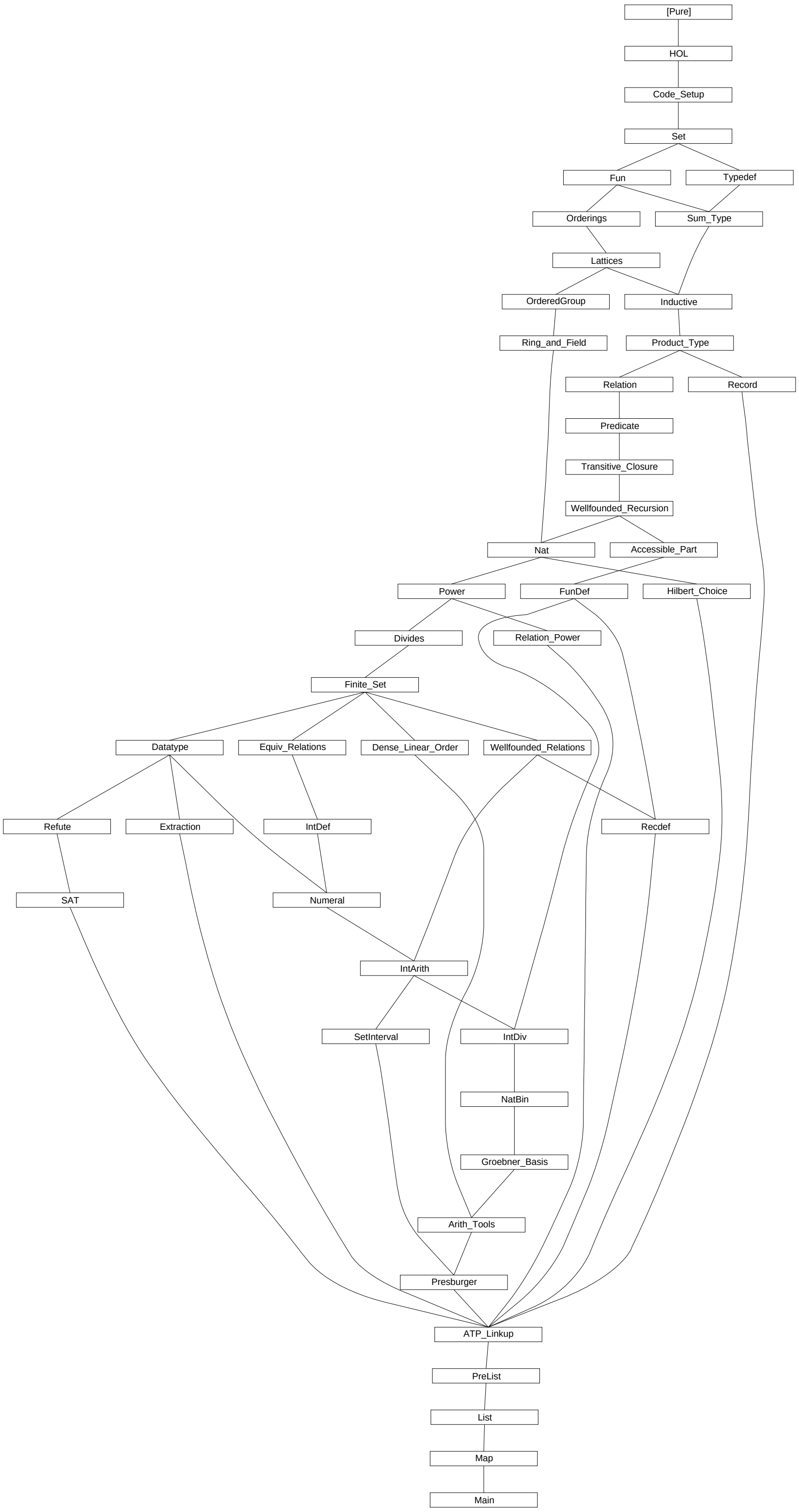
Isabelle/HOL — Higher-Order Logic

November 22, 2007

Contents


[Pure]
HOL
Code_Setup
Set
Fun
Typedef
Orderings
Sum_Type
Lattices
OrderedGroup
Inductive
Ring_and_Field
Product_Type
Relation
Record
Predicate
Transitive_Closure
Wellfounded_Recursion
Nat
Accessible_Part
Power
FunDef
Hilbert_Choice
Divides
Relation_Power
Finite_Set
Datatype
Equiv_Relations
Dense_Linear_Order
Wellfounded_Relations
Refute
Extraction
IntDef
Recdef
SAT
Numeral
IntArith
SetInterval
IntDiv
NatBin
Groebner_Basis
Arith_Tools
Presburger
ATP_Linkup
PreList
List
Map
Main

1 HOL: The basis of Higher-Order Logic

theory *HOL*
imports *CPure*
uses
(*hologic.ML*)
~~/*src*/*Tools*/*IsaPlanner*/*zipper.ML*
~~/*src*/*Tools*/*IsaPlanner*/*isand.ML*
~~/*src*/*Tools*/*IsaPlanner*/*rw-tools.ML*
~~/*src*/*Tools*/*IsaPlanner*/*rw-inst.ML*
~~/*src*/*Provers*/*project-rule.ML*
~~/*src*/*Provers*/*hypsubst.ML*
~~/*src*/*Provers*/*splitter.ML*
~~/*src*/*Provers*/*classical.ML*
~~/*src*/*Provers*/*blast.ML*
~~/*src*/*Provers*/*clasimp.ML*
~~/*src*/*Provers*/*eqsubst.ML*
~~/*src*/*Provers*/*quantifier1.ML*
(*simpdata.ML*)
~~/*src*/*Tools*/*induct.ML*
~~/*src*/*Tools*/*code*/*code-name.ML*
~~/*src*/*Tools*/*code*/*code-funcgr.ML*
~~/*src*/*Tools*/*code*/*code-thingol.ML*
~~/*src*/*Tools*/*code*/*code-target.ML*
~~/*src*/*Tools*/*code*/*code-package.ML*
~~/*src*/*Tools*/*nbe.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*

itself :: (*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*

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

local

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

1.1.2 Additional concrete syntax

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

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

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

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

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

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

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

nonterminals
  letbinds letbind
  case-syn cases-syn

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

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

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

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

print-translation ⟪
(∗ *To avoid eta*−*contraction of body*: ∗)
[(*The*, *fn* [*Abs abs*] =>
    let val (*x*,*t*) = *atomic-abs-tr*′ *abs*
    in Syntax.const -The $ *x* $ *t end*)]
⟫

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

notation (*xsymbols*)
  All (**binder** ∀ *10*) **and**
  Ex (**binder** ∃ *10*) **and**
  Ex1 (**binder** ∃! *10*)

notation (*HTML* **output**)
  All (**binder** ∀ *10*) **and**
  Ex (**binder** ∃ *10*) **and**
  Ex1 (**binder** ∃! *10*)

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

Ex1 (**binder** ?! *10*)

1.1.3 Axioms and basic definitions

axioms
eq-reflection: $(x{=}y) \;{=}{=}{>}\; (x{=}{=}y)$

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

ext: $(!!x{::}'a.\ (f\ x\ {::}'b) = g\ x) \;{=}{=}{>}\; (\%x.\ f\ x) = (\%x.\ g\ x)$
— Extensionality is built into the meta-logic, and this rule expresses a related property. It is an eta-expanded version of the traditional rule, and similar to the ABS rule of HOL

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

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

defs
True-def: $True \;{=}{=}\; ((\%x{::}bool.\ x) = (\%x.\ x))$
All-def: $All(P) \;{=}{=}\; (P = (\%x.\ True))$
Ex-def: $Ex(P) \;{=}{=}\; !Q.\ (!x.\ P\ x \;{-}{-}{>}\; Q) \;{-}{-}{>}\; Q$
False-def: $False \;{=}{=}\; (!P.\ P)$
not-def: $\sim P \;{=}{=}\; P{-}{-}{>}False$
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

axiomatization
undefined :: *'a*

axiomatization where

undefined-fun: *undefined* x = *undefined*

1.1.4 Generic classes and algebraic operations

class *default* = *type* +
  fixes *default* :: $'a$

class *zero* = *type* +
  fixes *zero* :: $'a$ (*0*)

class *one* = *type* +
  fixes *one* :: $'a$ (*1*)

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

class *plus* = *type* +
  fixes *plus* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** + *65*)

class *minus* = *type* +
  fixes *uminus* :: $'a \Rightarrow 'a$ (− - [*81*] *80*)
    and *minus* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** − *65*)

class *times* = *type* +
  fixes *times* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** ∗ *70*)

class *inverse* = *type* +
  fixes *inverse* :: $'a \Rightarrow 'a$
    and *divide* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** '/ *70*)

class *abs* = *type* +
  fixes *abs* :: $'a \Rightarrow 'a$
begin

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

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

end

class *sgn* = *type* +
  fixes *sgn* :: $'a \Rightarrow 'a$

class *ord* = *type* +
  fixes *less-eq* :: $'a \Rightarrow 'a \Rightarrow bool$
    and *less* :: $'a \Rightarrow 'a \Rightarrow bool$
begin

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

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

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

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

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

abbreviation (*input*)
greater (**infix** > *50*) **where**
$x > y \equiv y < x$

definition
Least :: (*'a* ⇒ *bool*) ⇒ *'a* (**binder** *LEAST* *10*) **where**
Least P == (*THE* x. P x ∧ (∀ y. P y ⟶ *less-eq* x y))

end

syntax
-index1 :: *index* ($_1$)
translations
(*index*) $_1$ => (*index*) $\diamond$

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 map tr' [@{*const-syntax HOL.one*}, @{*const-syntax HOL.zero*}] *end*;
⟫ — show types that are presumably too general

1.2 Fundamental rules

1.2.1 Equality

Thanks to Stephan Merz

lemma *subst*:
  assumes *eq*: $s = t$ **and** *p*: $P\ s$
  shows $P\ t$
proof –
  from *eq* **have** *meta*: $s \equiv t$
    by (*rule eq-reflection*)
  from *p* **show** *?thesis*
    by (*unfold meta*)
qed

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

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

lemma *trans*: [| $r=s$; $s=t$ |] ==> $r=t$
  by (*erule subst*)

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

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

lemma *box-equals*: [| $a=b$; $a=c$; $b=d$ |] ==> $c=d$
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*: $P\ a$ ==> $a = b$ ==> $P\ b$
  by (*rule subst*)

1.2.2 Congruence rules for application

lemma *fun-cong*: $(f::'a=>'b) = g$ ==> $f(x)=g(x)$
apply (*erule subst*)
apply (*rule refl*)

done

lemma *arg-cong*: $x=y ==> f(x)=f(y)$
apply (*erule subst*)
apply (*rule refl*)
done

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

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

1.2.3 Equality of booleans – iff

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

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

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

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

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

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

1.2.4 True

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

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

lemma *eqTrueE*: $P = True ==> P$

by (*erule iffD2*) (*rule TrueI*)

1.2.5 Universal quantifier

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

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

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

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

1.2.6 False

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

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

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

1.2.7 Negation

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

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

done

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

lemma *notE*: [| ~*P*; *P* |] ==> *R*
apply (*unfold not-def*)
apply (*erule mp* [*THEN FalseE*])
apply *assumption*
done

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

1.2.8 Implication

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

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

lemma *contrapos-nn*:
assumes *major*: ~*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*
by (*erule contrapos-nn*) (*erule sym*)

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

lemma *rev-contrapos*:

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

1.2.9 Existential quantifier

lemma *exI*: $P\ x$ ==> *EX* x::'a. $P\ x$
apply (*unfold Ex-def*)
apply (*iprover intro*: *allI allE impI mp*)
done

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

1.2.10 Conjunction

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

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

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

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

lemma *context-conjI*:
assumes P P ==> Q **shows** P & Q

by (*iprover intro*: *conjI assms*)

1.2.11 Disjunction

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

lemma *disjI2*: *Q* ==> *P*|*Q*
apply (*unfold or-def*)
apply (*iprover intro*: *allI impI mp*)
done

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

1.2.12 Classical logic

lemma *classical*:
  assumes *prem*: ~*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*)

1.2.13 Unique existence

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

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

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

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

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

1.2.14 THE: definite description operator

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

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

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

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

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

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

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

1.2.15 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
  assumes $\tilde{}\,Q\ ==>\ P$ **shows** $P|Q$
apply (*rule classical*)

apply (*iprover intro*: *assms disjI1 disjI2 notI elim*: *notE*)
done

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

case distinction as a natural deduction rule. Note that $\neg$ *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
lemmas *case-split* = *case-split-thm* [*case-names True False*]

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

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

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

lemma *exCI*:
  assumes *ALL x*. ~*P*(*x*) ==> *P*(*a*)
  shows *EX x*. *P*(*x*)
apply (*rule ccontr*)

```
apply (iprover intro: assms exI allI notI notE [of ∃x. P x])
done
```

1.2.16 Intuitionistic Reasoning

```
lemma impE':
  assumes 1: P --> Q
    and 2: Q ==> R
    and 3: P --> Q ==> P
  shows R
proof -
  from 3 and 1 have P .
  with 1 have Q by (rule impE)
  with 2 show R .
qed

lemma allE':
  assumes 1: ALL x. P x
    and 2: P x ==> 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

lemma TrueE: True ==> P ==> P .
lemma notFalseE: ~ False ==> P ==> P .

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

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

use hologic.ML
```

1.2.17 Atomizing meta-level connectives

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

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

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

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

lemma *atomize-conj* [*atomize*]:
  includes *meta-conjunction-syntax*
  shows (*A* && *B*) == *Trueprop* (*A* & *B*)
proof
  assume *conj*: *A* && *B*
  show *A* & *B*
  proof (*rule conjI*)
    from *conj* **show** *A* **by** (*rule conjunctionD1*)
    from *conj* **show** *B* **by** (*rule conjunctionD2*)
  qed
next
  assume *conj*: *A* & *B*

show *A* && *B*
proof −
from *conj* **show** *A* **..**
from *conj* **show** *B* **..**
qed
qed

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

1.3 Package setup

1.3.1 Classical Reasoner setup

lemma *thin-refl*:
$\bigwedge X.\ [\![\ x{=}x;\ PROP\ W\ ]\!] \Longrightarrow PROP\ W$ **.**

ML ⟪
```
structure Hypsubst = HypsubstFun(
struct
  structure Simplifier = Simplifier
  val dest-eq = HOLogic.dest-eq
  val dest-Trueprop = HOLogic.dest-Trueprop
  val dest-imp = HOLogic.dest-imp
  val eq-reflection = @{thm HOL.eq-reflection}
  val rev-eq-reflection = @{thm HOL.meta-eq-to-obj-eq}
  val imp-intr = @{thm HOL.impI}
  val rev-mp = @{thm HOL.rev-mp}
  val subst = @{thm HOL.subst}
  val sym = @{thm HOL.sym}
  val thin-refl = @{thm thin-refl};
end);
open Hypsubst;

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

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

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

structure ResAtpset = NamedThmsFun(val name = atp val description = ATP
```

rules);

structure ResBlacklist = *NamedThmsFun*(*val name* = *noatp val description* = *Theorems blacklisted for ATP*);
⟫

