (*replicate n x*) = *x*
by (*atomize* (*full*), *induct n*) *auto*

lemma *nth-replicate*[*simp*]: !!*i*. *i < n ==>* (*replicate n x*)!*i* = *x*
apply (*induct n*, *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 ≤ 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*]: !!*i*. *drop i* (*replicate k x*) = *replicate* (*k−i*) *x*
apply (*induct k*)
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 ≠ 0 ==> 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*) ==> *x* = *y*
by (*simp add*: *set-replicate-conv-if split*: *split-if-asm*)

38.1.20 *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-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*]: !!*xs*. *length*(*rotate n xs*) = *length xs*
by (*induct n*) (*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

38.1.21 *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*:
!!*is*. *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*)
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*: !!*I*. *set*(*sublist xs I*) = {*xs*!*i*|*i*. *i*<*size xs* ∧ *i* ∈ *I*}
apply(*induct xs*)
apply *simp*
apply(*auto simp add*:*sublist-Cons nth-Cons split*:*nat.split elim*: *lessE*)
apply(*erule lessE*)
apply *auto*
apply(*erule lessE*)
apply *auto*
done

lemma *set-sublist-subset*: *set*(*sublist xs I*) ⊆ *set xs*
by(*auto simp add*:*set-sublist*)

lemma *notin-set-sublistI*[*simp*]: *x* ∉ *set xs* ⟹ *x* ∉ *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*]: !!*I*. *distinct xs* ⟹ *distinct*(*sublist xs I*)
apply(*induct xs*)
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*: ⋀*s*. *distinct xs* ⟹
filter (%*x*. *x* ∈ *set*(*sublist xs s*)) *xs* = *sublist xs s*
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a xs*)
moreover hence !*x*. *x*: *set xs* ⟶ *x* ≠ *a* **by** *auto*
ultimately show *?case* **by**(*simp add*: *sublist-Cons cong*:*filter-cong*)
qed

38.1.22 Sets of Lists

38.1.23 *lists*: the list-forming operator over sets

consts *lists* :: *'a set* => *'a list set*
inductive *lists A*
intros
Nil [*intro*!]: []: *lists A*
Cons [*intro*!]: [| *a*: *A*;*l*: *lists A*|] ==> *a*#*l* : *lists A*

inductive-cases *listsE* [*elim*!]: *x*#*l* : *lists A*

lemma *lists-mono* [*mono*]: *A* ⊆ *B* ==> *lists A* ⊆ *lists B*
by (*unfold lists.defs*) (*blast intro*!: *lfp-mono*)

lemma *lists-IntI*:
assumes *l*: *l*: *lists A* **shows** *l*: *lists B* ==> *l*: *lists* (*A Int B*) **using** *l*
by *induct blast*+

lemma *lists-Int-eq* [*simp*]: *lists* $(A \cap B)$ = *lists* $A \cap$ *lists* B
proof (*rule mono-Int* [*THEN equalityI*])
  show *mono lists* **by** (*simp add*: *mono-def lists-mono*)
  show *lists* $A \cap$ *lists* $B \subseteq$ *lists* $(A \cap B)$ **by** (*blast intro*: *lists-IntI*)
qed

lemma *append-in-lists-conv* [*iff*]:
    $(xs$ @ ys : *lists* $A) = (xs$: *lists* $A \wedge ys$: *lists* $A)$
by (*induct xs*) *auto*

lemma *in-lists-conv-set*: $(xs$: *lists* $A) = (\forall x \in$ *set* $xs.\ x$: $A)$
— eliminate *lists* in favour of *set*
by (*induct xs*) *auto*

lemma *in-listsD* [*dest*!]: $xs \in$ *lists* $A \Longrightarrow \forall x \in$*set* $xs.\ x \in A$
by (*rule in-lists-conv-set* [*THEN iffD1*])

lemma *in-listsI* [*intro*!]: $\forall x \in$*set* $xs.\ x \in A \Longrightarrow xs \in$ *lists* A
by (*rule in-lists-conv-set* [*THEN iffD2*])

lemma *lists-UNIV* [*simp*]: *lists UNIV* = *UNIV*
by *auto*

38.1.24 For efficiency

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.

The functions *itrev*, *filtermap* and *map-filter* are just there to generate efficient code. Do not use them for modelling and proving.

lemma *mem-iff*: $(x$ *mem* $xs) = (x$: *set* $xs)$
by (*induct xs*) *auto*

lemma *list-inter-conv*: *set*(*list-inter xs ys*) = *set xs* $\cap$ *set ys*
by (*induct xs*) *auto*

lemma *list-all-iff*: *list-all P xs* = $(\forall x \in$ *set* $xs.\ P\ x)$
by (*induct xs*) *auto*

lemma *list-all-append* [*simp*]:
list-all P $(xs$ @ $ys)$ = (*list-all P xs* $\wedge$ *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-ex-iff*: *list-ex P xs* = (∃ *x* ∈ *set xs*. *P x*)
by (*induct xs*) *simp-all*

lemma *itrev*[*simp*]: *ALL ys*. *itrev xs ys* = *rev xs* @ *ys*
by (*induct xs*) *simp-all*

lemma *filtermap-conv*:
filtermap f xs = *map* (%*x*. *the*(*f x*)) (*filter* (%*x*. *f x* ≠ *None*) *xs*)
by (*induct xs*) *auto*

lemma *map-filter-conv*[*simp*]: *map-filter f P xs* = *map f* (*filter P xs*)
by (*induct xs*) *auto*

38.1.25 Code generation

Defaults for generating efficient code for some standard functions.

lemmas *in-set-code*[*code unfold*] = *mem-iff*[*symmetric*, *THEN eq-reflection*]

lemma *rev-code*[*code unfold*]: *rev xs* == *itrev xs* []
by *simp*

lemma *distinct-Cons-mem*[*code*]: *distinct* (*x*#*xs*) = (~(*x mem xs*) ∧ *distinct xs*)
by (*simp add*:*mem-iff*)

lemma *remdups-Cons-mem*[*code*]:
remdups (*x*#*xs*) = (*if x mem xs then remdups xs else x* # *remdups xs*)
by (*simp add*:*mem-iff*)

lemma *list-inter-Cons-mem*[*code*]: *list-inter* (*a*#*as*) *bs* =
(*if a mem bs then a*#(*list-inter as bs*) *else list-inter as bs*)
by(*simp add*:*mem-iff*)

For implementing bounded quantifiers over lists by *list-ex*/*list-all*:

lemmas *list-bex-code*[*code unfold*] = *list-ex-iff*[*symmetric*, *THEN eq-reflection*]
lemmas *list-ball-code*[*code unfold*] = *list-all-iff*[*symmetric*, *THEN eq-reflection*]

38.1.26 Inductive definition for membership

consts *ListMem* :: ('*a* × '*a list*)*set*
inductive *ListMem*
intros
elem: (*x*,*x*#*xs*) ∈ *ListMem*
insert: (*x*,*xs*) ∈ *ListMem* ⟹ (*x*,*y*#*xs*) ∈ *ListMem*

lemma *ListMem-iff*: ((*x*,*xs*) ∈ *ListMem*) = (*x* ∈ *set xs*)
apply (*rule iffI*)
apply (*induct set*: *ListMem*)
apply *auto*

apply (*induct xs*)
apply (*auto intro*: *ListMem.intros*)
done

38.1.27 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*}

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*)

38.2 Relations on Lists

38.2.1 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

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*. (*xs*, *ys*) : *lexn r n* ==> *length xs* = *n* ∧ *length ys* = *n*
by (*induct n*) *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*, *blast*)
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 diag-def lex-prod-def measure-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* [*iff*]:
((*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

38.2.2 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*))}

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*

38.2.3 Lifting a Relation on List Elements to the Lists

consts *listrel* :: $('a * 'a)set => ('a\ list * 'a\ list)set$

inductive *listrel*(r)
intros
Nil: $([],[]) \in listrel\ r$
Cons: $[|\ (x,y) \in r;\ (xs,ys) \in listrel\ r\ |] ==> (x\#xs,\ y\#ys) \in listrel\ r$

inductive-cases *listrel-Nil1* [*elim*!]: $([],xs) \in listrel\ r$
inductive-cases *listrel-Nil2* [*elim*!]: $(xs,[]) \in listrel\ r$
inductive-cases *listrel-Cons1* [*elim*!]: $(y\#ys,xs) \in listrel\ r$
inductive-cases *listrel-Cons2* [*elim*!]: $(xs,y\#ys) \in listrel\ r$

lemma *listrel-mono*: $r \subseteq s \Longrightarrow listrel\ r \subseteq listrel\ s$
apply *clarify*
apply (*erule listrel.induct*)
apply (*blast intro*: *listrel.intros*)+
done

lemma *listrel-subset*: $r \subseteq A \times A \Longrightarrow listrel\ r \subseteq lists\ A \times lists\ A$
apply *clarify*
apply (*erule listrel.induct*, *auto*)
done

lemma *listrel-refl*: *refl A r* $\Longrightarrow$ *refl* (*lists A*) (*listrel r*)
apply (*simp add*: *refl-def listrel-subset Ball-def*)
apply (*rule allI*)
apply (*induct-tac x*)
apply (*auto intro*: *listrel.intros*)
done

lemma *listrel-sym*: *sym r* $\Longrightarrow$ *sym* (*listrel r*)
apply (*auto simp add*: *sym-def*)
apply (*erule listrel.induct*)
apply (*blast intro*: *listrel.intros*)+
done

lemma *listrel-trans*: *trans r* $\Longrightarrow$ *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* $\Longrightarrow$ *equiv* (*lists A*) (*listrel r*)
by (*simp add*: *equiv-def listrel-refl 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*)