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

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

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

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

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

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

```
ML ‹ val HOL-cs = @{claset} ›

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

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

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


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

ML ‹
structure Blast = BlastFun
(
  type claset = Classical.claset
  val equality-name = @{const-name op =}
  val not-name = @{const-name Not}
  val notE = @{thm HOL.notE}
  val ccontr = @{thm HOL.ccontr}
  val contr-tac = Classical.contr-tac
  val dup-intr = Classical.dup-intr
  val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
  val claset = Classical.claset
  val rep-cs = Classical.rep-cs
  val cla-modifiers = Classical.cla-modifiers
  val cla-meth' = Classical.cla-meth'
);
val Blast-tac = Blast.Blast-tac;
val blast-tac = Blast.blast-tac;
›

setup Blast.setup
```

1.3.2 Simplifier

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

lemma *simp-thms*:
 shows *not-not*: $(\sim \sim P) = P$
 and *Not-eq-iff*: $((\sim P) = (\sim Q)) = (P = Q)$
 and
  $(P \sim= Q) = (P = (\sim Q))$
  $(P \mid \sim P) = True \quad (\sim P \mid P) = True$
  $(x = x) = True$
 and *not-True-eq-False*: $(\neg\ True) = False$
 and *not-False-eq-True*: $(\neg\ False) = True$
 and
  $(\sim P) \sim= P \;\; P \sim= (\sim P)$
  $(True{=}P) = P$
 and *eq-True*: $(P = True) = P$
 and $(False{=}P) = (\sim P)$
 and *eq-False*: $(P = False) = (\neg\ P)$
 and
  $(True \dashrightarrow P) = P \;\; (False \dashrightarrow P) = True$
  $(P \dashrightarrow True) = True \;\; (P \dashrightarrow P) = True$
  $(P \dashrightarrow False) = (\sim P) \;\; (P \dashrightarrow \sim P) = (\sim P)$
  $(P\ \&\ True) = P \;\; (True\ \&\ P) = P$
  $(P\ \&\ False) = False \;\; (False\ \&\ P) = False$
  $(P\ \&\ P) = P \;\; (P\ \&\ (P\ \&\ Q)) = (P\ \&\ Q)$
  $(P\ \&\ \sim P) = False \quad (\sim P\ \&\ P) = False$
  $(P \mid True) = True \;\; (True \mid P) = True$
  $(P \mid False) = P \;\; (False \mid P) = P$
  $(P \mid P) = P \;\; (P \mid (P \mid Q)) = (P \mid Q)$ **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 *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*: $(a\tilde{}=b) = (b\tilde{}=a)$ **by** *iprover*

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

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

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

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

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

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

lemma *imp-conjR*: $(P \dashrightarrow (Q\&R)) = ((P{-}{-}{>}Q) \;\&\; (P{-}{-}{>}R))$ **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*: $(P \;{-}{-}{>}\; Q \mid R) = (\tilde{}Q \;{-}{-}{>}\; P \;{-}{-}{>}\; R)$ **by** *blast*
lemma *imp-disj-not2*: $(P \;{-}{-}{>}\; Q \mid R) = (\tilde{}R \;{-}{-}{>}\; P \;{-}{-}{>}\; Q)$ **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 *imp-cong*: $(P = P') \;{=}{=}{>}\; (P' \;{=}{=}{>}\; (Q = Q')) \;{=}{=}{>}\; ((P \;{-}{-}{>}\; Q) = (P' \;{-}{-}{>}\; Q'))$
  by *iprover*

lemma *de-Morgan-disj*: $(\tilde{}(P \mid Q)) = (\tilde{}P \;\&\; \tilde{}Q)$ **by** *iprover*
lemma *de-Morgan-conj*: $(\tilde{}(P \;\&\; Q)) = (\tilde{}P \mid \tilde{}Q)$ **by** *blast*
lemma *not-imp*: $(\tilde{}(P \;{-}{-}{>}\; Q)) = (P \;\&\; \tilde{}Q)$ **by** *blast*
lemma *not-iff*: $(P\tilde{}=Q) = (P = (\tilde{}Q))$ **by** *blast*
lemma *disj-not1*: $(\tilde{}P \mid Q) = (P \;{-}{-}{>}\; Q)$ **by** *blast*
lemma *disj-not2*: $(P \mid \tilde{}Q) = (Q \;{-}{-}{>}\; P)$ — 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 *all-not-ex*: $(ALL\ x.\ P\ x) = (\sim\ (EX\ x.\ \sim\ P\ x\ ))$ **by** *blast*

declare *All-def* [*noatp*]

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

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

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

if-then-else rules

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

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

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

lemma *if-not-P*: $\sim P \Longrightarrow (if\ P\ then\ x\ else\ y) = y$
by (*unfold if-def*) *blast*

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

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

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

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

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

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

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

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

let rules for simproc

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

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

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

constdefs
  simp-implies :: [*prop*, *prop*] => *prop* (**infixr** =*simp*=> *1*)
  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*: $PROP\ Q \Longrightarrow PROP\ R$
shows *PROP R*
apply (*rule QR*)
apply (*rule PQ* [*unfolded simp-implies-def*])
apply (*rule P*)
done

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

lemma *uncurry*:
assumes $P \longrightarrow Q \longrightarrow R$
shows $P \wedge Q \longrightarrow R$
using *assms* **by** *blast*

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

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

lemma *all-comm*:

(∀ x y. P x y) = (∀ y x. P x y)
by *blast*

lemma *ex-comm*:
(∃ x y. P x y) = (∃ y x. P x y)
by *blast*

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

setup ⟪
```
  Simplifier.method-setup Splitter.split-modifiers
   #> (fn thy => (change-simpset-of thy (fn - => Simpdata.simpset-simprocs);
thy))
  #> Splitter.setup
  #> Clasimp.setup
  #> EqSubst.setup
```
⟫

Simproc for proving (y = x) == *False* from premise ~(x = y):

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

simproc-setup *let-simp* (Let x f) = ⟪
```
let
  val (f-Let-unfold, x-Let-unfold) =
    let val [(-$(f$x)$-)] = prems-of @{thm Let-unfold}
    in (cterm-of @{theory} f, cterm-of @{theory} x) end
  val (f-Let-folded, x-Let-folded) =
    let val [(-$(f$x)$-)] = prems-of @{thm Let-folded}
    in (cterm-of @{theory} f, cterm-of @{theory} x) end;
  val g-Let-folded =
     let val [(-$-$(g$-))] = prems-of @{thm Let-folded} in cterm-of @{theory} g
```

```
end;

 fun proc - ss ct =
  let
   val ctxt = Simplifier.the-context ss;
   val thy = ProofContext.theory-of ctxt;
   val t = Thm.term-of ct;
   val ([t'], ctxt') = Variable.import-terms false [t] ctxt;
  in Option.map (hd o Variable.export ctxt' ctxt o single)
   (case t' of Const (Let,-) $ x $ f => (* x and f are already in normal form *)
     if is-Free x orelse is-Bound x orelse is-Const x
     then SOME @{thm Let-def}
     else
      let
       val n = case f of (Abs (x,-,-)) => x | - => x;
       val cx = cterm-of thy x;
       val {T=xT,...} = rep-cterm cx;
       val cf = cterm-of thy f;
       val fx-g = Simplifier.rewrite ss (Thm.capply cf cx);
       val (-$-$g) = prop-of fx-g;
       val g' = abstract-over (x,g);
      in (if (g aconv g')
          then
            let
              val rl =
                   cterm-instantiate [(f-Let-unfold,cf),(x-Let-unfold,cx)] @{thm
Let-unfold};
            in SOME (rl OF [fx-g]) end
             else if Term.betapply (f,x) aconv g then NONE (*avoid identity
conversion*)
          else let
                val abs-g'= Abs (n,xT,g');
                val g'x = abs-g'$x;
                val g-g'x = symmetric (beta-conversion false (cterm-of thy g'x));
                val rl = cterm-instantiate
                       [(f-Let-folded,cterm-of thy f),(x-Let-folded,cx),
                        (g-Let-folded,cterm-of thy abs-g')]
                       @{thm Let-folded};
              in SOME (rl OF [transitive fx-g g-g'x])
              end)
      end
   | - => NONE)
  end
in proc end ⟩⟩
```

lemma *True-implies-equals*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
proof
 assume $True \Longrightarrow PROP\ P$

from *this* [*OF TrueI*] **show** *PROP P* **.**
next
assume *PROP P*
then show *PROP P* **.**
qed

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

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

lemmas [*simp*] =
triv-forall-equality
True-implies-equals
if-True
if-False
if-cancel
if-eq-cancel
imp-disjL

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

all-simps
simp-thms

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

ML ⟪ *val HOL-ss* = @{*simpset*} ⟫

Simplifies x assuming c and y assuming c

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

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

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

Prevents simplification of t: much faster

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

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

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

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

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

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

1.3.3 Generic cases and induction

Rule projections:

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

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

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

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

lemma *induct-equal-eq*: (x == y) == *Trueprop* (*induct-equal* x y)
  by (*unfold atomize-eq induct-equal-def*)

lemma *induct-conj-eq*:
  includes *meta-conjunction-syntax*
  shows (A && B) == *Trueprop* (*induct-conj* A B)
  by (*unfold atomize-conj induct-conj-def*)

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

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

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

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

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

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

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

Method setup.

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

setup *Induct.setup*

1.4 Other simple lemmas and lemma duplicates

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

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

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

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

lemma *mk-left-commute*:

fixes f (**infix** $\otimes$ *60*)
assumes a: $\bigwedge x\ y\ z.\ (x \otimes y) \otimes z = x \otimes (y \otimes z)$ **and**
c: $\bigwedge x\ y.\ x \otimes y = y \otimes x$
shows $x \otimes (y \otimes z) = y \otimes (x \otimes z)$
by (*rule trans* [*OF trans* [*OF c a*] *arg-cong* [*OF c*, *of f y*]])

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

lemma *nnf-simps*:
$(\neg(P \wedge Q)) = (\neg\ P \vee \neg\ Q)$ $(\neg\ (P \vee Q)) = (\neg\ P \wedge \neg Q)$ $(P \longrightarrow Q) = (\neg P \vee Q)$
$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg\ Q))$ $(\neg(P = Q)) = ((P \wedge \neg\ Q) \vee (\neg P \wedge Q))$
$(\neg\ \neg(P)) = P$
by *blast+*

1.5 Basic ML bindings

ML ⟪
val FalseE = @{*thm FalseE*}
val Let-def = @{*thm Let-def*}
val TrueI = @{*thm TrueI*}
val allE = @{*thm allE*}
val allI = @{*thm allI*}
val all-dupE = @{*thm all-dupE*}
val arg-cong = @{*thm arg-cong*}
val box-equals = @{*thm box-equals*}
val ccontr = @{*thm ccontr*}
val classical = @{*thm classical*}
val conjE = @{*thm conjE*}
val conjI = @{*thm conjI*}
val conjunct1 = @{*thm conjunct1*}
val conjunct2 = @{*thm conjunct2*}
val disjCI = @{*thm disjCI*}
val disjE = @{*thm disjE*}
val disjI1 = @{*thm disjI1*}
val disjI2 = @{*thm disjI2*}
val eq-reflection = @{*thm eq-reflection*}
val ex1E = @{*thm ex1E*}
val ex1I = @{*thm ex1I*}
val ex1-implies-ex = @{*thm ex1-implies-ex*}
val exE = @{*thm exE*}
val exI = @{*thm exI*}
val excluded-middle = @{*thm excluded-middle*}
val ext = @{*thm ext*}
val fun-cong = @{*thm fun-cong*}
val iffD1 = @{*thm iffD1*}
val iffD2 = @{*thm iffD2*}
val iffI = @{*thm iffI*}
val impE = @{*thm impE*}

val impI = @{*thm impI*}
val meta-eq-to-obj-eq = @{*thm meta-eq-to-obj-eq*}
val mp = @{*thm mp*}
val notE = @{*thm notE*}
val notI = @{*thm notI*}
val not-all = @{*thm not-all*}
val not-ex = @{*thm not-ex*}
val not-iff = @{*thm not-iff*}
val not-not = @{*thm not-not*}
val not-sym = @{*thm not-sym*}
val refl = @{*thm refl*}
val rev-mp = @{*thm rev-mp*}
val spec = @{*thm spec*}
val ssubst = @{*thm ssubst*}
val subst = @{*thm subst*}
val sym = @{*thm sym*}
val trans = @{*thm trans*}
⟫

1.6 Code generator basic setup – see further *Code-Setup.thy*

setup *CodeName.setup #> CodeTarget.setup #> Nbe.setup*

class *eq* (**attach** *op* =) = *type*

code-datatype *True False*

lemma [*code func*]:
  shows *False* ∧ *x* ⟷ *False*
    and *True* ∧ *x* ⟷ *x*
    and *x* ∧ *False* ⟷ *False*
    and *x* ∧ *True* ⟷ *x* **by** *simp-all*

lemma [*code func*]:
  shows *False* ∨ *x* ⟷ *x*
    and *True* ∨ *x* ⟷ *True*
    and *x* ∨ *False* ⟷ *x*
    and *x* ∨ *True* ⟷ *True* **by** *simp-all*

lemma [*code func*]:
  shows ¬ *True* ⟷ *False*
    and ¬ *False* ⟷ *True* **by** (*rule HOL.simp-thms*)+

instance *bool* :: *eq* **..**

lemma [*code func*]:
  shows *False* = *P* ⟷ ¬ *P*
    and *True* = *P* ⟷ *P*
    and *P* = *False* ⟷ ¬ *P*

and $P = True \longleftrightarrow P$ **by** *simp-all*

code-datatype *Trueprop prop*

code-datatype $TYPE('a)$

lemma *Let-case-cert*:
assumes $CASE \equiv (\lambda x.\ Let\ x\ f)$
shows $CASE\ x \equiv f\ x$
using *assms* **by** *simp-all*

lemma *If-case-cert*:
includes *meta-conjunction-syntax*
assumes $CASE \equiv (\lambda b.\ If\ b\ f\ g)$
shows $(CASE\ True \equiv f)$ && $(CASE\ False \equiv g)$
using *assms* **by** *simp-all*

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

1.7 Legacy tactics and ML bindings

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

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

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

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

⟫

end

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

theory *Code-Setup*
imports *HOL*
uses *~~/src/HOL/Tools/recfun-codegen.ML*
begin

2.1 SML code generator setup

setup *RecfunCodegen.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*];
⟫
  prop (*bool*)
attach (*term-of*) ⟪
fun term-of-prop b =
  HOLogic.mk-Trueprop (*if b then HOLogic.true-const else HOLogic.false-const*);
⟫

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

setup ⟪
let

fun eq-codegen thy defs 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

Codegen.add-codegen eq-codegen eq-codegen
end
⟫

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

Evaluation

method-setup *evaluation* = ⟪
Method.no-args (*Method.SIMPLE-METHOD'* (*CONVERSION Codegen.evaluation-conv*
THEN' rtac TrueI))
⟫ *solve goal by evaluation*

2.2 Generic code generator setup

using built-in Haskell equality

code-class *eq*
(*Haskell Eq* **where** *op* = ≡ (==))

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

type bool

lemmas [*code*] = *imp-conv-disj*

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

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

code-const *op* = :: *bool* ⇒ *bool* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

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

code-reserved *SML*
bool true false not

code-reserved *OCaml*
bool not

code generation for undefined as exception

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

Let and If

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

2.3 Evaluation oracle

oracle *eval-oracle* (*term*) = ⟪ *fn thy => fn t =>*
if CodePackage.satisfies thy (*HOLogic.dest-Trueprop t*) []
then t
else HOLogic.Trueprop $ *HOLogic.true-const* (*∗dummy∗*)
⟫

method-setup *eval* = ⟪
let
fun eval-tac thy =
SUBGOAL (*fn* (*t*, *i*) => *rtac* (*eval-oracle thy t*) *i*)
in
Method.ctxt-args (*fn ctxt =>*
Method.SIMPLE-METHOD′ (*eval-tac* (*ProofContext.theory-of ctxt*)))
end
⟫ *solve goal by evaluation*

2.4 Normalization by evaluation

method-setup *normalization* = ⟪
Method.no-args (*Method.SIMPLE-METHOD′*
(*CONVERSION* (*ObjectLogic.judgment-conv Nbe.norm-conv*)
THEN′ resolve-tac [*TrueI*, *refl*]))
⟫ *solve goal by normalization*

end

3 Set: Set theory for higher-order logic

theory *Set*
imports *Code-Setup*
begin

A set in HOL is simply a predicate.

3.1 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
op Int :: *'a set => 'a set => 'a set* (**infixl** *Int 70*)
op Un :: *'a set => 'a set => 'a set* (**infixl** *Un 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
Bex1 :: *'a set => ('a => bool) => bool* — bounded unique existential quantifiers
image :: *('a => 'b) => 'a set => 'b set* (**infixr** ' *90*)
op : :: *'a => 'a set => bool* — membership

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

local

3.2 Additional concrete syntax

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

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

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

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

op : (*op* ∈) **and**
op : ((-/ ∈ -) [*50*, *51*] *50*) **and**
not-mem (*op* ∉) **and**
not-mem ((-/ ∉ -) [*50*, *51*] *50*) **and**
Union (⋃ - [*90*] *90*) **and**
Inter (⋂ - [*90*] *90*)

notation (*HTML* **output**)
op Int (**infixl** ∩ *70*) **and**
op Un (**infixl** ∪ *65*) **and**
op : (*op* ∈) **and**
op : ((-/ ∈ -) [*50*, *51*] *50*) **and**
not-mem (*op* ∉) **and**
not-mem ((-/ ∉ -) [*50*, *51*] *50*)

syntax
@*Finset* :: *args* => *'a set* ({(-)})
@*Coll* :: *pttrn* => *bool* => *'a set* ((*1*{-./ -}))
@*SetCompr* :: *'a* => *idts* => *bool* => *'a set* ((*1*{- |/-./ -}))
@*Collect* :: *idt* => *'a set* => *bool* => *'a set* ((*1*{- :/ -./ -}))
@*INTER1* :: *pttrns* => *'b set* => *'b set* ((*3INT* -./ -) [*0*, *10*] *10*)
@*UNION1* :: *pttrns* => *'b set* => *'b set* ((*3UN* -./ -) [*0*, *10*] *10*)
@*INTER* :: *pttrn* => *'a set* => *'b set* => *'b set* ((*3INT* -:-./ -) [*0*, *10*] *10*)
@*UNION* :: *pttrn* => *'a set* => *'b set* => *'b set* ((*3UN* -:-./ -) [*0*, *10*] *10*)
-*Ball* :: *pttrn* => *'a set* => *bool* => *bool* ((*3ALL* -:-./ -) [*0*, *0*, *10*] *10*)
-*Bex* :: *pttrn* => *'a set* => *bool* => *bool* ((*3EX* -:-./ -) [*0*, *0*, *10*] *10*)
-*Bex1* :: *pttrn* => *'a set* => *bool* => *bool* ((*3EX*! -:-./ -) [*0*, *0*, *10*] *10*)
-*Bleast* :: *id* => *'a set* => *bool* => *'a* ((*3LEAST* -:-./ -) [*0*, *0*, *10*]
10)

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

translations
{*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)
  EX! x:A. P  == Bex1 A (%x. P)
  LEAST x:A. P => LEAST x. x:A & P
```

syntax (*xsymbols*)

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

syntax (*HTML* **output**)

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

syntax (*xsymbols*)

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

syntax (*latex* **output**)

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

instance *set* :: (*type*) *ord*
  subset-def: $A \le B \equiv \forall x{\in}A.\ x \in B$
  psubset-def: $A < B \equiv A \le B \wedge A \ne B$ **..**
lemmas [*code func del*] = *subset-def psubset-def*

abbreviation
  subset :: $'a\ set \Rightarrow\ 'a\ set \Rightarrow bool$ **where**
  subset $\equiv$ *less*

abbreviation
  subset-eq :: $'a\ set \Rightarrow\ 'a\ set \Rightarrow bool$ **where**
  subset-eq $\equiv$ *less-eq*

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

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

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

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

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

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

3.2.1 Bounded quantifiers

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

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

syntax (*HOL* **output**)

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

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

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

print-translation ‹‹
let
  val Type (set-type, -) = @{typ 'a set};
  val All-binder = Syntax.binder-name @{const-syntax All};
  val Ex-binder = Syntax.binder-name @{const-syntax Ex};
  val impl = @{const-syntax op -->};
  val conj = @{const-syntax op &};
  val sbset = @{const-syntax subset};
  val sbset-eq = @{const-syntax subset-eq};

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

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

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

```
in
  [tr' All-binder, tr' Ex-binder]
end
⟫
```

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

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

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

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

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

```
print-translation ⟪
let
  fun btr' syn [A,Abs abs] =
    let val (x,t) = atomic-abs-tr' abs
    in Syntax.const syn $ x $ A $ t end
in
[(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;
»
```

3.3 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)
  Bex1_def:   Bex1 A P    == EX! x. x:A & P(x)

instance set :: (type) minus
  Compl_def:     - A       == {x. ~x:A}
  set_diff_def: A - B      == {x. x:A & ~x:B} ..

lemmas [code func del] = Compl_def set_diff_def

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)}
```

3.4 Lemmas and proof tool setup

3.4.1 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*]

3.4.2 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*]: $ALL\ x{:}A.\ P\ x ==> (P\ x ==> Q) ==> (x\ \tilde{}{:}\ A ==> Q) ==> Q$
  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 ⟪
  fun ball-tac i = *etac* @{*thm ballE*} *i THEN contr-tac* (*i* + *1*)
⟫

Gives better instantiation for bound:

ML-setup ⟪
  change-claset (*fn cs* => *cs 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* --> *P x*) = (*a*:*A* --> *P a*)
by *blast*

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

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

val unfold-ball-tac = *unfold-tac* @{*thms Ball-def*};
fun prove-ball-tac ss = *unfold-ball-tac ss THEN Quantifier1*.*prove-one-point-all-tac*;
val rearrange-ball = *Quantifier1*.*rearrange-ball prove-ball-tac*;
in
val defBEX-regroup = *Simplifier*.*simproc* (*the-context* ())
defined BEX [*EX x*:*A*. *P x* & *Q x*] *rearrange-bex*;
val defBALL-regroup = *Simplifier*.*simproc* (*the-context* ())
defined BALL [*ALL x*:*A*. *P x* --> *Q x*] *rearrange-ball*;

end;

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

3.4.3 Congruence rules

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

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

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

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

3.4.4 Subsets

lemma *subsetI* [*atp*,*intro*!]: (!!x. x:A ==> x:B) ==> $A \subseteq B$
by (*simp add*: *subset-def*)

Map the type *'a set => anything* to just *'a*; for overloading constants whose first argument has type *'a set*.

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

declare *subsetD* [*intro?*] — FIXME

lemma *rev-subsetD*: $c \in A$ ==> $A \subseteq B$ ==> $c \in B$
— 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 ⟪

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

lemma *subsetCE* [*elim*]: $A \subseteq B$ ==> ($c \notin A$ ==> P) ==> ($c \in B$ ==> P) ==> 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 ⟪
fun set-mp-tac i = *etac* @{*thm subsetCE*} *i THEN mp-tac i*
⟫

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

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

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

3.4.5 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*!]: $A \subseteq B$ ==> $B \subseteq A$ ==> $A = B$
— 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*: $A = B$ ==> $A \subseteq B$
by (*simp add*: *subset-refl*)

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

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

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

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

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

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

3.4.6 The universal set – UNIV

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

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

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

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

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

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

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

3.4.7 The empty set

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

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

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

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

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

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

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

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

3.4.8 The Powerset operator – Pow

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

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

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

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

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

3.4.9 Set complement

lemma *Compl-iff* [*simp*]: $(c \in -A) = (c \notin A)$
by (*unfold Compl-def*) *blast*

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

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

3.4.10 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*

3.4.11 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*

3.4.12 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*!]: $c : A - B ==> (c{:}A ==> c^{\sim}{:}B ==> P) ==> P$
by *simp*

3.4.13 Augmenting a set – insert

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

lemma *insertI1*: $a : insert\ a\ B$
by *simp*

lemma *insertI2*: $a : B ==> a : insert\ b\ B$
by *simp*

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

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

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

lemma *set-insert*:
assumes $x \in A$
obtains B **where** $A = insert\ x\ B$ **and** $x \notin B$
proof
from *assms* **show** $A = insert\ x\ (A - \{x\})$ **by** *blast*
next
show $x \notin A - \{x\}$ **by** *blast*
qed

lemma *insert-ident*: $x\ ^{\sim}{:}\ A ==> x\ ^{\sim}{:}\ B ==> (insert\ x\ A = insert\ x\ B) = (A = B)$
by *auto*

3.4.14 Singletons, using insert

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

lemma *singletonD* [*dest*!,*noatp*]: $b : \{a\} ==> b = a$
by *blast*

lemmas *singletonE* = *singletonD* [*elim-format*]

lemma *singleton-iff*: $(b : \{a\}) = (b = a)$
by *blast*

lemma *singleton-inject* [*dest*!]: $\{a\} = \{b\} ==> a = b$
by *blast*

lemma *singleton-insert-inj-eq* [*iff*,*noatp*]:
$(\{b\} = insert\ a\ A) = (a = b\ \&\ A \subseteq \{b\})$
by *blast*

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

lemma *subset-singletonD*: $A \subseteq \{x\} ==> A = \{\} \mid A = \{x\}$
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*: $A - \{x\} \subseteq B ==> x \in A ==> A \subseteq insert\ x\ B$
by *blast*

lemma *doubleton-eq-iff*: $(\{a,b\} = \{c,d\}) = (a{=}c\ \&\ b{=}d \mid a{=}d\ \&\ b{=}c)$
by (*blast elim*: *equalityE*)

3.4.15 Unions of families

UN x:*A*. *B x* is $\bigcup B$ ' *A*.

declare *UNION-def* [*noatp*]

lemma *UN-iff* [*simp*]: (*b*: (*UN x*:*A*. *B x*)) = (*EX x*:*A*. *b*: *B x*)
by (*unfold UNION-def*) *blast*

lemma *UN-I* [*intro*]: *a*:*A* ==> *b*: *B a* ==> *b*: (*UN x*:*A*. *B x*)
— The order of the premises presupposes that *A* is rigid; *b* may be flexible.
by *auto*

lemma *UN-E* [*elim*!]: *b* : (*UN x*:*A*. *B x*) ==> (!!*x*. *x*:*A* ==> *b*: *B x* ==> *R*) ==> *R*
by (*unfold UNION-def*) *blast*

lemma *UN-cong* [*cong*]:
A = *B* ==> (!!*x*. *x*:*B* ==> *C x* = *D x*) ==> (*UN x*:*A*. *C x*) = (*UN x*:*B*. *D x*)
by (*simp add*: *UNION-def*)

3.4.16 Intersections of families

INT x:A. B x is $\bigcap B$ ' A.

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

lemma *INT-I* [*intro*!]: (!!*x*. *x*:*A* ==> *b*: *B x*) ==> *b* : (*INT x*:*A*. *B x*)
by (*unfold INTER-def*) *blast*

lemma *INT-D* [*elim*]: *b* : (*INT x*:*A*. *B x*) ==> *a*:*A* ==> *b*: *B a*
by *auto*

lemma *INT-E* [*elim*]: *b* : (*INT x*:*A*. *B x*) ==> (*b*: *B a* ==> *R*) ==> (*a*~:*A* ==> *R*) ==> *R*
— "Classical" elimination – by the Excluded Middle on $a \in A$.
by (*unfold INTER-def*) *blast*

lemma *INT-cong* [*cong*]:
A = *B* ==> (!!*x*. *x*:*B* ==> *C x* = *D x*) ==> (*INT x*:*A*. *C x*) = (*INT x*:*B*. *D x*)
by (*simp add*: *INTER-def*)

3.4.17 Union

lemma *Union-iff* [*simp*,*noatp*]: (*A* : *Union C*) = (*EX X*:*C*. *A*:*X*)
by (*unfold Union-def*) *blast*

lemma *UnionI* [*intro*]: *X*:*C* ==> *A*:*X* ==> *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* ==> (!!*X*. *A*:*X* ==> *X*:*C* ==> *R*) ==> *R*
by (*unfold Union-def*) *blast*

3.4.18 Inter

lemma *Inter-iff* [*simp*,*noatp*]: (*A* : *Inter C*) = (*ALL X*:*C*. *A*:*X*)
by (*unfold Inter-def*) *blast*

lemma *InterI* [*intro*!]: (!!*X*. *X*:*C* ==> *A*:*X*) ==> *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* ==> *X*:*C* ==> *A*:*X*
by *auto*

lemma *InterE* [*elim*]: A : *Inter* C ==> (X~:C ==> R) ==> (A:X ==> R) ==> R
— "Classical" elimination rule – does not require proving $X \in C$.
by (*unfold Inter-def*) *blast*

Image of a set under a function. Frequently b does not have the syntactic form of $f\ x$.

declare *image-def* [*noatp*]

lemma *image-eqI* [*simp*, *intro*]: $b = f\ x$ ==> x:A ==> b : f'A
by (*unfold image-def*) *blast*

lemma *imageI*: x : A ==> $f\ x$: f ' A
by (*rule image-eqI*) (*rule refl*)

lemma *rev-image-eqI*: x:A ==> $b = f\ x$ ==> 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 ==> (!!x. $b = f\ x$ ==> x:A ==> P) ==> 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. $x \in A$ ==> $f\ x \in B$) ==> 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*: $b = f\ x$ ==> $b \in$ *range* f
by *simp*

lemma *rangeI*: $f\,x \in range\ f$
  by *simp*

lemma *rangeE* [*elim?*]: $b \in range\ (\lambda x.\ f\,x)$ ==> (!!x. $b = f\,x$ ==> P) ==> P
  by *blast*

3.4.19 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 −−> $x = b$) & (~ Q −−> $y = b$))
  by (*rule split-if*)

lemma *split-if-eq2*: (a = (*if Q then x else y*)) = ((Q −−> $a = x$) & (~ Q −−> $a = y$))
  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 −−> $x : b$) & (~ Q −−> $y : b$))
  by (*rule split-if*)

lemma *split-if-mem2*: (a : (*if Q then x else y*)) = ((Q −−> $a : x$) & (~ Q −−> $a : y$))
  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*, @{*thms bspec*})] @ *mksimps-pairs*;
  change-simpset (*fn ss* => *ss setmksimps* (*mksimps mksimps-pairs*));
⟫

3.4.20 The "proper subset" relation

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

lemma *psubsetE* [*elim*!,*noatp*]:
    [|$A \subset B$; [|$A \subseteq B$; ~ ($B \subseteq A$)|] ==> R|] ==> R

by (*unfold psubset-def*) *blast*

lemma *psubset-insert-iff*:
$(A \subset insert\ x\ B) = (if\ x \in B\ then\ A \subset B\ else\ if\ x \in A\ then\ A - \{x\} \subset B\ else\ A \subseteq B)$
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*: $A \subset B ==> A \subseteq B$
by (*simp add*: *psubset-eq*)

lemma *psubset-trans*: $[|\ A \subset B;\ B \subset C\ |] ==> A \subset C$
apply (*unfold psubset-def*)
apply (*auto dest*: *subset-antisym*)
done

lemma *psubsetD*: $[|\ A \subset B;\ c \in A\ |] ==> c \in B$
apply (*unfold psubset-def*)
apply (*auto dest*: *subsetD*)
done

lemma *psubset-subset-trans*: $A \subset B ==> B \subseteq C ==> A \subset C$
by (*auto simp add*: *psubset-eq*)

lemma *subset-psubset-trans*: $A \subseteq B ==> B \subset C ==> A \subset C$
by (*auto simp add*: *psubset-eq*)

lemma *psubset-imp-ex-mem*: $A \subset B ==> \exists\, b.\ b \in (B - A)$
by (*unfold psubset-def*) *blast*

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

lemmas [*symmetric*, *rulify*] = *atomize-ball*
and [*symmetric*, *defn*] = *atomize-ball*

3.5 Further set-theory lemmas

3.5.1 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq insert\ a\ B$
by (*rule subsetI*) (*erule insertI2*)

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

lemma *subset-insert*: $x \notin A$ ==> $(A \subseteq insert\ x\ B) = (A \subseteq B)$
by *blast*

Big Union – least upper bound of a set.

lemma *Union-upper*: $B \in A$ ==> $B \subseteq Union\ A$
by (*iprover intro*: *subsetI UnionI*)

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

General union.