38.3 Miscellany

38.3.1 Characters and strings

datatype *nibble* =
Nibble0 | *Nibble1* | *Nibble2* | *Nibble3* | *Nibble4* | *Nibble5* | *Nibble6* | *Nibble7*
| *Nibble8* | *Nibble9* | *NibbleA* | *NibbleB* | *NibbleC* | *NibbleD* | *NibbleE* | *NibbleF*

datatype *char* = *Char nibble nibble*
— Note: canonical order of character encoding coincides with standard term ordering

types *string* = *char list*

syntax
-Char :: *xstr* => *char* (*CHR -*)
-String :: *xstr* => *string* (*-*)

parse-ast-translation ⟪
let
val constants = *Syntax.Appl o map Syntax.Constant*;

fun mk-nib n = *Nibble ^ chr* (*n* + (*if n* <= *9 then ord 0 else ord A* − *10*));
fun mk-char c =
if Symbol.is-ascii c andalso Symbol.is-printable c then
constants [*Char*, *mk-nib* (*ord c div 16*), *mk-nib* (*ord c mod 16*)]
else error (*Printable ASCII character expected*: ^ *quote c*);

fun mk-string [] = *Syntax.Constant Nil*
| *mk-string* (*c* :: *cs*) = *Syntax.Appl* [*Syntax.Constant Cons*, *mk-char c*, *mk-string cs*];

fun char-ast-tr [*Syntax.Variable xstr*] =
(*case Syntax.explode-xstr xstr of*
[*c*] => *mk-char c*
| - => *error* (*Single character expected*: ^ *xstr*))

```
    | char-ast-tr asts = raise AST (char-ast-tr, asts);

  fun string-ast-tr [Syntax.Variable xstr] =
      (case Syntax.explode-xstr xstr of
        [] => constants [Syntax.constrainC, Nil, string]
      | cs => mk-string cs)
    | string-ast-tr asts = raise AST (string-ast-tr, asts);
 in [(-Char, char-ast-tr), (-String, string-ast-tr)] end;
⟫

ML ⟪
fun int-of-nibble h =
  if 0 <= h andalso h <= 9 then ord h - ord 0
  else if A <= h andalso h <= F then ord h - ord A + 10
  else raise Match;

fun nibble-of-int i =
  if i <= 9 then chr (ord 0 + i) else chr (ord A + i - 10);
⟫

print-ast-translation ⟪
  let
    fun dest-nib (Syntax.Constant c) =
        (case explode c of
          [N, i, b, b, l, e, h] => int-of-nibble h
        | - => raise Match)
      | dest-nib - = raise Match;

    fun dest-chr c1 c2 =
      let val c = chr (dest-nib c1 * 16 + dest-nib c2)
      in if Symbol.is-printable c then c else raise Match end;

    fun dest-char (Syntax.Appl [Syntax.Constant Char, c1, c2]) = dest-chr c1 c2
      | dest-char - = raise Match;

   fun xstr cs = Syntax.Appl [Syntax.Constant -xstr, Syntax.Variable (Syntax.implode-xstr
cs)];

    fun char-ast-tr' [c1, c2] = Syntax.Appl [Syntax.Constant -Char, xstr [dest-chr
c1 c2]]
      | char-ast-tr' - = raise Match;

    fun list-ast-tr' [args] = Syntax.Appl [Syntax.Constant -String,
          xstr (map dest-char (Syntax.unfold-ast -args args))]
      | list-ast-tr' ts = raise Match;
  in [(Char, char-ast-tr'), (@list, list-ast-tr')] end;
⟫
```

38.3.2 Code generator setup

ML ⟪
local

fun list-codegen thy defs gr dep thyname b t =
let val (gr', ps) = foldl-map (Codegen.invoke-codegen thy defs dep thyname false)
(gr, HOLogic.dest-list t)
in SOME (gr', Pretty.list [ ] ps) end handle TERM - => NONE;

fun dest-nibble (Const (s, -)) = int-of-nibble (unprefix List.nibble.Nibble s)
| dest-nibble - = raise Match;

fun char-codegen thy defs gr dep thyname b (Const (List.char.Char, -) $ c1 $ c2)
=
*(let val c = chr (dest-nibble c1 * 16 + dest-nibble c2)*
in if Symbol.is-printable c then SOME (gr, Pretty.quote (Pretty.str c))
else NONE
end handle Fail - => NONE | Match => NONE)
| char-codegen thy defs gr dep thyname b - = NONE;

in

val list-codegen-setup =
[Codegen.add-codegen list-codegen list-codegen,
Codegen.add-codegen char-codegen char-codegen];

end;
⟫

types-code
list (- list)
attach (*term-of*) ⟪
val term-of-list = HOLogic.mk-list;
⟫
attach (*test*) ⟪
fun gen-list' aG i j = frequency
[(i, fn () => aG j :: gen-list' aG (i-1) j), (1, fn () => [])] ()
and gen-list aG i = gen-list' aG i i;
⟫
char (string)
attach (*term-of*) ⟪
val nibbleT = Type (List.nibble, []);

fun term-of-char c =
Const (List.char.Char, nibbleT --> nibbleT --> Type (List.char, [])) $
Const (List.nibble.Nibble ^ nibble-of-int (ord c div 16), nibbleT) $
Const (List.nibble.Nibble ^ nibble-of-int (ord c mod 16), nibbleT);
⟫
attach (*test*) ⟪

fun gen-char i = chr (random-range (ord a) (Int.min (ord a + i, ord z)));
⟫

consts-code *Cons* ((- ::/ -))

setup *list-codegen-setup*

end

39 Map: Maps

theory *Map*
imports *List*
begin

types ('a,'b) ~=> = 'a => 'b *option* (**infixr** *0*)
translations (*type*) *a* ~=> *b* <= (*type*) *a* => *b option*

consts
chg-map :: ('b => 'b) => 'a => ('a ~=> 'b) => ('a ~=> 'b)
map-add :: ('a ~=> 'b) => ('a ~=> 'b) => ('a ~=> 'b) (**infixl** ++ *100*)
restrict-map :: ('a ~=> 'b) => 'a *set* => ('a ~=> 'b) (**infixl** |` *110*)
dom :: ('a ~=> 'b) => 'a *set*
ran :: ('a ~=> 'b) => 'b *set*
map-of :: ('a * 'b)*list* => 'a ~=> 'b
map-upds:: ('a ~=> 'b) => 'a *list* => 'b *list* =>
('a ~=> 'b)
map-upd-s::('a ~=> 'b) => 'a *set* => 'b =>
('a ~=> 'b) (-/'(-{|->}-/') [*900*,*0*,*0*]*900*)
map-subst::('a ~=> 'b) => 'b => 'b =>
('a ~=> 'b) (-/'(-~>-/') [*900*,*0*,*0*]*900*)
map-le :: ('a ~=> 'b) => ('a ~=> 'b) => *bool* (**infix** $\subseteq_m$ *50*)

constdefs
map-comp :: ('b ~=> 'c) => ('a ~=> 'b) => ('a ~=> 'c) (**infixl** *o'-m 55*)
f o-m g == (λ*k*. *case g k of None* ⇒ *None* | *Some v* ⇒ *f v*)

nonterminals
maplets maplet

syntax
empty :: 'a ~=> 'b
-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*)
$\sim\!\!\Rightarrow$ :: [*type*, *type*] => *type* (**infixr** $\rightharpoonup$ *0*)

map-comp :: ('*b* $\sim\!\!\Rightarrow$ '*c*) => ('*a* $\sim\!\!\Rightarrow$ '*b*) => ('*a* $\sim\!\!\Rightarrow$ '*c*) (**infixl** $\circ_m$ *55*)