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

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

Big Intersection – greatest lower bound of a set.

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

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

lemma *Inter-greatest*: (!!X. $X \in A$ ==> $C \subseteq X$) ==> $C \subseteq Inter\ A$
by (*iprover intro*: *InterI subsetI dest*: *subsetD*)

lemma *INT-lower*: $a \in A$ ==> $(\bigcap x{\in}A.\ B\ x) \subseteq B\ a$
by *blast*

lemma *INT-greatest*: (!!x. $x \in A$ ==> $C \subseteq B\ x$) ==> $C \subseteq (\bigcap x{\in}A.\ B\ x)$
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*: $A \subseteq C$ ==> $B \subseteq C$ ==> $A \cup B \subseteq C$
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*: $C \subseteq A$ ==> $C \subseteq B$ ==> $C \subseteq A \cap B$
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*

3.5.2 Equalities involving union, intersection, inclusion, etc.

$\{\}$.

lemma *Collect-const* [*simp*]: $\{s.\ P\} = ($*if* P *then* *UNIV* *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*]: $(Collect\ P = \{\}) = (\forall\, x.\ \neg\ P\ x)$
by *blast*

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

lemma *Collect-neg-eq*: $\{x.\ \neg\ P\ x\} = -\ \{x.\ P\ x\}$
by *blast*

lemma *Collect-disj-eq*: $\{x.\ P\ x\ |\ Q\ x\} = \{x.\ P\ x\} \cup \{x.\ Q\ x\}$
by *blast*

lemma *Collect-imp-eq*: $\{x.\ P\ x$ --> $Q\ x\} = -\{x.\ P\ x\} \cup \{x.\ Q\ x\}$
by *blast*

lemma *Collect-conj-eq*: $\{x.\ P\ x\ \&\ Q\ x\} = \{x.\ P\ x\} \cap \{x.\ Q\ x\}$
by *blast*

lemma *Collect-all-eq*: $\{x.\ \forall\, y.\ P\ x\ y\} = (\bigcap\, y.\ \{x.\ P\ x\ y\})$

by *blast*

lemma *Collect-ball-eq*: $\{x.\ \forall\, y{\in}A.\ P\ x\ y\} = (\bigcap\, y{\in}A.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-ex-eq* [*noatp*]: $\{x.\ \exists\, y.\ P\ x\ y\} = (\bigcup\, y.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-bex-eq* [*noatp*]: $\{x.\ \exists\, y{\in}A.\ P\ x\ y\} = (\bigcup\, y{\in}A.\ \{x.\ P\ x\ y\})$
by *blast*

insert.

lemma *insert-is-Un*: $insert\ a\ A = \{a\}\ Un\ A$
— NOT SUITABLE FOR REWRITING since $\{a\}$ == *insert a* {}
by *blast*

lemma *insert-not-empty* [*simp*]: $insert\ a\ A \neq \{\}$
by *blast*

lemmas *empty-not-insert* = *insert-not-empty* [*symmetric*, *standard*]
declare *empty-not-insert* [*simp*]

lemma *insert-absorb*: $a \in A$ ==> $insert\ a\ A = A$
— [*simp*] causes recursive calls when there are nested inserts
— with *quadratic* running time
by *blast*

lemma *insert-absorb2* [*simp*]: $insert\ x\ (insert\ x\ A) = insert\ x\ A$
by *blast*

lemma *insert-commute*: $insert\ x\ (insert\ y\ A) = insert\ y\ (insert\ x\ A)$
by *blast*

lemma *insert-subset* [*simp*]: $(insert\ x\ A \subseteq B) = (x \in B\ \&\ A \subseteq B)$
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*,*noatp*]:
$(insert\ a\ A \cap B = \{\}) = (a \notin B \wedge A \cap B = \{\})$
$(\{\} = insert\ a\ A \cap B) = (a \notin B \wedge \{\} = A \cap B)$
by *auto*

lemma *disjoint-insert* [*simp*,*noatp*]:
$(B \cap insert\ a\ A = \{\}) = (a \notin B \wedge B \cap A = \{\})$
$(\{\} = A \cap insert\ b\ B) = (b \notin A \wedge \{\} = A \cap B)$
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-constant-conv*: $(\%x.\ c)$ ‘ $A = (if\ A = \{\}\ then\ \{\}\ else\ \{c\})$
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* [*noatp*]: f ‘ $\{x.\ P\ x\} = \{f\ x \mid x.\ P\ x\}$
— NOT suitable as a default simprule: the RHS isn't simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.
by *blast*

lemma *if-image-distrib* [*simp*]:
$(\lambda x.\ if\ P\ x\ then\ f\ x\ else\ g\ x)$ ‘ S
$= (f$ ‘ $(S \cap \{x.\ P\ x\})) \cup (g$ ‘ $(S \cap \{x.\ \neg\ P\ x\}))$
by (*auto simp add*: *image-def*)

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

range.

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

lemma *range-composition* [*simp*]: $range\ (\lambda x.\ f\ (g\ x)) = f\text{`}range\ g$
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*: $B \subseteq A ==> A \cap B = B$
by *blast*

lemma *Int-absorb2*: $A \subseteq B ==> A \cap B = A$
by *blast*

lemma *Int-empty-left* [*simp*]: $\{\} \cap B = \{\}$
by *blast*

lemma *Int-empty-right* [*simp*]: $A \cap \{\} = \{\}$
by *blast*

lemma *disjoint-eq-subset-Compl*: $(A \cap B = \{\}) = (A \subseteq -B)$
by *blast*

lemma *disjoint-iff-not-equal*: $(A \cap B = \{\}) = (\forall\, x{\in}A.\ \forall\, y{\in}B.\ x \neq y)$
by *blast*

lemma *Int-UNIV-left* [*simp*]: $UNIV \cap B = B$
by *blast*

lemma *Int-UNIV-right* [*simp*]: $A \cap UNIV = A$

by *blast*

lemma *Int-eq-Inter*: $A \cap B = \bigcap\{A, B\}$
by *blast*

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by *blast*

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
by *blast*

lemma *Int-UNIV* [*simp*,*noatp*]: $(A \cap B = UNIV) = (A = UNIV \ \& \ B = UNIV)$
by *blast*

lemma *Int-subset-iff* [*simp*]: $(C \subseteq A \cap B) = (C \subseteq A \ \& \ C \subseteq B)$
by *blast*

lemma *Int-Collect*: $(x \in A \cap \{x.\ P\ x\}) = (x \in A \ \& \ P\ x)$
by *blast*

Un.

lemma *Un-absorb* [*simp*]: $A \cup A = A$
by *blast*

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
by *blast*

lemma *Un-commute*: $A \cup B = B \cup A$
by *blast*

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$
by *blast*

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
by *blast*

lemmas *Un-ac* = *Un-assoc Un-left-absorb Un-commute Un-left-commute*
— Union is an AC-operator

lemma *Un-absorb1*: $A \subseteq B \implies A \cup B = B$
by *blast*

lemma *Un-absorb2*: $B \subseteq A \implies A \cup B = A$
by *blast*

lemma *Un-empty-left* [*simp*]: $\{\} \cup B = B$
by *blast*

lemma *Un-empty-right* [*simp*]: $A \cup \{\} = A$

by *blast*

lemma *Un-UNIV-left* [*simp*]: $UNIV \cup B = UNIV$
by *blast*

lemma *Un-UNIV-right* [*simp*]: $A \cup UNIV = UNIV$
by *blast*

lemma *Un-eq-Union*: $A \cup B = \bigcup\{A, B\}$
by *blast*

lemma *Un-insert-left* [*simp*]: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
by *blast*

lemma *Un-insert-right* [*simp*]: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
by *blast*

lemma *Int-insert-left*:
$(insert\ a\ B)\ Int\ C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$
by *auto*

lemma *Int-insert-right*:
$A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
by *auto*

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
by *blast*

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
by *blast*

lemma *Un-Int-crazy*:
$(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
by *blast*

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

lemma *Un-empty* [*iff*]: $(A \cup B = \{\}) = (A = \{\}\ \&\ B = \{\})$
by *blast*

lemma *Un-subset-iff* [*simp*]: $(A \cup B \subseteq C) = (A \subseteq C\ \&\ B \subseteq C)$
by *blast*

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
by *blast*

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
by *blast*

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap -A = \{\}$
by *blast*

lemma *Compl-disjoint2* [*simp*]: $-A \cap A = \{\}$
by *blast*

lemma *Compl-partition*: $A \cup -A = UNIV$
by *blast*

lemma *Compl-partition2*: $-A \cup A = UNIV$
by *blast*

lemma *double-complement* [*simp*]: $- (-A) = (A::'a\ set)$
by *blast*

lemma *Compl-Un* [*simp*]: $-(A \cup B) = (-A) \cap (-B)$
by *blast*

lemma *Compl-Int* [*simp*]: $-(A \cap B) = (-A) \cup (-B)$
by *blast*

lemma *Compl-UN* [*simp*]: $-(\bigcup x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ -B\ x)$
by *blast*

lemma *Compl-INT* [*simp*]: $-(\bigcap x{\in}A.\ B\ x) = (\bigcup x{\in}A.\ -B\ x)$
by *blast*

lemma *subset-Compl-self-eq*: $(A \subseteq -A) = (A = \{\})$
by *blast*

lemma *Un-Int-assoc-eq*: $((A \cap B) \cup C = A \cap (B \cup C)) = (C \subseteq A)$
— Halmos, Naive Set Theory, page 16.
by *blast*

lemma *Compl-UNIV-eq* [*simp*]: $-UNIV = \{\}$
by *blast*

lemma *Compl-empty-eq* [*simp*]: $-\{\} = UNIV$
by *blast*

lemma *Compl-subset-Compl-iff* [*iff*]: $(-A \subseteq -B) = (B \subseteq A)$
by *blast*

lemma *Compl-eq-Compl-iff* [*iff*]: $(-A = -B) = (A = (B::'a\ set))$
by *blast*

Union.

lemma *Union-empty* [*simp*]: $Union(\{\}) = \{\}$

by *blast*

lemma *Union-UNIV* [*simp*]: *Union UNIV* = *UNIV*
by *blast*

lemma *Union-insert* [*simp*]: *Union* (*insert a B*) = $a \cup \bigcup B$
by *blast*

lemma *Union-Un-distrib* [*simp*]: $\bigcup(A\ Un\ B) = \bigcup A \cup \bigcup B$
by *blast*

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

lemma *Union-empty-conv* [*simp*,*noatp*]: $(\bigcup A = \{\}) = (\forall x{\in}A.\ x = \{\})$
by *blast*

lemma *empty-Union-conv* [*simp*,*noatp*]: $(\{\} = \bigcup A) = (\forall x{\in}A.\ x = \{\})$
by *blast*

lemma *Union-disjoint*: $(\bigcup C \cap A = \{\}) = (\forall B{\in}C.\ B \cap A = \{\})$
by *blast*

Inter.

lemma *Inter-empty* [*simp*]: $\bigcap\{\} = UNIV$
by *blast*

lemma *Inter-UNIV* [*simp*]: $\bigcap UNIV = \{\}$
by *blast*

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

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap(A \cap B)$
by *blast*

lemma *Inter-Un-distrib*: $\bigcap(A \cup B) = \bigcap A \cap \bigcap B$
by *blast*

lemma *Inter-UNIV-conv* [*simp*,*noatp*]:
$(\bigcap A = UNIV) = (\forall x{\in}A.\ x = UNIV)$
$(UNIV = \bigcap A) = (\forall x{\in}A.\ x = UNIV)$
by *blast+*

UN and *INT*.

Basic identities:

lemma *UN-empty* [*simp*,*noatp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
by *blast*

lemma *UN-empty2* [*simp*]: $(\bigcup x{\in}A.\ \{\}) = \{\}$
by *blast*

lemma *UN-singleton* [*simp*]: $(\bigcup x{\in}A.\ \{x\}) = A$
by *blast*

lemma *UN-absorb*: $k \in I \text{ ==> } A\ k \cup (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ A\ i)$
by *auto*

lemma *INT-empty* [*simp*]: $(\bigcap x{\in}\{\}.\ B\ x) = UNIV$
by *blast*

lemma *INT-absorb*: $k \in I \text{ ==> } A\ k \cap (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ A\ i)$
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*[*simp*]: $(\bigcup i \in A \cup B.\ M\ i) = (\bigcup i{\in}A.\ M\ i) \cup (\bigcup i{\in}B.\ M\ i)$
by *blast*

lemma *UN-UN-flatten*: $(\bigcup x \in (\bigcup y{\in}A.\ B\ y).\ C\ x) = (\bigcup y{\in}A.\ \bigcup x{\in}B\ y.\ C\ x)$
by *blast*

lemma *UN-subset-iff*: $((\bigcup i{\in}I.\ A\ i) \subseteq B) = (\forall i{\in}I.\ A\ i \subseteq B)$
by *blast*

lemma *INT-subset-iff*: $(B \subseteq (\bigcap i{\in}I.\ A\ i)) = (\forall i{\in}I.\ B \subseteq A\ i)$
by *blast*

lemma *INT-insert* [*simp*]: $(\bigcap x \in insert\ a\ A.\ B\ x) = B\ a \cap INTER\ A\ B$
by *blast*

lemma *INT-Un*: $(\bigcap i \in A \cup B.\ M\ i) = (\bigcap i \in A.\ M\ i) \cap (\bigcap i{\in}B.\ M\ i)$
by *blast*

lemma *INT-insert-distrib*:
$u \in A \text{ ==> } (\bigcap x{\in}A.\ insert\ a\ (B\ x)) = insert\ a\ (\bigcap x{\in}A.\ B\ x)$
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*]: $(\bigcup y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \{\}\ \mathit{else}\ c)$
  by *auto*

lemma *INT-constant* [*simp*]: $(\bigcap y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ c)$
  by *auto*

lemma *UN-eq*: $(\bigcup x{\in}A.\ B\ x) = \bigcup(\{Y.\ \exists x{\in}A.\ Y = B\ x\})$
  by *blast*

lemma *INT-eq*: $(\bigcap x{\in}A.\ B\ x) = \bigcap(\{Y.\ \exists x{\in}A.\ Y = B\ x\})$
  — Look: it has an *existential* quantifier
  by *blast*

lemma *UNION-empty-conv*[*simp*]:
  $(\{\} = (\mathit{UN}\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = \{\})$
  $((\mathit{UN}\ x{:}A.\ B\ x) = \{\}) = (\forall x{\in}A.\ B\ x = \{\})$
by *blast+*

lemma *INTER-UNIV-conv*[*simp*]:
 $(\mathit{UNIV} = (\mathit{INT}\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = \mathit{UNIV})$
 $((\mathit{INT}\ x{:}A.\ B\ x) = \mathit{UNIV}) = (\forall x{\in}A.\ B\ x = \mathit{UNIV})$
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*: $(\forall z \in \mathit{UNION}\ A\ B.\ P\ z) = (\forall x{\in}A.\ \forall z \in B\ x.\ P\ z)$
by *blast*

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

Set difference.