-maplet :: ['*a*, '*a*] => *maplet* (- /$\mapsto$/ -)
-maplets :: ['*a*, '*a*] => *maplet* (- /[$\mapsto$]/ -)

map-upd-s :: ('*a* $\sim\!\!\Rightarrow$ '*b*) => '*a set* => '*b* => ('*a* $\sim\!\!\Rightarrow$ '*b*)
(-/'(-/{$\mapsto$}/-') [*900*,*0*,*0*]*900*)
map-subst :: ('*a* $\sim\!\!\Rightarrow$ '*b*) => '*b* => '*b* =>
('*a* $\sim\!\!\Rightarrow$ '*b*) (-/'($\rightsquigarrow$-/') [*900*,*0*,*0*]*900*)
@*chg-map* :: ('*a* $\sim\!\!\Rightarrow$ '*b*) => '*a* => ('*b* => '*b*) => ('*a* $\sim\!\!\Rightarrow$ '*b*)
(-/'(-/$\mapsto\lambda$-. -') [*900*,*0*,*0*,*0*] *900*)

syntax (*latex* **output**)
restrict-map :: ('*a* $\sim\!\!\Rightarrow$ '*b*) => '*a set* => ('*a* $\sim\!\!\Rightarrow$ '*b*) (-$\restriction$- [*111*,*110*] *110*)
— requires amssymb!

translations
empty => *-K None*
empty <= *%x. None*

$m(x\mapsto\lambda y.\ f)$ == *chg-map* ($\lambda y.\ f$) *x m*

-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 empty ms*
-Map (*-Maplets ms1 ms2*) <= *-MapUpd* (*-Map ms1*) *ms2*
-Maplets ms1 (*-Maplets ms2 ms3*) <= *-Maplets* (*-Maplets ms1 ms2*) *ms3*

defs
chg-map-def: *chg-map f a m == case m a of None => m | Some b => m*(a|->f b)

map-add-def: *m1++m2 == %x. case m2 x of None => m1 x | Some y => Some y*
restrict-map-def: $m|`A$ == *%x. if x : A then m x else None*

map-upds-def: m(*xs* [|->] *ys*) == *m ++ map-of* (*rev*(*zip xs ys*))
map-upd-s-def: m(*as*{|->}*b*) == *%x. if x : as then Some b else m x*
map-subst-def: m(*a*$\sim$>*b*) == *%x. if m x = Some a then Some b else m x*

dom-def: *dom*(m) == {*a*. *m a* $\sim$= *None*}
ran-def: *ran*(m) == {*b*. *EX a*. *m a = Some b*}

map-le-def: $m_1 \subseteq_m m_2$ == *ALL a : dom* m_1. m_1 *a* = m_2 *a*

primrec
map-of [] = *empty*
map-of (*p*#*ps*) = (*map-of ps*)(*fst p* |−> *snd p*)

39.1 *empty*

lemma *empty-upd-none*[*simp*]: *empty*(*x* := *None*) = *empty*
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *sum-case-empty-empty*[*simp*]: *sum-case empty empty* = *empty*
apply (*rule ext*)
apply (*simp* (*no-asm*) *split add*: *sum.split*)
done

39.2 *map-upd*

lemma *map-upd-triv*: *t k* = *Some x* ==> *t*(*k*|−>*x*) = *t*
apply (*rule ext*)
apply (*simp* (*no-asm-simp*))
done

lemma *map-upd-nonempty*[*simp*]: *t*(*k*|−>*x*) ~= *empty*
apply *safe*
apply (*drule-tac x* = *k* **in** *fun-cong*)
apply (*simp* (*no-asm-use*))
done

lemma *map-upd-eqD1*: *m*(*a*↦*x*) = *n*(*a*↦*y*) ⟹ *x* = *y*
by (*drule fun-cong* [*of* - - *a*], *auto*)

lemma *map-upd-Some-unfold*:
((*m*(*a*|−>*b*)) *x* = *Some y*) = (*x* = *a* ∧ *b* = *y* ∨ *x* ≠ *a* ∧ *m x* = *Some y*)
by *auto*

lemma *image-map-upd*[*simp*]: *x* ∉ *A* ⟹ *m*(*x* ↦ *y*) ‘ *A* = *m* ‘ *A*
by *fastsimp*

lemma *finite-range-updI*: *finite* (*range f*) ==> *finite* (*range* (*f*(*a*|−>*b*)))
apply (*unfold image-def*)
apply (*simp* (*no-asm-use*) *add*: *full-SetCompr-eq*)
apply (*rule finite-subset*)
prefer *2* **apply** *assumption*
apply *auto*
done

39.3 *sum-case* and *empty/map-upd*

lemma *sum-case-map-upd-empty*[*simp*]:
sum-case (*m*(*k*|−>*y*)) *empty* = (*sum-case m empty*)(*Inl k*|−>*y*)
apply (*rule ext*)
apply (*simp* (*no-asm*) *split add*: *sum.split*)
done

lemma *sum-case-empty-map-upd*[*simp*]:
sum-case empty (*m*(*k*|−>*y*)) = (*sum-case empty m*)(*Inr k*|−>*y*)
apply (*rule ext*)
apply (*simp* (*no-asm*) *split add*: *sum.split*)
done

lemma *sum-case-map-upd-map-upd*[*simp*]:
sum-case (*m1*(*k1*|−>*y1*)) (*m2*(*k2*|−>*y2*)) = (*sum-case* (*m1*(*k1*|−>*y1*)) *m2*)(*Inr k2*|−>*y2*)
apply (*rule ext*)
apply (*simp* (*no-asm*) *split add*: *sum.split*)
done

39.4 *chg-map*

lemma *chg-map-new*[*simp*]: *m a* = *None* ==> *chg-map f a m* = *m*
by (*unfold chg-map-def*, *auto*)

lemma *chg-map-upd*[*simp*]: *m a* = *Some b* ==> *chg-map f a m* = *m*(*a*|−>*f b*)
by (*unfold chg-map-def*, *auto*)

lemma *chg-map-other* [*simp*]: *a* ≠ *b* ⟹ *chg-map f a m b* = *m b*
by (*auto simp*: *chg-map-def split add*: *option.split*)

39.5 *map-of*

lemma *map-of-eq-None-iff*:
(*map-of xys x* = *None*) = (*x* ∉ *fst* ' (*set xys*))
by (*induct xys*) *simp-all*

lemma *map-of-is-SomeD*:
map-of xys x = *Some y* ⟹ (*x*,*y*) ∈ *set xys*
apply(*induct xys*)
apply *simp*
apply(*clarsimp split*:*if-splits*)
done

lemma *map-of-eq-Some-iff*[*simp*]:
distinct(*map fst xys*) ⟹ (*map-of xys x* = *Some y*) = ((*x*,*y*) ∈ *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* $\Longrightarrow$ (*map-of* (*zip xs ys*) *x* = *None*) = (*x* $\notin$ *set xs*)
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *finite-range-map-of*: *finite* (*range* (*map-of xys*))
apply (*induct xys*)
apply (*simp-all* (*no-asm*) *add*: *image-constant*)
apply (*rule finite-subset*)
prefer *2* **apply** *assumption*
apply *auto*
done

lemma *map-of-SomeD* [*rule-format*]: *map-of xs k* = *Some y* −−> (*k*,*y*):*set xs*
by (*induct xs*, *auto*)

lemma *map-of-mapk-SomeI* [*rule-format*]:
inj f ==> *map-of t k* = *Some x* −−>
map-of (*map* (*split* (%*k*. *Pair* (*f k*))) *t*) (*f k*) = *Some x*
apply (*induct t*)
apply (*auto simp add*: *inj-eq*)
done

lemma *weak-map-of-SomeI* [*rule-format*]:
(*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*
apply (*rule mp*)
prefer *2* **apply** *assumption*
apply (*erule thin-rl*)
apply (*induct xs*, *auto*)
done

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*)

39.6 *option-map* related

lemma *option-map-o-empty*[*simp*]: *option-map f o empty = empty*
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

lemma *option-map-o-map-upd*[*simp*]:
option-map f o m(*a*|−>*b*) = (*option-map f o m*)(*a*|−>*f b*)
apply (*rule ext*)
apply (*simp* (*no-asm*))
done

39.7 *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*)

39.8 ++

lemma *map-add-empty*[*simp*]: *m* ++ *empty* = *m*
apply (*unfold map-add-def*)
apply (*simp* (*no-asm*))
done

lemma *empty-map-add*[*simp*]: *empty* ++ *m* = *m*
apply (*unfold map-add-def*)
apply (*rule ext*)
apply (*simp split add*: *option.split*)
done

lemma *map-add-assoc*[*simp*]: *m1* ++ (*m2* ++ *m3*) = (*m1* ++ *m2*) ++ *m3*

apply(*rule ext*)
apply(*simp add*: *map-add-def split*:*option.split*)
done

lemma *map-add-Some-iff*:
 $((m ++ n)\ k = Some\ x) = (n\ k = Some\ x \mid n\ k = None\ \&\ m\ k = Some\ x)$
apply (*unfold map-add-def*)
apply (*simp* (*no-asm*) *split add*: *option.split*)
done

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

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*)
apply (*unfold map-add-def*)
apply (*simp* (*no-asm*) *split add*: *option.split*)
done

lemma *map-add-upd*[*simp*]: *f* ++ *g*(*x*|->*y*) = (*f* ++ *g*)(*x*|->*y*)
apply (*unfold map-add-def*)
apply (*rule ext*, *auto*)
done

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*
apply (*unfold map-add-def*)
apply (*induct xs*)
apply (*simp* (*no-asm*))
apply (*rule ext*)
apply (*simp* (*no-asm-simp*) *split add*: *option.split*)
done

declare *fun-upd-apply* [*simp del*]
lemma *finite-range-map-of-map-add*:
 finite (*range f*) ==> *finite* (*range* (*f* ++ *map-of l*))
apply (*induct l*, *auto*)
apply (*erule finite-range-updI*)
done
declare *fun-upd-apply* [*simp*]

lemma *inj-on-map-add-dom*[*iff*]:
 inj-on (*m* ++ *m'*) (*dom m'*) = *inj-on m'* (*dom m'*)

by(*fastsimp simp add*:*map-add-def dom-def inj-on-def split*:*option.splits*)

39.9 *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*]: $x \in A \Longrightarrow (m|`A)\ x = m\ x$
by (*auto simp*: *restrict-map-def*)

lemma *restrict-out* [*simp*]: $x \notin A \Longrightarrow (m|`A)\ x = None$
by (*auto simp*: *restrict-map-def*)

lemma *ran-restrictD*: $y \in ran\ (m|`A) \Longrightarrow \exists\, x{\in}A.\ m\ x = Some\ y$
by (*auto simp*: *restrict-map-def ran-def split*: *split-if-asm*)

lemma *dom-restrict* [*simp*]: $dom\ (m|`A) = dom\ m \cap A$
by (*auto simp*: *restrict-map-def dom-def split*: *split-if-asm*)

lemma *restrict-upd-same* [*simp*]: $m(x{\mapsto}y)|`(-\{x\}) = m|`(-\{x\})$
by (*rule ext*, *auto simp*: *restrict-map-def*)

lemma *restrict-restrict* [*simp*]: $m|`A|`B = m|`(A{\cap}B)$
by (*rule ext*, *auto simp*: *restrict-map-def*)

lemma *restrict-fun-upd*[*simp*]:
$m(x := y)|`D = (if\ x \in D\ then\ (m|`(D{-}\{x\}))(x := y)\ else\ m|`D)$
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*)

39.10 *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*, *simp*, *simp*)
done

lemma *map-upds-list-update2-drop*[*simp*]:
$\bigwedge m\ ys\ i.\ [\![size\ xs \le i;\ i < size\ ys]\!]$
$\Longrightarrow m(xs[\mapsto]ys[i:=y]) = m(xs[\mapsto]ys)$
apply (*induct xs*, *simp*)
apply (*case-tac ys*, *simp*)
apply(*simp split*:*nat.split*)
done

lemma *map-upd-upds-conv-if*: $!!x\ y\ ys\ f.$
$(f(x|{-}{>}y))(xs\ [|{-}{>}]\ ys) =$
$(if\ x : set(take\ (length\ ys)\ xs)\ then\ f(xs\ [|{-}{>}]\ ys)$
$else\ (f(xs\ [|{-}{>}]\ ys))(x|{-}{>}y))$
apply (*induct xs*, *simp*)
apply(*case-tac ys*)
apply(*auto split*:*split-if simp*:*fun-upd-twist*)
done

lemma *map-upds-twist* [*simp*]:
$a\ {\sim}: set\ as \Longrightarrow m(a|{-}{>}b)(as[|{-}{>}]bs) = m(as[|{-}{>}]bs)(a|{-}{>}b)$
apply(*insert set-take-subset*)
apply (*fastsimp simp add*: *map-upd-upds-conv-if*)
done

lemma *map-upds-apply-nontin*[*simp*]:
$!!ys.\ x\ {\sim}: set\ xs \Longrightarrow (f(xs[|{-}{>}]ys))\ x = f\ x$
apply (*induct xs*, *simp*)
apply(*case-tac ys*)
apply(*auto simp*: *map-upd-upds-conv-if*)
done

lemma *fun-upds-append-drop*[*simp*]:
$!!m\ ys.\ size\ xs = size\ ys \Longrightarrow m(xs@zs[\mapsto]ys) = m(xs[\mapsto]ys)$
apply(*induct xs*)
apply (*simp*)
apply(*case-tac ys*)
apply *simp-all*
done

lemma *fun-upds-append2-drop*[*simp*]:
!!*m ys*. *size xs* = *size ys* ⟹ *m*(*xs*[↦]*ys*@*zs*) = *m*(*xs*[↦]*ys*)
apply(*induct xs*)
apply (*simp*)
apply(*case-tac ys*)
apply *simp-all*
done

lemma *restrict-map-upds*[*simp*]: !!*m ys*.
⟦ *length xs* = *length ys*; *set xs* ⊆ *D* ⟧
⟹ *m*(*xs* [↦] *ys*)|`*D* = (*m*|`(*D* − *set xs*))(*xs* [↦] *ys*)
apply (*induct xs*, *simp*)
apply (*case-tac ys*, *simp*)
apply(*simp add*:*Diff-insert*[*symmetric*] *insert-absorb*)
apply(*simp add*: *map-upd-upds-conv-if*)
done

39.11 *map-upd-s*

lemma *map-upd-s-apply* [*simp*]:
(*m*(*as*{|−>}*b*)) *x* = (*if x* : *as then Some b else m x*)
by (*simp add*: *map-upd-s-def*)

lemma *map-subst-apply* [*simp*]:
(*m*(*a*~>*b*)) *x* = (*if m x* = *Some a then Some b else m x*)
by (*simp add*: *map-subst-def*)

39.12 *dom*

lemma *domI*: *m a* = *Some b* ==> *a* : *dom m*
by (*unfold dom-def*, *auto*)

lemma *domD*: *a* : *dom m* ==> ∃ *b*. *m a* = *Some b*
by (*unfold dom-def*, *auto*)

lemma *domIff*[*iff*]: (*a* : *dom m*) = (*m a* ~= *None*)
by (*unfold dom-def*, *auto*)
declare *domIff* [*simp del*]

lemma *dom-empty*[*simp*]: *dom empty* = {}
apply (*unfold dom-def*)
apply (*simp* (*no-asm*))
done

lemma *dom-fun-upd*[*simp*]:
dom(*f*(*x* := *y*)) = (*if y*=*None then dom f* − {*x*} *else insert x* (*dom f*))
by (*simp add*:*dom-def*) *blast*

lemma *dom-map-of*: *dom*(*map-of xys*) = {*x*. ∃ *y*. (*x*,*y*) : *set xys*}
apply(*induct xys*)
apply(*auto simp del*:*fun-upd-apply*)
done

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*))
apply (*unfold dom-def*)
apply (*induct l*)
apply (*auto simp add*: *insert-Collect* [*symmetric*])
done

lemma *dom-map-upds*[*simp*]:
!!*m ys*. *dom*(*m*(*xs*[|->]*ys*)) = *set*(*take* (*length ys*) *xs*) *Un dom m*
apply (*induct xs*, *simp*)
apply (*case-tac ys*, *auto*)
done

lemma *dom-map-add*[*simp*]: *dom*(*m*++*n*) = *dom n Un dom m*
by (*unfold dom-def*, *auto*)

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 add*: *dom-def override-on-def*)

lemma *map-add-comm*: *dom m1* ∩ *dom m2* = {} ⟹ *m1*++*m2* = *m2*++*m1*
apply(*rule ext*)
apply(*fastsimp simp*:*map-add-def split*:*option.split*)
done

39.13 *ran*

lemma *ranI*: *m a* = *Some b* ==> *b* : *ran m*
by (*auto simp add*: *ran-def*)

lemma *ran-empty*[*simp*]: *ran empty* = {}
apply (*unfold ran-def*)
apply (*simp* (*no-asm*))
done

lemma *ran-map-upd*[*simp*]: $m\ a = None ==> ran(m(a|->b)) = insert\ b\ (ran\ m)$
apply (*unfold ran-def*, *auto*)
apply (*subgoal-tac* $^{\sim}$ $(aa = a)$)
apply *auto*
done

39.14 *map-le*

lemma *map-le-empty* [*simp*]: $empty \subseteq_m g$
by(*simp add*:*map-le-def*)

lemma *upd-None-map-le* [*simp*]: $f(x := None) \subseteq_m f$
by(*force simp add*:*map-le-def*)

lemma *map-le-upd*[*simp*]: $f \subseteq_m g ==> f(a := b) \subseteq_m g(a := b)$
by(*fastsimp simp add*:*map-le-def*)

lemma *map-le-imp-upd-le* [*simp*]: $m1 \subseteq_m m2 \Longrightarrow m1(x := None) \subseteq_m m2(x \mapsto y)$
by(*force simp add*:*map-le-def*)

lemma *map-le-upds*[*simp*]:
$!!f\ g\ bs.\ f \subseteq_m g ==> f(as\ [|->]\ bs) \subseteq_m g(as\ [|->]\ bs)$
apply (*induct as*, *simp*)
apply (*case-tac bs*, *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(*force simp add*:*map-le-def*)

lemma *map-le-antisym*: $[\![\ f \subseteq_m g;\ g \subseteq_m f\ ]\!] \Longrightarrow f = g$
apply (*unfold 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 add*:*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