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

lemma *Diff-eq-empty-iff* [*simp*,*noatp*]: $(A - B = \{\}) = (A \subseteq B)$
by *blast*

lemma *Diff-cancel* [*simp*]: $A - A = \{\}$
by *blast*

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - (B{::}'a\ set)$
by *blast*

lemma *Diff-triv*: $A \cap B = \{\}$ ==> $A - B = A$
by (*blast elim*: *equalityE*)

lemma *empty-Diff* [*simp*]: $\{\} - A = \{\}$

by *blast*

lemma *Diff-empty* [*simp*]: $A - \{\} = A$
by *blast*

lemma *Diff-UNIV* [*simp*]: $A - UNIV = \{\}$
by *blast*

lemma *Diff-insert0* [*simp*,*noatp*]: $x \notin A \implies A - insert\ x\ B = A - B$
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*]: $x \in B \implies insert\ x\ A - B = A - B$
by *blast*

lemma *insert-Diff-single*[*simp*]: $insert\ a\ (A - \{a\}) = insert\ a\ A$
by *blast*

lemma *insert-Diff*: $a \in A \implies insert\ a\ (A - \{a\}) = A$
by *blast*

lemma *Diff-insert-absorb*: $x \notin A \implies (insert\ x\ A) - \{x\} = A$
by *auto*

lemma *Diff-disjoint* [*simp*]: $A \cap (B - A) = \{\}$
by *blast*

lemma *Diff-partition*: $A \subseteq B \implies A \cup (B - A) = B$
by *blast*

lemma *double-diff*: $A \subseteq B \implies B \subseteq C \implies B - (C - A) = A$
by *blast*

lemma *Un-Diff-cancel* [*simp*]: $A \cup (B - A) = A \cup B$
by *blast*

lemma *Un-Diff-cancel2* [*simp*]: $(B - A) \cup A = B \cup A$
by *blast*

lemma *Diff-Un*: $A - (B \cup C) = (A - B) \cap (A - C)$
by *blast*

lemma *Diff-Int*: $A - (B \cap C) = (A - B) \cup (A - C)$
by *blast*

lemma *Un-Diff*: $(A \cup B) - C = (A - C) \cup (B - C)$
by *blast*

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

lemma *Diff-Int-distrib*: $C \cap (A - B) = (C \cap A) - (C \cap B)$
by *blast*

lemma *Diff-Int-distrib2*: $(A - B) \cap C = (A \cap C) - (B \cap C)$
by *blast*

lemma *Diff-Compl* [*simp*]: $A - (- B) = A \cap B$
by *auto*

lemma *Compl-Diff-eq* [*simp*]: $- (A - B) = -A \cup B$
by *blast*

Quantification over type *bool*.

lemma *bool-induct*: $P\ True \Longrightarrow P\ False \Longrightarrow P\ x$
by (*cases x*) *auto*

lemma *all-bool-eq*: $(\forall b.\ P\ b) \longleftrightarrow P\ True \wedge P\ False$
by (*auto intro*: *bool-induct*)

lemma *bool-contrapos*: $P\ x \Longrightarrow \neg\ P\ False \Longrightarrow P\ True$
by (*cases x*) *auto*

lemma *ex-bool-eq*: $(\exists b.\ P\ b) \longleftrightarrow P\ True \vee P\ False$
by (*auto intro*: *bool-contrapos*)

lemma *Un-eq-UN*: $A \cup B = (\bigcup b.\ if\ b\ then\ A\ else\ B)$
by (*auto simp add*: *split-if-mem2*)

lemma *UN-bool-eq*: $(\bigcup b::bool.\ A\ b) = (A\ True \cup A\ False)$
by (*auto intro*: *bool-contrapos*)

lemma *INT-bool-eq*: $(\bigcap b::bool.\ A\ b) = (A\ True \cap A\ False)$
by (*auto intro*: *bool-induct*)

Pow

lemma *Pow-empty* [*simp*]: $Pow\ \{\} = \{\{\}\}$
by (*auto simp add*: *Pow-def*)

lemma *Pow-insert*: $Pow\ (insert\ a\ A) = Pow\ A \cup (insert\ a\ `\ Pow\ A)$
by (*blast intro*: *image-eqI* [**where** $?x = u - \{a\}$, *standard*])

lemma *Pow-Compl*: $Pow\ (-\ A) = \{-B \mid B.\ A \in Pow\ B\}$
by (*blast intro*: *exI* [**where** $?x = -\ u$, *standard*])

lemma *Pow-UNIV* [*simp*]: $Pow\ UNIV = UNIV$
by *blast*

lemma *Un-Pow-subset*: $Pow\ A \cup Pow\ B \subseteq Pow\ (A \cup B)$
by *blast*

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

lemma *subset-Pow-Union*: $A \subseteq Pow\ (\bigcup A)$
by *blast*

lemma *Union-Pow-eq* [*simp*]: $\bigcup(Pow\ A) = A$
by *blast*

lemma *Pow-Int-eq* [*simp*]: $Pow\ (A \cap B) = Pow\ A \cap Pow\ B$
by *blast*

lemma *Pow-INT-eq*: $Pow\ (\bigcap x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ Pow\ (B\ x))$
by *blast*

Miscellany.

lemma *set-eq-subset*: $(A = B) = (A \subseteq B\ \&\ B \subseteq A)$
by *blast*

lemma *subset-iff*: $(A \subseteq B) = (\forall t.\ t \in A\ {-}{-}{>}\ t \in B)$
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* [*simp*]: $(\forall x.\ x \notin A) = (A = \{\})$
by *blast*

lemma *ex-in-conv*: $(\exists x.\ x \in A) = (A \neq \{\})$
by *blast*

lemma *distinct-lemma*: $f\ x \neq f\ y\ {=}{=}{>}\ x \neq y$
by *iprover*

Miniscoping: pushing in quantifiers and big Unions and Intersections.

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

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

lemma *ball-simps* [*simp*,*noatp*]:
!!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. (~(ALL x:A. P x)) = (EX x:A. ~P x)
by *auto*

lemma *bex-simps* [*simp*,*noatp*]:
!!*A P Q*. (*EX x*:*A*. *P x* & *Q*) = ((*EX x*:*A*. *P x*) & *Q*)
!!*A P Q*. (*EX x*:*A*. *P* & *Q x*) = (*P* & (*EX x*:*A*. *Q x*))
!!*P*. (*EX x*:{}. *P x*) = *False*
!!*P*. (*EX x*:*UNIV*. *P x*) = (*EX x*. *P x*)
!!*a B P*. (*EX x*:*insert a B*. *P x*) = (*P*(*a*) | (*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*. (~(*EX x*:*A*. *P x*)) = (*ALL x*:*A*. ~*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*

3.5.3 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \implies f`A \subseteq f`B$
  by *blast*

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

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

lemma *Inter-anti-mono*: $B \subseteq A \implies \bigcap A \subseteq \bigcap B$
  by *blast*

lemma *UN-mono*:
  $A \subseteq B \implies (!!x.\ x \in A \implies f\ x \subseteq g\ x) \implies$
    $(\bigcup x{\in}A.\ f\ x) \subseteq (\bigcup x{\in}B.\ g\ x)$
  by (*blast dest*: *subsetD*)

lemma *INT-anti-mono*:
  $B \subseteq A \implies (!!x.\ x \in A \implies f\ x \subseteq g\ x) \implies$
    $(\bigcap x{\in}A.\ f\ x) \subseteq (\bigcap x{\in}A.\ g\ x)$
  — The last inclusion is POSITIVE!
  by (*blast dest*: *subsetD*)

lemma *insert-mono*: $C \subseteq D \implies insert\ a\ C \subseteq insert\ a\ D$
  by *blast*

lemma *Un-mono*: $A \subseteq C \implies B \subseteq D \implies A \cup B \subseteq C \cup D$
  by *blast*

lemma *Int-mono*: $A \subseteq C \implies B \subseteq D \implies A \cap B \subseteq C \cap D$
  by *blast*

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

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

Monotonicity of implications.

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

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

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

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

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

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

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

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

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

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

lemma *eq-to-mono*: $a = b \Longrightarrow c = d \Longrightarrow b \dashrightarrow d \Longrightarrow a \dashrightarrow c$
by *iprover*

lemma *eq-to-mono2*: $a = b \Longrightarrow c = d \Longrightarrow \sim b \dashrightarrow \sim d \Longrightarrow \sim a \dashrightarrow \sim c$
by *iprover*

3.6 Inverse image of a function

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

3.6.1 Basic rules

lemma *vimage-eq* [*simp*]: $(a : f\ -`\ B) = (f\ a : B)$
by (*unfold vimage-def*) *blast*

lemma *vimage-singleton-eq*: $(a : f\ -`\ \{b\}) = (f\ a = b)$
by *simp*

lemma *vimageI* [*intro*]: $f\ a = b \Longrightarrow b{:}B \Longrightarrow a : f\ -`\ B$

by (*unfold vimage-def*) *blast*

lemma *vimageI2*: $f\ a : A \Longrightarrow a : f\ -`\ A$
by (*unfold vimage-def*) *fast*

lemma *vimageE* [*elim!*]: $a: f\ -`\ B \Longrightarrow (!!x.\ f\ a = x \Longrightarrow x{:}B \Longrightarrow P) \Longrightarrow P$
by (*unfold vimage-def*) *blast*

lemma *vimageD*: $a : f\ -`\ A \Longrightarrow f\ a : A$
by (*unfold vimage-def*) *fast*

3.6.2 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*: $(!!x.\ P\ (f\ x) = Q\ x) \Longrightarrow f\ -`\ (Collect\ P) = Collect\ Q$
by *blast*

lemma *vimage-insert*: $f-`(insert\ a\ B) = (f-`\{a\})\ Un\ (f-`B)$
— NOT suitable for rewriting because of the recurrence of $\{a\}$.
by *blast*

lemma *vimage-Diff*: $f\ -`\ (A - B) = (f\ -`\ A) - (f\ -`\ B)$
by *blast*

lemma *vimage-UNIV* [*simp*]: $f\ -`\ UNIV = UNIV$

by *blast*

lemma *vimage-eq-UN*: $f-`B = (UN\ y: B.\ f-`\{y\})$
— NOT suitable for rewriting
by *blast*

lemma *vimage-mono*: $A \subseteq B ==> f\ -`\ A \subseteq f\ -`\ B$
— monotonicity
by *blast*

3.7 Getting the Contents of a Singleton Set

definition
contents :: $'a\ set \Rightarrow\ 'a$
where
[*code func del*]: *contents* $X = (THE\ x.\ X = \{x\})$

lemma *contents-eq* [*simp*]: *contents* $\{x\} = x$
by (*simp add*: *contents-def*)

3.8 Transitivity rules for calculational reasoning

lemma *set-rev-mp*: $x{:}A ==> A \subseteq B ==> x{:}B$
by (*rule subsetD*)

lemma *set-mp*: $A \subseteq B ==> x{:}A ==> x{:}B$
by (*rule subsetD*)

3.9 Code generation for finite sets

code-datatype {} *insert*

3.9.1 Primitive predicates

definition
is-empty :: $'a\ set \Rightarrow bool$
where
[*code func del*]: *is-empty* $A \longleftrightarrow A = \{\}$
lemmas [*code inline*] = *is-empty-def* [*symmetric*]

lemma *is-empty-insert* [*code func*]:
is-empty (*insert a A*) $\longleftrightarrow$ *False*
by (*simp add*: *is-empty-def*)

lemma *is-empty-empty* [*code func*]:
is-empty {} $\longleftrightarrow$ *True*
by (*simp add*: *is-empty-def*)

lemma *Ball-insert* [*code func*]:
Ball (*insert a A*) $P \longleftrightarrow P\ a \wedge Ball\ A\ P$

by *simp*

lemma *Ball-empty* [*code func*]:
$Ball\ \{\}\ P \longleftrightarrow True$
by *simp*

lemma *Bex-insert* [*code func*]:
$Bex\ (insert\ a\ A)\ P \longleftrightarrow P\ a \vee Bex\ A\ P$
by *simp*

lemma *Bex-empty* [*code func*]:
$Bex\ \{\}\ P \longleftrightarrow False$
by *simp*

3.9.2 Primitive operations

lemma *minus-insert* [*code func*]:
$insert\ (a::'a::eq)\ A - B = (let\ C = A - B\ in\ if\ a \in B\ then\ C\ else\ insert\ a\ C)$
by (*auto simp add*: *Let-def*)

lemma *minus-empty1* [*code func*]:
$\{\} - A = \{\}$
by *simp*

lemma *minus-empty2* [*code func*]:
$A - \{\} = A$
by *simp*

lemma *inter-insert* [*code func*]:
$insert\ a\ A \cap B = (let\ C = A \cap B\ in\ if\ a \in B\ then\ insert\ a\ C\ else\ C)$
by (*auto simp add*: *Let-def*)

lemma *inter-empty1* [*code func*]:
$\{\} \cap A = \{\}$
by *simp*

lemma *inter-empty2* [*code func*]:
$A \cap \{\} = \{\}$
by *simp*

lemma *union-insert* [*code func*]:
$insert\ a\ A \cup B = (let\ C = A \cup B\ in\ if\ a \in B\ then\ C\ else\ insert\ a\ C)$
by (*auto simp add*: *Let-def*)

lemma *union-empty1* [*code func*]:
$\{\} \cup A = A$
by *simp*

lemma *union-empty2* [*code func*]:

$A \cup \{\} = A$
by *simp*

lemma *INTER-insert* [*code func*]:
$INTER\ (insert\ a\ A)\ f = f\ a \cap INTER\ A\ f$
by *auto*

lemma *INTER-singleton* [*code func*]:
$INTER\ \{a\}\ f = f\ a$
by *auto*

lemma *UNION-insert* [*code func*]:
$UNION\ (insert\ a\ A)\ f = f\ a \cup UNION\ A\ f$
by *auto*

lemma *UNION-empty* [*code func*]:
$UNION\ \{\}\ f = \{\}$
by *auto*

lemma *contents-insert* [*code func*]:
$contents\ (insert\ a\ A) = contents\ (insert\ a\ (A - \{a\}))$
by *auto*
declare *contents-eq* [*code func*]