40 Refute: Refute

theory *Refute*
imports *Map*
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.  Defining equations for constants are added  *)
(* automatically, as are sort axioms.  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

41 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* = ⟪ *Method.no-args* (*Method.SIMPLE-METHOD sat.sat-tac*)
⟫
SAT solver

method-setup *satx* = ⟪ *Method.no-args* (*Method.SIMPLE-METHOD sat.satx-tac*)
⟫
SAT solver (*with definitional CNF*)

end

42 Hilbert-Choice: Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*

imports *NatArith*
uses (*Tools/meson.ML*) (*Tools/specification-package.ML*)
begin

42.1 Hilbert's epsilon

consts
  Eps :: (*'a => bool*) => *'a*

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 == Eps (%x. P)

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

axioms
  someI: *P* (*x*::*'a*) ==> *P* (*SOME x. P x*)
finalconsts
  Eps

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

42.2 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

42.3 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*)

42.4 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 *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*: $[|\ surj\ f;\ \forall x.\ g(f\ x) = x\ |] ==> inv\ f = g$
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

42.5 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

42.6 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 add*: *split-Pair-apply*)

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*:
  $wf\ r = (\sim(\exists f.\ \forall i.\ (f(Suc\ i),f\ i) \in r))$
apply (*simp only*: *wf-eq-minimal*)
apply (*rule iffI*)
 apply (*rule notI*)
 apply (*erule exE*)
 apply (*erule-tac* $x = \{w.\ \exists i.\ w=f\ i\}$ **in** *allE*, *blast*)
apply (*erule contrapos-np*, *simp*, *clarify*)
apply (*subgoal-tac* $\forall n.\ nat\text{-}rec\ x\ (\%i\ y.\ @z.\ z{:}Q\ \&\ (z,y)\ {:}r)\ n \in Q$)
 apply (*rule-tac* $x = nat\text{-}rec\ x\ (\%i\ y.\ @z.\ z{:}Q\ \&\ (z,y)\ {:}r)$ **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

A dynamically-scoped fact for TFL

lemma *tfl-some*: $\forall P\ x.\ P\ x \longrightarrow P\ (Eps\ P)$
  by (*blast intro*: *someI*)

42.7 Least value operator

constdefs
  $LeastM :: ['a \Rightarrow 'b{::}ord, 'a \Rightarrow bool] \Rightarrow 'a$
  $LeastM\ m\ P == SOME\ x.\ P\ x\ \&\ (\forall y.\ P\ y \longrightarrow m\ x <= m\ y)$

syntax
  -LeastM :: [*pttrn*, $'a \Rightarrow 'b{::}ord$, *bool*] $\Rightarrow 'a$ (*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*)

42.8 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*:
$P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow m\ y < b$
$\Longrightarrow (m\ x::nat) \leq m\ (GreatestM\ m\ P)$
apply (*blast dest*: *GreatestM-nat-lemma* [*THEN conjunct2*, *THEN spec*])
done

Specialization to *GREATEST*.

lemma *GreatestI*: $P\ (k::nat) \Longrightarrow \forall y.\ P\ y \longrightarrow y < b \Longrightarrow P\ (GREATEST\ x.\ P\ x)$
apply (*simp add*: *Greatest-def*)
apply (*rule GreatestM-natI*, *auto*)
done

lemma *Greatest-le*:
$P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow y < b \Longrightarrow (x::nat) \leq (GREATEST\ x.\ P\ x)$
apply (*simp add*: *Greatest-def*)
apply (*rule GreatestM-nat-le*, *auto*)
done

42.9 The Meson proof procedure

42.9.1 Negation Normal Form

de Morgan laws

lemma *meson-not-conjD*: $\sim(P\&Q) \Longrightarrow \sim P \mid \sim Q$
and *meson-not-disjD*: $\sim(P|Q) \Longrightarrow \sim P\ \&\ \sim Q$
and *meson-not-notD*: $\sim\sim P \Longrightarrow P$
and *meson-not-allD*: $!!P.\ \sim(\forall x.\ P(x)) \Longrightarrow \exists x.\ \sim P(x)$
and *meson-not-exD*: $!!P.\ \sim(\exists x.\ P(x)) \Longrightarrow \forall x.\ \sim P(x)$
by *fast+*

Removal of $\longrightarrow$ and $<\!\!\longrightarrow$ (positive and negative occurrences)

lemma *meson-imp-to-disjD*: $P\longrightarrow Q \Longrightarrow \sim P \mid Q$
and *meson-not-impD*: $\sim(P\longrightarrow Q) \Longrightarrow P\ \&\ \sim Q$
and *meson-iff-to-disjD*: $P=Q \Longrightarrow (\sim P \mid Q)\ \&\ (\sim Q \mid P)$
and *meson-not-iffD*: $\sim(P=Q) \Longrightarrow (P \mid Q)\ \&\ (\sim P \mid \sim Q)$
— Much more efficient than $P \wedge \neg\ Q \vee Q \wedge \neg\ P$ for computing CNF
by *fast+*

42.9.2 Pulling out the existential quantifiers

Conjunction

lemma *meson-conj-exD1*: !!$P\ Q.\ (\exists x.\ P(x))\ \&\ Q \Longrightarrow \exists x.\ P(x)\ \&\ Q$
and *meson-conj-exD2*: !!$P\ Q.\ P\ \&\ (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P\ \&\ Q(x)$
by *fast+*

Disjunction

lemma *meson-disj-exD*: !!$P\ Q.\ (\exists x.\ P(x)) \mid (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P(x) \mid Q(x)$
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants
and *meson-disj-exD1*: !!$P\ Q.\ (\exists x.\ P(x)) \mid Q \Longrightarrow \exists x.\ P(x) \mid Q$
and *meson-disj-exD2*: !!$P\ Q.\ P \mid (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P \mid Q(x)$
by *fast+*

42.9.3 Generating clauses for the Meson Proof Procedure

Disjunctions

lemma *meson-disj-assoc*: $(P|Q)|R \Longrightarrow P|(Q|R)$
and *meson-disj-comm*: $P|Q \Longrightarrow Q|P$
and *meson-disj-FalseD1*: $False|P \Longrightarrow P$
and *meson-disj-FalseD2*: $P|False \Longrightarrow P$
by *fast+*

42.10 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*: $\sim P|Q \Longrightarrow ((\sim P \Longrightarrow P) \Longrightarrow Q)$
by *blast*

Version for Plaisted's "Postive refinement" of the Meson procedure

lemma *make-refined-neg-rule*: $\sim P|Q \Longrightarrow (P \Longrightarrow Q)$
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*: $^{\sim}P$ ==> (($^{\sim}P$==>P) ==> *False*)
by *blast*

lemma *make-pos-goal*: P ==> ((P==>$^{\sim}P$) ==> *False*)
by *blast*

42.10.1 Lemmas for Forward Proof

There is a similarity to congruence rules

lemma *conj-forward*: [| P'&Q'; P' ==> P; Q' ==> Q |] ==> P&Q
by *blast*

lemma *disj-forward*: [| P'|Q'; P' ==> P; Q' ==> Q |] ==> P|Q
by *blast*

lemma *disj-forward2*:
[| P'|Q'; P' ==> P; [| Q'; P==>*False* |] ==> Q |] ==> P|Q
apply *blast*
done

lemma *all-forward*: [| $\forall x.\ P'(x)$; !!$x.\ P'(x)$ ==> $P(x)$ |] ==> $\forall x.\ P(x)$
by *blast*

lemma *ex-forward*: [| $\exists x.\ P'(x)$; !!$x.\ P'(x)$ ==> $P(x)$ |] ==> $\exists x.\ P(x)$
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;
⟩

use *Tools/meson.ML*
setup *Meson.skolemize-setup*

use *Tools/specification-package.ML*

end

43 Infinite-Set: Infnite Sets and Related Concepts

theory *Infinite-Set*
imports *Hilbert-Choice Binomial*
begin

43.1 Infinite Sets

Some elementary facts about infinite sets, by Stefan Merz.

syntax
infinite :: *'a set* ⇒ *bool*
translations
infinite S == *S* ∉ *Finites*

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

lemma *infinite-nonempty*:
¬ (*infinite* {})
by *simp*

lemma *infinite-remove*:
infinite S ⟹ *infinite* (*S* − {*a*})
by *simp*

lemma *Diff-infinite-finite*:
assumes *T*: *finite T* **and** *S*: *infinite S*
shows *infinite* (*S*−*T*)
using *T*
proof (*induct*)
from *S*
show *infinite* (*S* − {}) **by** *auto*
next
fix *T x*
assume *ih*: *infinite* (*S*−*T*)
have *S* − (*insert x T*) = (*S*−*T*) − {*x*}
by (*rule Diff-insert*)
with *ih*
show *infinite* (*S* − (*insert x T*))
by (*simp add*: *infinite-remove*)
qed

lemma *Un-infinite*:

infinite $S \Longrightarrow$ *infinite* $(S \cup T)$
by *simp*

lemma *infinite-super*:
assumes T: $S \subseteq T$ **and** S: *infinite* S
shows *infinite* T
proof (*rule ccontr*)
assume $\neg$(*infinite* T)
with T
have *finite* S **by** (*simp add*: *finite-subset*)
with S
show *False* **by** *simp*
qed

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

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

lemma *finite-nat-iff-bounded*:
finite (S::*nat set*) = ($\exists k.\ S \subseteq \{..<k\}$) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
thus *?rhs* **by** (*rule finite-nat-bounded*)
next
assume *?rhs*
then obtain k **where** $S \subseteq \{..<k\}$ **..**
thus *finite* S
by (*rule finite-subset*, *simp*)
qed

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

lemma *infinite-nat-iff-unbounded*:
infinite (*S*::*nat set*) = (∀ *m*. ∃ *n*. *m*<*n* ∧ *n*∈*S*)
(**is** *?lhs* = *?rhs*)
proof
assume *inf*: *?lhs*
show *?rhs*
proof (*rule ccontr*)
assume ¬ *?rhs*
then obtain *m* **where** *m*: ∀ *n*. *m*<*n* ⟶ *n*∉*S* **by** *blast*
hence *S* ⊆ {..*m*}
by (*auto simp add*: *sym*[*OF linorder-not-less*])
with *inf* **show** *False*
by (*simp add*: *finite-nat-iff-bounded-le*)
qed
next
assume *unbounded*: *?rhs*
show *?lhs*
proof
assume *finite S*
then obtain *m* **where** *S* ⊆ {..*m*}
by (*auto simp add*: *finite-nat-iff-bounded-le*)
hence ∀ *n*. *m*<*n* ⟶ *n*∉*S* **by** *auto*
with *unbounded* **show** *False* **by** *blast*
qed
qed

lemma *infinite-nat-iff-unbounded-le*:
infinite (*S*::*nat set*) = (∀ *m*. ∃ *n*. *m*≤*n* ∧ *n*∈*S*)
(**is** *?lhs* = *?rhs*)
proof
assume *inf*: *?lhs*
show *?rhs*
proof
fix *m*

```
    from inf obtain n where m<n ∧ n∈S
      by (auto simp add: infinite-nat-iff-unbounded)
    hence m≤n ∧ n∈S by auto
    thus ∃ n. m ≤ n ∧ n ∈ S ..
  qed
next
  assume unbounded: ?rhs
  show ?lhs
  proof (auto simp add: infinite-nat-iff-unbounded)
    fix m
    from unbounded obtain n where (Suc m)≤n ∧ n∈S
      by blast
    hence m<n ∧ n∈S by auto
    thus ∃ n. m < n ∧ n ∈ S ..
  qed
qed
```

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some k, there is some larger number that is an element of the set.

```
lemma unbounded-k-infinite:
  assumes k: ∀ m. k<m ⟶ (∃ n. m<n ∧ n∈S)
  shows infinite (S::nat set)
proof (auto simp add: infinite-nat-iff-unbounded)
  fix m show ∃ n. m<n ∧ n∈S
  proof (cases k<m)
    case True
    with k show ?thesis by blast
  next
    case False
    from k obtain n where Suc k < n ∧ n∈S by auto
    with False have m<n ∧ n∈S by auto
    thus ?thesis ..
  qed
qed

theorem nat-infinite [simp]:
  infinite (UNIV :: nat set)
by (auto simp add: infinite-nat-iff-unbounded)

theorem nat-not-finite [elim]:
  finite (UNIV::nat set) ⟹ R
by simp
```

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

```
lemma range-inj-infinite:
  inj (f::nat ⇒ 'a) ⟹ infinite (range f)
```

proof
  assume *inj f*
    and *finite* (*range f*)
  hence *finite* (*UNIV*::*nat set*)
    by (*auto intro*: *finite-imageD simp del*: *nat-infinite*)
  thus *False* **by** *simp*
qed

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

lemma *linorder-injI*:
  assumes *hyp*: $\forall\, x\ y.\ x < (y::'a::linorder) \longrightarrow f\ x \neq f\ y$
  shows *inj f*
proof (*rule inj-onI*)
  fix *x y*
  assume *f-eq*: $f\ x = f\ y$
  show $x = y$
  proof (*rule linorder-cases*)
    assume $x < y$
    with *hyp* **have** $f\ x \neq f\ y$ **by** *blast*
    with *f-eq* **show** *?thesis* **by** *simp*
  next
    assume $x = y$
    thus *?thesis* **.**
  next
    assume $y < x$
    with *hyp* **have** $f\ y \neq f\ x$ **by** *blast*
    with *f-eq* **show** *?thesis* **by** *simp*
  qed
qed

lemma *infinite-countable-subset*:
  assumes *inf*: *infinite* (*S*::*'a set*)
  shows $\exists f.\ inj\ (f::nat \Rightarrow 'a) \wedge range\ f \subseteq S$
proof −
  def *Sseq* ≡ *nat-rec S* ($\lambda n\ T.\ T - \{SOME\ e.\ e \in T\}$)
  def *pick* ≡ $\lambda n.$ (*SOME e*. $e \in Sseq\ n$)
  have *Sseq-inf*: $\bigwedge n.$ *infinite* (*Sseq n*)
  proof −
    fix *n*
    show *infinite* (*Sseq n*)
    proof (*induct n*)
      from *inf* **show** *infinite* (*Sseq 0*)
        by (*simp add*: *Sseq-def*)
    next
      fix *n*
      assume *infinite* (*Sseq n*) **thus** *infinite* (*Sseq* (*Suc n*))

```
      by (simp add: Sseq-def infinite-remove)
  qed
qed
have Sseq-S: ⋀n. Sseq n ⊆ S
proof −
  fix n
  show Sseq n ⊆ S
    by (induct n, auto simp add: Sseq-def)
qed
have Sseq-pick: ⋀n. pick n ∈ Sseq n
proof −
  fix n
  show pick n ∈ Sseq n
  proof (unfold pick-def, rule someI-ex)
    from Sseq-inf have infinite (Sseq n) .
    hence Sseq n ≠ {} by auto
    thus ∃ x. x ∈ Sseq n by auto
  qed
qed
with Sseq-S have rng: range pick ⊆ S
  by auto
have pick-Sseq-gt: ⋀n m. pick n ∉ Sseq (n + Suc m)
proof −
  fix n m
  show pick n ∉ Sseq (n + Suc m)
    by (induct m, auto simp add: Sseq-def pick-def)
qed
have pick-pick: ⋀n m. pick n ≠ pick (n + Suc m)
proof −
  fix n m
  from Sseq-pick have pick (n + Suc m) ∈ Sseq (n + Suc m) .
  moreover from pick-Sseq-gt
  have pick n ∉ Sseq (n + Suc m) .
  ultimately show pick n ≠ pick (n + Suc m)
    by auto
qed
have inj: inj pick
proof (rule linorder-injI)
  show ∀ i j. i<(j::nat) ⟶ pick i ≠ pick j
  proof (clarify)
    fix i j
    assume ij: i<(j::nat)
      and eq: pick i = pick j
    from ij obtain k where j = i + (Suc k)
      by (auto simp add: less-iff-Suc-add)
    with pick-pick have pick i ≠ pick j by simp
    with eq show False by simp
  qed
qed
```

from *rng inj* **show** *?thesis* **by** *auto*
qed

theorem *infinite-iff-countable-subset*:
infinite $S = (\exists f.\ inj\ (f{::}nat \Rightarrow 'a) \wedge range\ f \subseteq S)$
(**is** *?lhs* = *?rhs*)
by (*auto simp add*: *infinite-countable-subset*
range-inj-infinite infinite-super)

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

theorem *inf-img-fin-dom*:
assumes *img*: *finite* (*f*`*A*) **and** *dom*: *infinite A*
shows $\exists y \in f`A.$ *infinite* $(f - `\ \{y\})$
proof (*rule ccontr*)
assume $\neg\ (\exists y \in f`A.$ *infinite* $(f - `\ \{y\}))$
with *img* **have** *finite* $(UN\ y{:}f`A.\ f - `\ \{y\})$
by (*blast intro*: *finite-UN-I*)
moreover have $A \subseteq (UN\ y{:}f`A.\ f - `\ \{y\})$ **by** *auto*
moreover note *dom*
ultimately show *False*
by (*simp add*: *infinite-super*)
qed

theorems *inf-img-fin-domE* = *inf-img-fin-dom*[*THEN bexE*]