3.9.3 Derived predicates

lemma *in-code* [*code func*]:
$a \in A \longleftrightarrow (\exists x \in A.\ a = x)$
by *simp*

instance *set* :: (*eq*) *eq* **..**

lemma *eq-set-code* [*code func*]:
fixes $A\ B$ :: $'a::eq\ set$
shows $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
by *auto*

lemma *subset-eq-code* [*code func*]:
fixes $A\ B$ :: $'a::eq\ set$
shows $A \subseteq B \longleftrightarrow (\forall x \in A.\ x \in B)$
by *auto*

lemma *subset-code* [*code func*]:
fixes $A\ B$ :: $'a::eq\ set$
shows $A \subset B \longleftrightarrow A \subseteq B \wedge \neg\ B \subseteq A$
by *auto*

3.9.4 Derived operations

lemma *image-code* [*code func*]:

image f A = *UNION A* (λx. {*f x*}) **by** *auto*

definition
project :: (*'a* ⇒ *bool*) ⇒ *'a set* ⇒ *'a set* **where**
[*code func del*, *code post*]: *project P A* = {*a*∈*A*. *P a*}

lemmas [*symmetric*, *code inline*] = *project-def*

lemma *project-code* [*code func*]:
project P A = *UNION A* (λa. *if P a then* {*a*} *else* {})
by (*auto simp add*: *project-def split*: *if-splits*)

lemma *Inter-code* [*code func*]:
Inter A = *INTER A* (λx. *x*)
by *auto*

lemma *Union-code* [*code func*]:
Union A = *UNION A* (λx. *x*)
by *auto*

code-reserved *SML union inter*

3.10 Basic ML bindings

ML ⟪
val Ball-def = @{*thm Ball-def*}
val Bex-def = @{*thm Bex-def*}
val CollectD = @{*thm CollectD*}
val CollectE = @{*thm CollectE*}
val CollectI = @{*thm CollectI*}
val Collect-conj-eq = @{*thm Collect-conj-eq*}
val Collect-mem-eq = @{*thm Collect-mem-eq*}
val IntD1 = @{*thm IntD1*}
val IntD2 = @{*thm IntD2*}
val IntE = @{*thm IntE*}
val IntI = @{*thm IntI*}
val Int-Collect = @{*thm Int-Collect*}
val UNIV-I = @{*thm UNIV-I*}
val UNIV-witness = @{*thm UNIV-witness*}
val UnE = @{*thm UnE*}
val UnI1 = @{*thm UnI1*}
val UnI2 = @{*thm UnI2*}
val ballE = @{*thm ballE*}
val ballI = @{*thm ballI*}
val bexCI = @{*thm bexCI*}
val bexE = @{*thm bexE*}
val bexI = @{*thm bexI*}
val bex-triv = @{*thm bex-triv*}
val bspec = @{*thm bspec*}

```
val contra-subsetD = @{thm contra-subsetD}
val distinct-lemma = @{thm distinct-lemma}
val eq-to-mono = @{thm eq-to-mono}
val eq-to-mono2 = @{thm eq-to-mono2}
val equalityCE = @{thm equalityCE}
val equalityD1 = @{thm equalityD1}
val equalityD2 = @{thm equalityD2}
val equalityE = @{thm equalityE}
val equalityI = @{thm equalityI}
val imageE = @{thm imageE}
val imageI = @{thm imageI}
val image-Un = @{thm image-Un}
val image-insert = @{thm image-insert}
val insert-commute = @{thm insert-commute}
val insert-iff = @{thm insert-iff}
val mem-Collect-eq = @{thm mem-Collect-eq}
val rangeE = @{thm rangeE}
val rangeI = @{thm rangeI}
val range-eqI = @{thm range-eqI}
val subsetCE = @{thm subsetCE}
val subsetD = @{thm subsetD}
val subsetI = @{thm subsetI}
val subset-refl = @{thm subset-refl}
val subset-trans = @{thm subset-trans}
val vimageD = @{thm vimageD}
val vimageE = @{thm vimageE}
val vimageI = @{thm vimageI}
val vimageI2 = @{thm vimageI2}
val vimage-Collect = @{thm vimage-Collect}
val vimage-Int = @{thm vimage-Int}
val vimage-Un = @{thm vimage-Un}
»
```

end

4 Fun: Notions about functions

theory *Fun*
imports *Set*
begin

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

definition
  override-on :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b$
where
  override-on f g A = $(\lambda a.\ if\ a \in A\ then\ g\ a\ else\ f\ a)$

definition
  id :: $'a \Rightarrow 'a$
where
  id = $(\lambda x.\ x)$

definition
  comp :: $('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** *o 55*)
where
  f o g = $(\lambda x.\ f\ (g\ x))$

notation (*xsymbols*)
  comp (**infixl** ∘ *55*)

notation (*HTML* **output**)
  comp (**infixl** ∘ *55*)

compatibility

lemmas *o-def* = *comp-def*

constdefs
  inj-on :: ['a => 'b, 'a set] => bool
  inj-on f A == ! x:A. ! y:A. f(x)=f(y) --> x=y

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

abbreviation
  inj f == *inj-on f UNIV*

constdefs
  surj :: ('a => 'b) => bool
  surj f == ! y. ? x. y=f(x)

  bij :: ('a => 'b) => bool
  bij f == *inj f* & *surj f*

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

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

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

4.1 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*)

4.2 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* $\Longrightarrow$ *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*)

4.3 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* $\Longrightarrow$ *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 $]\!]$ $\Longrightarrow$ *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

4.4 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

4.5 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*)

4.6 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* [*noatp*]: *f* −‘ (*f* ‘ *A*) = {*y*. *EX x*:*A*. *f x* = *f y*}
by (*blast intro*: *sym*)

lemma *image-vimage-subset*: *f* ‘ (*f* −‘ *A*) <= *A*
by *blast*

lemma *image-vimage-eq* [*simp*]: *f* ‘ (*f* −‘ *A*) = *A Int range f*
by *blast*

lemma *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

4.7 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*: $a \ \tilde{}= c \ ==> (m(a:=b))(c:=d) = (m(c:=d))(a:=b)$
by (*rule ext*, *auto*)

lemma *inj-on-fun-updI*: $[\![$ *inj-on f A*; $y \notin f`A$ $]\!] \Longrightarrow$ *inj-on* $(f(x:=y))$ A
by(*fastsimp simp*:*inj-on-def image-def*)

lemma *fun-upd-image*:
$f(x:=y)$ ` $A =$ (*if* $x \in A$ *then insert* $y$ $(f$ ` $(A-\{x\}))$ *else* f ` $A)$
by *auto*

4.8 *override-on*

lemma *override-on-emptyset*[*simp*]: *override-on f g* $\{\} = f$
by(*simp add*:*override-on-def*)

lemma *override-on-apply-notin*[*simp*]: $a \ \tilde{}: A ==>$ (*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*)

4.9 swap

definition
swap :: $'a \Rightarrow 'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$
where
swap $a\ b\ f = f\ (a := f\ b,\ b:= f\ a)$

lemma *swap-self*: *swap* $a\ a\ f = f$
by (*simp add*: *swap-def*)

lemma *swap-commute*: *swap* $a\ b\ f =$ *swap* $b\ a\ f$
by (*rule ext*, *simp add*: *fun-upd-def swap-def*)

lemma *swap-nilpotent* [*simp*]: *swap* $a\ b$ (*swap* $a\ b\ f$) $= f$
by (*rule ext*, *simp add*: *fun-upd-def swap-def*)

lemma *inj-on-imp-inj-on-swap*:
[|*inj-on f A*; $a \in A$; $b \in A$|] ==> *inj-on* (*swap* $a\ b\ f$) A
by (*simp add*: *inj-on-def swap-def*, *blast*)

lemma *inj-on-swap-iff* [*simp*]:
assumes A: $a \in A$ $b \in A$ **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*)

4.10 Proof tool setup

simplifies terms of the form f(...,x:=y,...,x:=z,...) to f(...,x:=z,...)

simproc-setup *fun-upd2* (*f*(*v* := *w*, *x* := *y*)) = ⟨⟨ *fn* - =>
let
  fun gen-fun-upd NONE T - - = *NONE*
    | *gen-fun-upd* (*SOME f*) *T x y* = *SOME* (*Const* (@{*const-name fun-upd*}, *T*)
$ *f* $ *x* $ *y*)
  fun dest-fun-T1 (*Type* (-, *T* :: *Ts*)) = *T*
  fun find-double (*t as Const* (@{*const-name fun-upd*},*T*) $ *f* $ *x* $ *y*) =
    let
      fun find (*Const* (@{*const-name fun-upd*},*T*) $ *g* $ *v* $ *w*) =
          if v aconv x then SOME g else gen-fun-upd (*find g*) *T v w*
        | *find t* = *NONE*
    in (*dest-fun-T1 T*, *gen-fun-upd* (*find f*) *T x y*) *end*

  fun proc ss ct =
    let
      val ctxt = *Simplifier.the-context ss*
      val t = *Thm.term-of ct*
    in
      case find-double t of
        (*T*, *NONE*) => *NONE*
      | (*T*, *SOME rhs*) =>

```
        SOME (Goal.prove ctxt [] [] (Term.equals T $ t $ rhs)
          (fn - =>
            rtac eq-reflection 1 THEN
            rtac ext 1 THEN
            simp-tac (Simplifier.inherit-context ss @{simpset}) 1))
    end
in proc end
»
```

4.11 Code generator setup

code-const *op* ∘
  (*SML* **infixl** *5 o*)
  (*Haskell* **infixr** *9* .)

code-const *id*
  (*Haskell id*)

4.12 ML legacy bindings

ML «
val set-cs = claset() delrules [equalityI]
»

ML «
```
val id-apply = @{thm id-apply}
val id-def = @{thm id-def}
val o-apply = @{thm o-apply}
val o-assoc = @{thm o-assoc}
val o-def = @{thm o-def}
val injD = @{thm injD}
val datatype-injI = @{thm datatype-injI}
val range-ex1-eq = @{thm range-ex1-eq}
val expand-fun-eq = @{thm expand-fun-eq}
```
»

end

5 Orderings: Syntactic and abstract orders

theory *Orderings*
imports *Set Fun*
uses
  ~~/src/Provers/order.ML
begin

5.1 Partial orders

class *order* = *ord* +
  assumes *less-le*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$
  and *order-refl* [*iff*]: $x \leq x$
  and *order-trans*: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
  assumes *antisym*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
begin

Reflexivity.

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

lemma *less-irrefl* [*iff*]: $\neg\ x < x$
by (*simp add*: *less-le*)

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

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

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

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

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

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

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

Transitivity rules for calculational reasoning

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

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

Asymmetry.

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

lemma *less-asym*: $x < y \Longrightarrow (\neg\ P \Longrightarrow y < x) \Longrightarrow P$

by (*drule less-not-sym*, *erule contrapos-np*) *simp*

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

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

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

Transitivity.

lemma *less-trans*: $x < y \Longrightarrow y < z \Longrightarrow x < z$
by (*simp add*: *less-le*) (*blast intro*: *order-trans antisym*)

lemma *le-less-trans*: $x \leq y \Longrightarrow y < z \Longrightarrow x < z$
by (*simp add*: *less-le*) (*blast intro*: *order-trans antisym*)

lemma *less-le-trans*: $x < y \Longrightarrow y \leq z \Longrightarrow x < z$
by (*simp add*: *less-le*) (*blast intro*: *order-trans antisym*)

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

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

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

Transitivity rules for calculational reasoning

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

Reverse order

lemma *order-reverse*:
  order (*op* $\geq$) (*op* $>$)
by *unfold-locales*
  (*simp add*: *less-le*, *auto intro*: *antisym order-trans*)

end

5.2 Linear (total) orders

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

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

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

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

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

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

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

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

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

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

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

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

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

Replacing the old Nat.leI

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

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

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

Reverse order

lemma *linorder-reverse*:
  linorder $(op \geq)$ $(op >)$
by *unfold-locales*
  (*simp add*: *less-le*, *auto intro*: *antisym order-trans simp add*: *linear*)

min/max

for historic reasons, definitions are done in context ord

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

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

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

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

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

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

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

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

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

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

end

5.3 Reasoning tools setup

ML ⟪

```
signature ORDERS =
sig
  val print-structures: Proof.context -> unit
  val setup: theory -> theory
  val order-tac: thm list -> Proof.context -> int -> tactic
end;

structure Orders: ORDERS =
struct

(** Theory and context data **)

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

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

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

```
  end;


(** Method **)

fun struct-tac ((s, [eq, le, less]), thms) prems =
  let
    fun decomp thy (Trueprop $ t) =
      let
        fun excluded t =
          (* exclude numeric types: linear arithmetic subsumes transitivity *)
          let val T = type-of t
          in
            T = HOLogic.natT orelse T = HOLogic.intT orelse T = HOLogic.realT
          end;
        fun rel (bin-op $ t1 $ t2) =
              if excluded t1 then NONE
              else if Pattern.matches thy (eq, bin-op) then SOME (t1, =, t2)
              else if Pattern.matches thy (le, bin-op) then SOME (t1, <=, t2)
              else if Pattern.matches thy (less, bin-op) then SOME (t1, <, t2)
              else NONE
          | rel - = NONE;
        fun dec (Const (@{const-name Not}, -) $ t) = (case rel t
              of NONE => NONE
               | SOME (t1, rel, t2) => SOME (t1, ~ ^ rel, t2))
          | dec x = rel x;
      in dec t end;
  in
    case s of
      order => Order-Tac.partial-tac decomp thms prems
    | linorder => Order-Tac.linear-tac decomp thms prems
     | - => error (Unknown kind of order ‘ ^ s ^ ′ encountered in transitivity
reasoner.)
  end

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


(** Attribute **)

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

val attribute = Attrib.syntax
(Scan.lift ((Args.add −− Args.name >> (fn (-, s) => SOME s) ||
Args.del >> K NONE) −−| Args.colon (∗ FIXME ||
Scan.succeed true ∗)) −− Scan.lift Args.name −−
Scan.repeat Args.term
>> (fn ((SOME tag, n), ts) => add-struct-thm (n, ts) tag
| ((NONE, n), ts) => del-struct (n, ts)));

(∗∗ Diagnostic command ∗∗)

val print = Toplevel.unknown-context o
Toplevel.keep (Toplevel.node-case
(Context.cases (print-structures o ProofContext.init) print-structures)
(print-structures o Proof.context-of));

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

(∗∗ Setup ∗∗)

val setup =
Method.add-methods
[(order, Method.ctxt-args (Method.SIMPLE-METHOD′ o order-tac []), transitivity reasoner)] #>
Attrib.add-attributes [(order, attribute, theorems controlling transitivity reasoner)];

end;

⟫

setup *Orders.setup*

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

context *order*
begin

lemmas
[order add less-reflE: order op = :: ′a ⇒ ′a ⇒ bool op <= op <] =
less-irrefl [THEN notE]
lemmas
[order add le-refl: order op = :: ′a => ′a => bool op <= op <] =
order-refl

lemmas
[*order add less-imp-le*: *order op = :: 'a => 'a => bool op <= op <*] =
less-imp-le
lemmas
[*order add eqI*: *order op = :: 'a => 'a => bool op <= op <*] =
antisym
lemmas
[*order add eqD1*: *order op = :: 'a => 'a => bool op <= op <*] =
eq-refl
lemmas
[*order add eqD2*: *order op = :: 'a => 'a => bool op <= op <*] =
sym [*THEN eq-refl*]
lemmas
[*order add less-trans*: *order op = :: 'a => 'a => bool op <= op <*] =
less-trans
lemmas
[*order add less-le-trans*: *order op = :: 'a => 'a => bool op <= op <*] =
less-le-trans
lemmas
[*order add le-less-trans*: *order op = :: 'a => 'a => bool op <= op <*] =
le-less-trans
lemmas
[*order add le-trans*: *order op = :: 'a => 'a => bool op <= op <*] =
order-trans
lemmas
[*order add le-neq-trans*: *order op = :: 'a => 'a => bool op <= op <*] =
le-neq-trans
lemmas
[*order add neq-le-trans*: *order op = :: 'a => 'a => bool op <= op <*] =
neq-le-trans
lemmas
[*order add less-imp-neq*: *order op = :: 'a => 'a => bool op <= op <*] =
less-imp-neq
lemmas
[*order add eq-neq-eq-imp-neq*: *order op = :: 'a => 'a => bool op <= op <*] =
eq-neq-eq-imp-neq
lemmas
[*order add not-sym*: *order op = :: 'a => 'a => bool op <= op <*] =
not-sym

end

context *linorder*
begin

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

lemmas

[*order add less-reflE*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
less-irrefl [*THEN notE*]
lemmas
[*order add le-refl*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
order-refl
lemmas
[*order add less-imp-le*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
less-imp-le
lemmas
[*order add not-lessI*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
not-less [*THEN iffD2*]
lemmas
[*order add not-leI*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
not-le [*THEN iffD2*]
lemmas
[*order add not-lessD*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
not-less [*THEN iffD1*]
lemmas
[*order add not-leD*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
not-le [*THEN iffD1*]
lemmas
[*order add eqI*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
antisym
lemmas
[*order add eqD1*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
eq-refl
lemmas
[*order add eqD2*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
sym [*THEN eq-refl*]
lemmas
[*order add less-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
less-trans
lemmas
[*order add less-le-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
less-le-trans
lemmas
[*order add le-less-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
le-less-trans
lemmas
[*order add le-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
order-trans
lemmas
[*order add le-neq-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
le-neq-trans
lemmas
[*order add neq-le-trans*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =
neq-le-trans
lemmas
[*order add less-imp-neq*: *linorder op* = :: *'a* => *'a* => *bool op* <= *op* <] =

less-imp-neq
lemmas
[*order add eq-neq-eq-imp-neq*: *linorder op* = :: 'a => 'a => *bool op* <= *op* <] =
eq-neq-eq-imp-neq
lemmas
[*order add not-sym*: *linorder op* = :: 'a => 'a => *bool op* <= *op* <] =
not-sym

end

setup ⟪
```
let

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

fun prove-antisym-le sg ss ((le as Const(-,T)) $ r $ s) =
  let val prems = prems-of-ss ss;
      val less = Const (@{const-name less}, T);
      val t = HOLogic.mk-Trueprop(le $ s $ r);
  in case find-first (prp t) prems of
       NONE =>
         let val t = HOLogic.mk-Trueprop(HOLogic.Not $ (less $ r $ s))
         in case find-first (prp t) prems of
              NONE => NONE
            | SOME thm => SOME(mk-meta-eq(thm RS @{thm linorder-class.antisym-conv1}))
         end
     | SOME thm => SOME(mk-meta-eq(thm RS @{thm order-class.antisym-conv}))
  end
  handle THM - => NONE;

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

fun add-simprocs procs thy =
  (Simplifier.change-simpset-of thy (fn ss => ss
    addsimprocs (map (fn (name, raw-ts, proc) =>
```

Simplifier.simproc thy name raw-ts proc)) *procs*); *thy*);
fun add-solver name tac thy =
(*Simplifier.change-simpset-of thy* (*fn ss* => *ss addSolver*
(*mk-solver′ name* (*fn ss* => *tac* (*MetaSimplifier.prems-of-ss ss*) (*MetaSimplifier.the-context ss*)))); *thy*);

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

5.4 Dense orders

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

begin

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

end

5.5 Name duplicates

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

lemmas *order-antisym* = *antisym*
lemmas *order-less-not-sym* = *order-class.less-not-sym*
lemmas *order-less-asym* = *order-class.less-asym*

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

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

5.6 Bounded quantifiers

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

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

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

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

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

```
  -Ex-less-eq :: [idt, 'a, bool] => bool    ((3? -<=-./ -) [0, 0, 10] 10)

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

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

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

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

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

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

  fun tr' q = (q,
    fn [Const (-bound, -) $ Free (v, -), Const (c, -) $ (Const (d, -) $ t $ u) $ P]
=>
      (case AList.lookup (op =) trans (q, c, d) of
        NONE => raise Match
```

```
    | SOME (l, g) =>
        if matches-bound v t andalso not (contains-var v u) then mk v l u P
        else if matches-bound v u andalso not (contains-var v t) then mk v g t P
        else raise Match)
    | - => raise Match);
in [tr' All-binder, tr' Ex-binder] end
⟫
```

5.7 Transitivity reasoning

context *ord*
begin

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

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

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

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

end

lemma *order-less-subst2*: $(a::'a::order) < b$ ==> $f\ b < (c::'c::order)$ ==>
 (!!$x\ y.\ x < y$ ==> $f\ x < f\ y$) ==> $f\ a < c$
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-trans*) **show** *?thesis* **.**
qed

lemma *order-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-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 *order-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
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

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

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

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

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

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

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

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

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

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

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

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

5.8 Order on bool

instance *bool* :: *order*
le-bool-def: $P \le Q \equiv P \longrightarrow Q$
less-bool-def: $P < Q \equiv P \le Q \wedge P \ne Q$
by *intro-classes* (*auto simp add*: *le-bool-def less-bool-def*)
lemmas [*code func del*] = *le-bool-def less-bool-def*

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

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

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

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

lemma [*code func*]:
False $\le b \longleftrightarrow$ *True*

True $\le$ *b* $\longleftrightarrow$ *b*
False $<$ *b* $\longleftrightarrow$ *b*
True $<$ *b* $\longleftrightarrow$ *False*
unfolding *le-bool-def less-bool-def* **by** *simp-all*

5.9 Order on sets

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

lemmas *basic-trans-rules* [*trans*] =
order-trans-rules set-rev-mp set-mp

5.10 Order on functions

instance *fun* :: (*type*, *ord*) *ord*
le-fun-def: $f \le g \equiv \forall x.\ f\ x \le g\ x$
less-fun-def: $f < g \equiv f \le g \wedge f \neq g$ **..**

lemmas [*code func del*] = *le-fun-def less-fun-def*

instance *fun* :: (*type*, *order*) *order*
by *default*
(*auto simp add*: *le-fun-def less-fun-def expand-fun-eq*
intro: *order-trans order-antisym*)

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

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

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

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

lemma *predicate1I* [*Pure.intro*!, *intro*!]:
assumes *PQ*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $P \le Q$
apply (*rule le-funI*)
apply (*rule le-boolI*)
apply (*rule PQ*)
apply *assumption*
done

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

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

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

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

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

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

5.11 Monotonicity, least value operator and min/max

context *order*
begin

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

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

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

end

context *linorder*
begin

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

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

end

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

lemma *Least-mono*:
mono (f::$'a$::*order* => $'b$::*order*) ==> *EX* x:S. *ALL* y:S. x <= y
==> (*LEAST* y. y : f ‘ S) = f (*LEAST* x. x : S)
— Courtesy of Stephan Merz
apply *clarify*
apply (*erule-tac* P = %x. x : S **in** *LeastI2-order*, *fast*)
apply (*rule LeastI2-order*)
apply (*auto elim*: *monoD intro*!: *order-antisym*)
done

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

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

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

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

done

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

end

6 Lattices: Abstract lattices

theory *Lattices*
imports *Orderings*
begin

6.1 Lattices

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

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

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

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

6.1.1 Intro and elim rules

context *lower-semilattice*
begin

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

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

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

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

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

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

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

end

context *upper-semilattice*
begin

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

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

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

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

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

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

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

end

6.1.2 Equational laws

context *lower-semilattice*
begin

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

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

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

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

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

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

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

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

end

context *upper-semilattice*
begin

lemma *sup-commute*: $(x \sqcup y) = (y \sqcup x)$

by (*blast intro*: *antisym*)

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

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

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

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

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

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

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

end

context *lattice*
begin

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

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

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

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

Towards distributivity

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

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

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

lemma *distrib-imp1*:
assumes *D*: !!$x\ y\ z.\ x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
shows $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
proof−

have $x \sqcup (y \sqcap z) = (x \sqcup (x \sqcap z)) \sqcup (y \sqcap z)$ **by**(*simp add*:*sup-inf-absorb*)
also have $\ldots = x \sqcup (z \sqcap (x \sqcup y))$ **by**(*simp add*:*D inf-commute sup-assoc*)
also have $\ldots = ((x \sqcup y) \sqcap x) \sqcup ((x \sqcup y) \sqcap z)$
by(*simp add*:*inf-sup-absorb inf-commute*)
also have $\ldots = (x \sqcup y) \sqcap (x \sqcup z)$ **by**(*simp add*:*D*)
finally show *?thesis* **.**
qed

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

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

end

6.2 Distributive lattices

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

context *distrib-lattice*
begin

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

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

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

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

end

6.3 Uniqueness of inf and sup

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

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

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

lemma (**in** *linorder*) *distrib-lattice-min-max*:
  distrib-lattice (*op* $\leq$) (*op* $<$) *min max*
proof *unfold-locales*
  have *aux*: $\bigwedge x\ y :: 'a.\ x < y \Longrightarrow y \leq x \Longrightarrow x = y$
    by (*auto simp add*: *less-le antisym*)
  fix $x\ y\ z$
  show *max x* (*min y z*) = *min* (*max x y*) (*max x z*)
  unfolding *min-def max-def*
  by *auto*
qed (*auto simp add*: *min-def max-def not-le less-imp-le*)

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

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

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

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

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

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

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

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

6.5 Complete lattices

class *complete-lattice* = *lattice* +
fixes *Inf* :: $'a$ *set* $\Rightarrow$ $'a$ ($\bigsqcap$- [*900*] *900*)
and *Sup* :: $'a$ *set* $\Rightarrow$ $'a$ ($\bigsqcup$- [*900*] *900*)
assumes *Inf-lower*: $x \in A \Longrightarrow \bigsqcap A \sqsubseteq x$
and *Inf-greatest*: $(\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x) \Longrightarrow z \sqsubseteq \bigsqcap A$
assumes *Sup-upper*: $x \in A \Longrightarrow x \sqsubseteq \bigsqcup A$
and *Sup-least*: $(\bigwedge x.\ x \in A \Longrightarrow x \sqsubseteq z) \Longrightarrow \bigsqcup A \sqsubseteq z$
begin

lemma *Inf-Sup*: $\bigsqcap A = \bigsqcup \{b.\ \forall a \in A.\ b \le a\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Sup-Inf*: $\bigsqcup A = \bigsqcap \{b.\ \forall a \in A.\ a \le b\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Inf-Univ*: $\bigsqcap$ *UNIV* = $\bigsqcup \{\}$
unfolding *Sup-Inf* **by** *auto*

lemma *Sup-Univ*: $\bigsqcup$ *UNIV* = $\bigsqcap \{\}$
unfolding *Inf-Sup* **by** *auto*

lemma *Inf-insert*: $\bigsqcap$ *insert a A* = $a \sqcap \bigsqcap A$
apply (*rule antisym*)
apply (*rule le-infI*)
apply (*rule Inf-lower*)
apply *simp*
apply (*rule Inf-greatest*)
apply (*rule Inf-lower*)
apply *simp*
apply (*rule Inf-greatest*)
apply (*erule insertE*)

apply (*rule le-infI1*)
apply *simp*
apply (*rule le-infI2*)
apply (*erule Inf-lower*)
done

lemma *Sup-insert*: $\bigsqcup$ *insert a A* $= a \sqcup \bigsqcup A$
apply (*rule antisym*)
apply (*rule Sup-least*)
apply (*erule insertE*)
apply (*rule le-supI1*)
apply *simp*
apply (*rule le-supI2*)
apply (*erule Sup-upper*)
apply (*rule le-supI*)
apply (*rule Sup-upper*)
apply *simp*
apply (*rule Sup-least*)
apply (*rule Sup-upper*)
apply *simp*
done

lemma *Inf-singleton* [*simp*]:
$\bigsqcap\{a\} = a$
by (*auto intro*: *antisym Inf-lower Inf-greatest*)

lemma *Sup-singleton* [*simp*]:
$\bigsqcup\{a\} = a$
by (*auto intro*: *antisym Sup-upper Sup-least*)

lemma *Inf-insert-simp*:
$\bigsqcap$ *insert a A* $=$ (*if* $A = \{\}$ *then a else* $a \sqcap \bigsqcap A$)
by (*cases* $A = \{\}$) (*simp-all*, *simp add*: *Inf-insert*)

lemma *Sup-insert-simp*:
$\bigsqcup$ *insert a A* $=$ (*if* $A = \{\}$ *then a else* $a \sqcup \bigsqcup A$)
by (*cases* $A = \{\}$) (*simp-all*, *simp add*: *Sup-insert*)

lemma *Inf-binary*:
$\bigsqcap\{a, b\} = a \sqcap b$
by (*simp add*: *Inf-insert-simp*)

lemma *Sup-binary*:
$\bigsqcup\{a, b\} = a \sqcup b$
by (*simp add*: *Sup-insert-simp*)

definition
top :: *'a* **where**
top $= \bigsqcap\{\}$

definition
  bot :: *'a* **where**
  bot = $\bigsqcup${}

lemma *top-greatest* [*simp*]: *x* ≤ *top*
  by (*unfold top-def*, *rule Inf-greatest*, *simp*)

lemma *bot-least* [*simp*]: *bot* ≤ *x*
  by (*unfold bot-def*, *rule Sup-least*, *simp*)

definition
  SUPR :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a*
where
  SUPR A f == $\bigsqcup$ (*f* ‘ *A*)

definition
  INFI :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a*
where
  INFI A f == $\bigsqcap$ (*f* ‘ *A*)

end

syntax
```
  -SUP1   :: pttrns => 'b => 'b          ((3SUP -./ -) [0, 10] 10)
  -SUP    :: pttrn => 'a set => 'b => 'b  ((3SUP -:-./ -) [0, 10] 10)
  -INF1   :: pttrns => 'b => 'b          ((3INF -./ -) [0, 10] 10)
  -INF    :: pttrn => 'a set => 'b => 'b  ((3INF -:-./ -) [0, 10] 10)
```

translations
```
  SUP x y. B  == SUP x. SUP y. B
  SUP x. B    == CONST SUPR UNIV (%x. B)
  SUP x. B    == SUP x:UNIV. B
  SUP x:A. B  == CONST SUPR A (%x. B)
  INF x y. B  == INF x. INF y. B
  INF x. B    == CONST INFI UNIV (%x. B)
  INF x. B    == INF x:UNIV. B
  INF x:A. B  == CONST INFI A (%x. B)
```

print-translation ⟪
```
let
  fun btr' syn (A :: Abs abs :: ts) =
    let val (x,t) = atomic-abs-tr' abs
    in list-comb (Syntax.const syn $ x $ A $ t, ts) end
  val const-syntax-name = Sign.const-syntax-name @{theory} o fst o dest-Const
in
[(const-syntax-name @{term SUPR}, btr' -SUP),(const-syntax-name @{term INFI},
btr' -INF)]
```