43.2 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

consts
Inf-many :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *INF* *10*)
Alm-all :: $('a \Rightarrow bool) \Rightarrow bool$ (**binder** *MOST* *10*)

defs
INF-def: *Inf-many P* $\equiv$ *infinite* $\{x.\ P\ x\}$
MOST-def: *Alm-all P* $\equiv \neg(INF\ x.\ \neg\ P\ x)$

syntax (*xsymbols*)
MOST :: [*idts*, *bool*] $\Rightarrow$ *bool* ((*3*$\forall_\infty$-./ -) [*0*,*10*] *10*)
INF :: [*idts*, *bool*] $\Rightarrow$ *bool* ((*3*$\exists_\infty$-./ -) [*0*,*10*] *10*)

syntax (*HTML* **output**)
MOST :: [*idts*, *bool*] $\Rightarrow$ *bool* ((*3*$\forall_\infty$-./ -) [*0*,*10*] *10*)
INF :: [*idts*, *bool*] $\Rightarrow$ *bool* ((*3*$\exists_\infty$-./ -) [*0*,*10*] *10*)

lemma *INF-EX*:
$(\exists_\infty x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$
proof (*unfold INF-def*, *rule ccontr*)
assume *inf*: *infinite* $\{x.\ P\ x\}$
and *notP*: $\neg(\exists x.\ P\ x)$
from *notP* **have** $\{x.\ P\ x\} = \{\}$ **by** *simp*
hence *finite* $\{x.\ P\ x\}$ **by** *simp*
with *inf* **show** *False* **by** *simp*
qed

lemma *MOST-iff-finiteNeg*:
$(\forall_\infty x.\ P\ x) = \mathit{finite}\ \{x.\ \neg\ P\ x\}$
by (*simp add*: *MOST-def INF-def*)

lemma *ALL-MOST*:
$\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$
by (*simp add*: *MOST-iff-finiteNeg*)

lemma *INF-mono*:
assumes *inf*: $\exists_\infty x.\ P\ x$ **and** *q*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $\exists_\infty x.\ Q\ x$
proof −
from *inf* **have** *infinite* $\{x.\ P\ x\}$ **by** (*unfold INF-def*)
moreover from *q* **have** $\{x.\ P\ x\} \subseteq \{x.\ Q\ x\}$ **by** *auto*
ultimately show *?thesis*
by (*simp add*: *INF-def infinite-super*)
qed

lemma *MOST-mono*:
$[\![\ \forall_\infty x.\ P\ x;\ \bigwedge x.\ P\ x \Longrightarrow Q\ x\ ]\!] \Longrightarrow \forall_\infty x.\ Q\ x$
by (*unfold MOST-def*, *blast intro*: *INF-mono*)

lemma *INF-nat*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall m.\ \exists n.\ m<n \wedge P\ n)$
by (*simp add*: *INF-def infinite-nat-iff-unbounded*)

lemma *INF-nat-le*: $(\exists_\infty n.\ P\ (n::nat)) = (\forall m.\ \exists n.\ m\leq n \wedge P\ n)$
by (*simp add*: *INF-def infinite-nat-iff-unbounded-le*)

lemma *MOST-nat*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists m.\ \forall n.\ m<n \longrightarrow P\ n)$
by (*simp add*: *MOST-def INF-nat*)

lemma *MOST-nat-le*: $(\forall_\infty n.\ P\ (n::nat)) = (\exists m.\ \forall n.\ m\leq n \longrightarrow P\ n)$
by (*simp add*: *MOST-def INF-nat-le*)

43.3 Miscellaneous

A few trivial lemmas about sets that contain at most one element. These simplify the reasoning about deterministic automata.

constdefs
atmost-one :: *'a set* ⇒ *bool*
atmost-one S ≡ ∀ *x y*. *x*∈*S* ∧ *y*∈*S* ⟶ *x*=*y*

lemma *atmost-one-empty*: *S*={} ⟹ *atmost-one S*
by (*simp add*: *atmost-one-def*)

lemma *atmost-one-singleton*: *S* = {*x*} ⟹ *atmost-one S*
by (*simp add*: *atmost-one-def*)

lemma *atmost-one-unique* [*elim*]: ⟦ *atmost-one S*; *x* ∈ *S*; *y* ∈ *S* ⟧ ⟹ *y*=*x*
by (*simp add*: *atmost-one-def*)

end

44 Extraction: Program extraction for HOL

theory *Extraction*
imports *Datatype*
uses *Tools/rewrite-hol-proof.ML*
begin

44.1 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*, *binder-types U* @
[*HOLogic.dest-setT* (*body-type U*)] ---> *HOLogic.boolT*), *ts* @ [*x*]))
| (*Free* (*s*, *U*), *ts*) => *SOME* (*list-comb* (*Free* (*s*, *binder-types U* @
[*HOLogic.dest-setT* (*body-type U*)] ---> *HOLogic.boolT*), *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* :: *binder-types U* @
[*HOLogic.dest-setT* (*body-type U*)] ---> *HOLogic.boolT*), *r* :: *ts* @ [*x*]))
| (*Free* (*s*, *U*), *ts*) => *SOME* (*list-comb* (*Free* (*s*, *T* :: *binder-types U* @
[*HOLogic.dest-setT* (*body-type U*)] ---> *HOLogic.boolT*), *r* :: *ts* @ [*x*]))
| - => *NONE*)
| *realizes-set-proc* - = *NONE*;

fun mk-realizes-set r rT s (*setT as Type* (*set*, [*elT*])) =
Abs (*x*, *elT*, *Const* (*realizes*, *rT* --> *HOLogic.boolT* --> *HOLogic.boolT*) $
incr-boundvars 1 r $ (*Const* (*op* :, *elT* --> *setT* --> *HOLogic.boolT*) $
Bound 0 $ *incr-boundvars 1 s*));
in
[*Extraction.add-types*

```
    [(bool, ([], NONE)),
     (set, ([realizes-set-proc], SOME mk-realizes-set))],
   Extraction.set-preprocessor (fn thy =>
     Proofterm.rewrite-proof-notypes
       ([], (HOL/elim-cong, RewriteHOLProof.elim-cong) ::
         ProofRewriteRules.rprocs true) o
     Proofterm.rewrite-proof thy
       (RewriteHOLProof.rews, ProofRewriteRules.rprocs true) o
     ProofRewriteRules.elim-vars (curry Const arbitrary))]
end
⟩⟩
```

lemmas [*extraction-expand*] =
atomize-eq atomize-all atomize-imp
allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
notE′ impE′ impE iffE imp-cong simp-thms
induct-forall-eq induct-implies-eq induct-equal-eq
induct-forall-def induct-implies-def induct-impliesI
induct-atomize induct-rulify1 induct-rulify2

datatype *sumbool* = *Left* | *Right*

44.2 Type of extracted program

extract-type

$$typeof\ (Trueprop\ P) \equiv typeof\ P$$

$$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('Q))$$

$$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE(Null))$$

$$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('P \Rightarrow 'Q))$$

$$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow typeof\ (\forall x.\ P\ x) \equiv Type\ (TYPE(Null))$$

$$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow typeof\ (\forall x::'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$$

$$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow typeof\ (\exists x::'a.\ P\ x) \equiv Type\ (TYPE('a))$$

$$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow typeof\ (\exists x::'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$$

$$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

44.3 Realizability

realizability

$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv (\forall x::'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P::bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

44.4 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 $R\ (case\ x\ of\ Inl\ p \Rightarrow f\ p \mid Inr\ q \Rightarrow g\ q)$
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 ⟹ R f* **and** *r2*: *⋀q. Q q ⟹ 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 ⟹ R f* **and** *r2*: *Q ⟹ 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 ⟹ Q q ⟹ P (fst (p, q)) ∧ Q (snd (p, q))
by *simp*

theorem *exI-realizer*:
P y x ⟹ P (snd (x, y)) (fst (x, y)) **by** *simp*

theorem *exE-realizer*: *P (snd p) (fst p) ⟹*
(⋀x y. P y x ⟹ Q (f x y)) ⟹ Q (let (x, y) = p in f x y)
by (*cases p*) (*simp add*: *Let-def*)

theorem *exE-realizer'*: *P (snd p) (fst p) ⟹*
(⋀x y. P y x ⟹ Q) ⟹ Q **by** (*cases p*) *simp*

realizers
impI (P, Q): *λpq. 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): *λq. q Λ P Q q. impI · - · -*

impI: *Null impI*

mp (P, Q): *λpq. pq*

$\Lambda\ P\ Q\ pq\ (h\colon \text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (spec \cdot \text{-} \cdot p \bullet h)$

mp (P): *Null*
$\Lambda\ P\ Q\ (h\colon \text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (spec \cdot \text{-} \cdot p \bullet h)$

mp (Q): $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ mp \cdot \text{-} \cdot \text{-}$

mp: *Null mp*

allI (P): $\lambda p.\ p\ \Lambda\ P\ p.\ allI \cdot \text{-}$

allI: *Null allI*

spec (P): $\lambda x\ p.\ p\ x\ \Lambda\ P\ x\ p.\ spec \cdot \text{-} \cdot x$

spec: *Null spec*

exI (P): $\lambda x\ p.\ (x,\ p)\ \Lambda\ P\ x\ p.\ exI\text{-}realizer \cdot P \cdot p \cdot x$

exI: $\lambda x.\ x\ \Lambda\ P\ x\ (h\colon \text{-}).\ h$

exE $(P,\ Q)$: $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
$\Lambda\ P\ Q\ p\ (h\colon \text{-})\ pq.\ exE\text{-}realizer \cdot P \cdot p \cdot Q \cdot pq \bullet h$

exE (P): *Null*
$\Lambda\ P\ Q\ p.\ exE\text{-}realizer' \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

exE (Q): $\lambda x\ pq.\ pq\ x$
$\Lambda\ P\ Q\ x\ (h1\colon \text{-})\ pq\ (h2\colon \text{-}).\ h2 \cdot x \bullet h1$

exE: *Null*
$\Lambda\ P\ Q\ x\ (h1\colon \text{-})\ (h2\colon \text{-}).\ h2 \cdot x \bullet h1$

conjI $(P,\ Q)$: *Pair*
$\Lambda\ P\ Q\ p\ (h\colon \text{-})\ q.\ conjI\text{-}realizer \cdot P \cdot p \cdot Q \cdot q \bullet h$

conjI (P): $\lambda p.\ p$
$\Lambda\ P\ Q\ p.\ conjI \cdot \text{-} \cdot \text{-}$

conjI (Q): $\lambda q.\ q$
$\Lambda\ P\ Q\ (h\colon \text{-})\ q.\ conjI \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet h$

conjI: *Null conjI*

conjunct1 $(P,\ Q)$: *fst*
$\Lambda\ P\ Q\ pq.\ conjunct1 \cdot \text{-} \cdot \text{-}$

conjunct1 (P): $\lambda p.\ p$
$\Lambda\ P\ Q\ p.\ conjunct1 \cdot \text{-} \cdot \text{-}$

conjunct1 (*Q*): *Null*
  Λ *P Q q*. *conjunct1* · - · -

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*
  Λ *P Q pq*. *conjunct2* · - · -

conjunct2 (*P*): *Null*
  Λ *P Q p*. *conjunct2* · - · -

conjunct2 (*Q*): λ*p*. *p*
  Λ *P Q p*. *conjunct2* · - · -

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*
  Λ *P Q p*. *iffD2* · - · - • (*sum.cases-1* · *P* · - · *p*)

disjI1 (*P*): *Some*
  Λ *P Q p*. *iffD2* · - · - • (*option.cases-2* · - · *P* · *p*)

disjI1 (*Q*): *None*
  Λ *P Q*. *iffD2* · - · - • (*option.cases-1* · - · -)

disjI1: *Left*
  Λ *P Q*. *iffD2* · - · - • (*sumbool.cases-1* · - · -)

disjI2 (*P*, *Q*): *Inr*
  Λ *Q P q*. *iffD2* · - · - • (*sum.cases-2* · - · *Q* · *q*)

disjI2 (*P*): *None*
  Λ *Q P*. *iffD2* · - · - • (*option.cases-1* · - · -)

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*): *arbitrary*
Λ *P*. *FalseE* · -

FalseE: *Null FalseE*

notI (*P*): *Null*
Λ *P* (*h*: -). *allI* · - • (Λ *x*. *notI* · - • (*h* · *x*))

notI: *Null notI*

notE (*P*, *R*): λ*p*. *arbitrary*
Λ *P R* (*h*: -) *p*. *notE* · - · - · - • (*spec* · - · *p* • *h*)

notE (*P*): *Null*
Λ *P R* (*h*: -) *p*. *notE* · - · - · - • (*spec* · - · *p* • *h*)

notE (*R*): *arbitrary*
Λ *P R*. *notE* · - · - · -

notE: *Null notE*

subst (*P*): λ*s t ps*. *ps*
Λ *s t P* (*h*: -) *ps*. *subst* · *s* · *t* · *P ps* • *h*

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
 $\Lambda\ Q\ P\ pq\ (h\text{: -})\ p.$
  $mp \cdot - \cdot - \cdot (spec \cdot - \cdot p \cdot (conjunct1 \cdot - \cdot - \cdot h))$

iffD1 (*P*): $\lambda p.\ p$
 $\Lambda\ Q\ P\ p\ (h\text{: -}).\ mp \cdot - \cdot - \cdot (conjunct1 \cdot - \cdot - \cdot h)$

iffD1 (*Q*): *Null*
 $\Lambda\ Q\ P\ q1\ (h\text{: -})\ q2.$
  $mp \cdot - \cdot - \cdot (spec \cdot - \cdot q2 \cdot (conjunct1 \cdot - \cdot - \cdot h))$

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
 $\Lambda\ P\ Q\ pq\ (h\text{: -})\ q.$
  $mp \cdot - \cdot - \cdot (spec \cdot - \cdot q \cdot (conjunct2 \cdot - \cdot - \cdot h))$

iffD2 (*P*): $\lambda p.\ p$
 $\Lambda\ P\ Q\ p\ (h\text{: -}).\ mp \cdot - \cdot - \cdot (conjunct2 \cdot - \cdot - \cdot h)$

iffD2 (*Q*): *Null*
 $\Lambda\ P\ Q\ q1\ (h\text{: -})\ q2.$
  $mp \cdot - \cdot - \cdot (spec \cdot - \cdot q2 \cdot (conjunct2 \cdot - \cdot - \cdot h))$

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
 $\Lambda\ P\ Q\ pq\ (h1 : \text{-})\ qp\ (h2 : \text{-}).\ conjI\text{-}realizer \cdot$
  $(\lambda pq.\ \forall x.\ P\ x \longrightarrow Q\ (pq\ x)) \cdot pq \cdot$
  $(\lambda qp.\ \forall x.\ Q\ x \longrightarrow P\ (qp\ x)) \cdot qp \cdot$
  $(allI \cdot - \cdot (\Lambda\ x.\ impI \cdot - \cdot - \cdot (h1 \cdot x))) \cdot$
  $(allI \cdot - \cdot (\Lambda\ x.\ impI \cdot - \cdot - \cdot (h2 \cdot x)))$

iffI (*P*): $\lambda p.\ p$
 $\Lambda\ P\ Q\ (h1 : \text{-})\ p\ (h2 : \text{-}).\ conjI \cdot - \cdot - \cdot$
  $(allI \cdot - \cdot (\Lambda\ x.\ impI \cdot - \cdot - \cdot (h1 \cdot x))) \cdot$
  $(impI \cdot - \cdot - \cdot h2)$

iffI (*Q*): $\lambda q.\ q$
 $\Lambda\ P\ Q\ q\ (h1 : \text{-})\ (h2 : \text{-}).\ conjI \cdot - \cdot - \cdot$
  $(impI \cdot - \cdot - \cdot h1) \cdot$
  $(allI \cdot - \cdot (\Lambda\ x.\ impI \cdot - \cdot - \cdot (h2 \cdot x)))$

iffI: *Null iffI*

end

45 Reconstruction: Reconstructing external resolution proofs

theory *Reconstruction*
imports *Hilbert-Choice Map Infinite-Set Extraction*
uses *Tools/res-lib.ML*

Tools/res-clause.ML
Tools/res-skolem-function.ML
Tools/res-axioms.ML
Tools/res-types-sorts.ML

Tools/ATP/recon-order-clauses.ML
Tools/ATP/recon-translate-proof.ML
Tools/ATP/recon-parse.ML
Tools/ATP/recon-transfer-proof.ML
Tools/ATP/AtpCommunication.ML
Tools/ATP/watcher.ML
Tools/ATP/res-clasimpset.ML
Tools/res-atp.ML
Tools/reconstruction.ML

begin

setup *ResAxioms.meson-method-setup*
setup *Reconstruction.setup*

end

46 Main: Main HOL

theory *Main*
imports *SAT Reconstruction*
begin

Theory *Main* includes everything. Note that theory *PreList* already includes most HOL theories.

Late clause setup: installs *all* simprules and claset rules into the clause cache; cf. theory *Reconstruction*.

setup *ResAxioms.clause-setup*

end

References

[1] H. Davenport. *The Higher Arithmetic.* Cambridge University Press, 1992.