end
$\rangle\rangle$

context *complete-lattice*
begin

lemma *le-SUPI*: $i : A \Longrightarrow M\ i \leq (SUP\ i{:}A.\ M\ i)$
by (*auto simp add*: *SUPR-def intro*: *Sup-upper*)

lemma *SUP-leI*: $(\bigwedge i.\ i : A \Longrightarrow M\ i \leq u) \Longrightarrow (SUP\ i{:}A.\ M\ i) \leq u$
by (*auto simp add*: *SUPR-def intro*: *Sup-least*)

lemma *INF-leI*: $i : A \Longrightarrow (INF\ i{:}A.\ M\ i) \leq M\ i$
by (*auto simp add*: *INFI-def intro*: *Inf-lower*)

lemma *le-INFI*: $(\bigwedge i.\ i : A \Longrightarrow u \leq M\ i) \Longrightarrow u \leq (INF\ i{:}A.\ M\ i)$
by (*auto simp add*: *INFI-def intro*: *Inf-greatest*)

lemma *SUP-const*[*simp*]: $A \neq \{\} \Longrightarrow (SUP\ i{:}A.\ M) = M$
by (*auto intro*: *antisym SUP-leI le-SUPI*)

lemma *INF-const*[*simp*]: $A \neq \{\} \Longrightarrow (INF\ i{:}A.\ M) = M$
by (*auto intro*: *antisym INF-leI le-INFI*)

end

6.6 Bool as lattice

instance *bool* :: *distrib-lattice*
inf-bool-eq: $P \sqcap Q \equiv P \wedge Q$
sup-bool-eq: $P \sqcup Q \equiv P \vee Q$
by *intro-classes* (*auto simp add*: *inf-bool-eq sup-bool-eq le-bool-def*)

instance *bool* :: *complete-lattice*
Inf-bool-def: $\bigsqcap A \equiv \forall x{\in}A.\ x$
Sup-bool-def: $\bigsqcup A \equiv \exists x{\in}A.\ x$
by *intro-classes* (*auto simp add*: *Inf-bool-def Sup-bool-def le-bool-def*)

lemma *Inf-empty-bool* [*simp*]:
$\bigsqcap \{\}$
unfolding *Inf-bool-def* **by** *auto*

lemma *not-Sup-empty-bool* [*simp*]:
$\neg\ Sup\ \{\}$
unfolding *Sup-bool-def* **by** *auto*

lemma *top-bool-eq*: $top = True$
by (*iprover intro*!: *order-antisym le-boolI top-greatest*)

lemma *bot-bool-eq*: $bot = False$
  by (*iprover intro*!: *order-antisym le-boolI bot-least*)

6.7 Set as lattice

instance *set* :: (*type*) *distrib-lattice*
  inf-set-eq: $A \sqcap B \equiv A \cap B$
  sup-set-eq: $A \sqcup B \equiv A \cup B$
  by *intro-classes* (*auto simp add*: *inf-set-eq sup-set-eq*)

lemmas [*code func del*] = *inf-set-eq sup-set-eq*

lemma *mono-Int*: $mono\ f \Longrightarrow f\ (A \cap B) \subseteq f\ A \cap f\ B$
  apply (*fold inf-set-eq sup-set-eq*)
  apply (*erule mono-inf*)
  done

lemma *mono-Un*: $mono\ f \Longrightarrow f\ A \cup f\ B \subseteq f\ (A \cup B)$
  apply (*fold inf-set-eq sup-set-eq*)
  apply (*erule mono-sup*)
  done

instance *set* :: (*type*) *complete-lattice*
  Inf-set-def: $\bigsqcap S \equiv \bigcap S$
  Sup-set-def: $\bigsqcup S \equiv \bigcup S$
  by *intro-classes* (*auto simp add*: *Inf-set-def Sup-set-def*)

lemmas [*code func del*] = *Inf-set-def Sup-set-def*

lemma *top-set-eq*: $top = UNIV$
  by (*iprover intro*!: *subset-antisym subset-UNIV top-greatest*)

lemma *bot-set-eq*: $bot = \{\}$
  by (*iprover intro*!: *subset-antisym empty-subsetI bot-least*)

6.8 Fun as lattice

instance *fun* :: (*type*, *lattice*) *lattice*
  inf-fun-eq: $f \sqcap g \equiv (\lambda x.\ f\ x \sqcap g\ x)$
  sup-fun-eq: $f \sqcup g \equiv (\lambda x.\ f\ x \sqcup g\ x)$
apply *intro-classes*
unfolding *inf-fun-eq sup-fun-eq*
apply (*auto intro*: *le-funI*)
apply (*rule le-funI*)
apply (*auto dest*: *le-funD*)
apply (*rule le-funI*)
apply (*auto dest*: *le-funD*)
done

lemmas [*code func del*] = *inf-fun-eq sup-fun-eq*

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

instance *fun* :: (*type*, *complete-lattice*) *complete-lattice*
  Inf-fun-def: $\bigsqcap A \equiv (\lambda x.\ \bigsqcap\{y.\ \exists f{\in}A.\ y = f\ x\})$
  Sup-fun-def: $\bigsqcup A \equiv (\lambda x.\ \bigsqcup\{y.\ \exists f{\in}A.\ y = f\ x\})$
  by *intro-classes*
    (*auto simp add*: *Inf-fun-def Sup-fun-def le-fun-def*
      intro: *Inf-lower Sup-upper Inf-greatest Sup-least*)

lemmas [*code func del*] = *Inf-fun-def Sup-fun-def*

lemma *Inf-empty-fun*:
  $\bigsqcap\{\} = (\lambda\text{-}.\ \bigsqcap\{\})$
  by *rule* (*auto simp add*: *Inf-fun-def*)

lemma *Sup-empty-fun*:
  $\bigsqcup\{\} = (\lambda\text{-}.\ \bigsqcup\{\})$
  by *rule* (*auto simp add*: *Sup-fun-def*)

lemma *top-fun-eq*: *top* = (λx. *top*)
  by (*iprover intro*!: *order-antisym le-funI top-greatest*)

lemma *bot-fun-eq*: *bot* = (λx. *bot*)
  by (*iprover intro*!: *order-antisym le-funI bot-least*)

redundant bindings

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

no-notation
  less-eq (**infix** $\sqsubseteq$ *50*) **and**
  less (**infix** $\sqsubset$ *50*) **and**
  inf (**infixl** $\sqcap$ *70*) **and**
  sup (**infixl** $\sqcup$ *65*) **and**
  Inf ($\bigsqcap$ - [*900*] *900*) **and**
  Sup ($\bigsqcup$ - [*900*] *900*)

end

7 Typedef: HOL type definitions

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

(*Tools/typedef-codegen.ML*)
begin

ML ⟪
structure HOL = struct val thy = theory HOL end;
⟫ — belongs to theory HOL

locale *type-definition* =
  fixes *Rep* **and** *Abs* **and** *A*
  assumes *Rep*: *Rep* $x \in A$
    and *Rep-inverse*: *Abs* (*Rep* x) = x
    and *Abs-inverse*: $y \in A$ ==> *Rep* (*Abs* y) = y
  — This will be axiomatized for each typedef!
begin

lemma *Rep-inject*:
  (*Rep* x = *Rep* y) = ($x = y$)
proof
  assume *Rep* x = *Rep* y
  then have *Abs* (*Rep* x) = *Abs* (*Rep* y) **by** (*simp only*:)
  moreover have *Abs* (*Rep* x) = x **by** (*rule Rep-inverse*)
  moreover have *Abs* (*Rep* y) = y **by** (*rule Rep-inverse*)
  ultimately show $x = y$ **by** *simp*
next
  assume $x = y$
  thus *Rep* x = *Rep* y **by** (*simp only*:)
qed

lemma *Abs-inject*:
  assumes x: $x \in A$ **and** y: $y \in A$
  shows (*Abs* x = *Abs* y) = ($x = y$)
proof
  assume *Abs* x = *Abs* y
  then have *Rep* (*Abs* x) = *Rep* (*Abs* y) **by** (*simp only*:)
  moreover from x **have** *Rep* (*Abs* x) = x **by** (*rule Abs-inverse*)
  moreover from y **have** *Rep* (*Abs* y) = y **by** (*rule Abs-inverse*)
  ultimately show $x = y$ **by** *simp*
next
  assume $x = y$
  thus *Abs* x = *Abs* y **by** (*simp only*:)
qed

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

```
qed

lemma Abs-cases [cases type]:
  assumes r: !!y. x = Abs y ==> 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 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)
  moreover from y have Rep (Abs y) = y by (rule Abs-inverse)
  ultimately show P y by simp
qed

lemma Abs-induct [induct type]:
  assumes r: !!y. y ∈ A ==> P (Abs y)
  shows P x
proof −
  have Rep x ∈ A by (rule Rep)
  then have P (Abs (Rep x)) by (rule r)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  ultimately show P x by simp
qed

lemma Rep-range:
  shows range Rep = A
proof
  show range Rep <= A using Rep by (auto simp add: image-def)
  show A <= range Rep
  proof
    fix x assume x : A
    hence x = Rep (Abs x) by (rule Abs-inverse [symmetric])
    thus x : range Rep by (rule range-eqI)
  qed
qed

end

use Tools/typedef-package.ML
use Tools/typecopy-package.ML
use Tools/typedef-codegen.ML
```

setup ⟪
  TypecopyPackage.setup
  #> TypedefCodegen.setup
⟫

end

8 Sum-Type: The Disjoint Sum of Two Types

theory *Sum-Type*
imports *Typedef Fun*
begin

The representations of the two injections

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

  Inr-Rep :: ['*b*, '*a*, '*b*, *bool*] => *bool*
  Inr-Rep == (%*b*. %*x y p*. *y*=*b* & ~*p*)

global

typedef (*Sum*)
  ('*a*, '*b*) +   (**infixr** + *10*)
    = {*f*. (? *a*. *f* = *Inl-Rep*(*a*::'*a*)) | (? *b*. *f* = *Inr-Rep*(*b*::'*b*))}
  by *auto*

local

abstract constants and syntax

constdefs
  Inl :: '*a* => '*a* + '*b*
   Inl == (%*a*. *Abs-Sum*(*Inl-Rep*(*a*)))

  Inr :: '*b* => '*a* + '*b*
   Inr == (%*b*. *Abs-Sum*(*Inr-Rep*(*b*)))

  Plus :: ['*a set*, '*b set*] => ('*a* + '*b*) *set*   (**infixr** <+> *65*)
   A <+> *B* == (*Inl*'*A*) *Un* (*Inr*'*B*)
   — disjoint sum for sets; the operator + is overloaded with wrong type!

  Part :: ['*a set*, '*b* => '*a*] => '*a set*
   Part A h == *A Int* {*x*. ? *z*. *x* = *h*(*z*)}
   — for selecting out the components of a mutually recursive definition

lemma *Inl-RepI*: *Inl-Rep*(*a*) : *Sum*
by (*auto simp add*: *Sum-def*)

lemma *Inr-RepI*: *Inr-Rep*(*b*) : *Sum*
by (*auto simp add*: *Sum-def*)

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

8.1 Freeness Properties for *Inl* and *Inr*

Distinctness

lemma *Inl-Rep-not-Inr-Rep*: *Inl-Rep*(*a*) ~= *Inr-Rep*(*b*)
by (*auto simp add*: *Inl-Rep-def Inr-Rep-def expand-fun-eq*)

lemma *Inl-not-Inr* [*iff*]: *Inl*(*a*) ~= *Inr*(*b*)
apply (*simp add*: *Inl-def Inr-def*)
apply (*rule inj-on-Abs-Sum* [*THEN inj-on-contraD*])
apply (*rule Inl-Rep-not-Inr-Rep*)
apply (*rule Inl-RepI*)
apply (*rule Inr-RepI*)
done

lemmas *Inr-not-Inl* = *Inl-not-Inr* [*THEN not-sym*, *standard*]
declare *Inr-not-Inl* [*iff*]

lemmas *Inl-neq-Inr* = *Inl-not-Inr* [*THEN notE*, *standard*]
lemmas *Inr-neq-Inl* = *sym* [*THEN Inl-neq-Inr*, *standard*]

Injectiveness

lemma *Inl-Rep-inject*: *Inl-Rep*(*a*) = *Inl-Rep*(*c*) ==> *a*=*c*
by (*auto simp add*: *Inl-Rep-def expand-fun-eq*)

lemma *Inr-Rep-inject*: *Inr-Rep*(*b*) = *Inr-Rep*(*d*) ==> *b*=*d*
by (*auto simp add*: *Inr-Rep-def expand-fun-eq*)

lemma *inj-Inl*: *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*)

8.2 Projections

definition
  sum-case f g x =
  (*if* (∃!*y*. *x* = *Inl y*)
  then f (*THE y*. *x* = *Inl y*)
  else g (*THE y*. *x* = *Inr y*))
definition *Projl x* = *sum-case id arbitrary x*
definition *Projr x* = *sum-case arbitrary id x*

lemma *sum-cases*[*simp*]:
  sum-case f g (*Inl x*) = *f x*
  sum-case f g (*Inr y*) = *g y*
  unfolding *sum-case-def*
  by *auto*

lemma *Projl-Inl*[*simp*]: *Projl* (*Inl x*) = *x*
  unfolding *Projl-def* **by** *simp*

lemma *Projr-Inr*[*simp*]: *Projr* (*Inr x*) = *x*
  unfolding *Projr-def* **by** *simp*

8.3 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

8.4 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*

8.5 Code generator setup

instance + :: (*eq*, *eq*) *eq* **..**

lemma [*code func*]:
 (*Inl x* :: ′*a*::*eq* + ′*b*::*eq*) = *Inl y* ⟷ *x* = *y*
 unfolding *Inl-eq* **..**

lemma [*code func*]:
 (*Inr x* :: ′*a*::*eq* + ′*b*::*eq*) = *Inr y* ⟷ *x* = *y*
 unfolding *Inr-eq* **..**

lemma [*code func*]:
 Inl (*x*::′*a*::*eq*) = *Inr* (*y*::′*b*::*eq*) ⟷ *False*
 using *Inl-not-Inr* **by** *auto*

lemma [*code func*]:
 Inr (*x*::′*b*::*eq*) = *Inl* (*y*::′*a*::*eq*) ⟷ *False*
 using *Inr-not-Inl* **by** *auto*

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

9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*
imports *Lattices Sum-Type*
uses
(*Tools/inductive-package.ML*)
Tools/dseq.ML
(*Tools/inductive-codegen.ML*)
(*Tools/datatype-aux.ML*)
(*Tools/datatype-prop.ML*)
(*Tools/datatype-rep-proofs.ML*)
(*Tools/datatype-abs-proofs.ML*)
(*Tools/datatype-case.ML*)
(*Tools/datatype-package.ML*)
(*Tools/primrec-package.ML*)
begin

9.1 Least and greatest fixed points

definition
$lfp :: ('a::complete\text{-}lattice \Rightarrow 'a) \Rightarrow 'a$ **where**
$lfp\ f = Inf\ \{u.\ f\ u \leq u\}$ — least fixed point

definition
$gfp :: ('a::complete\text{-}lattice \Rightarrow 'a) \Rightarrow 'a$ **where**
$gfp\ f = Sup\ \{u.\ u \leq f\ u\}$ — greatest fixed point

9.2 Proof of Knaster-Tarski Theorem using *lfp*

lfp f is the least upper bound of the set $\{u.\ f\ u \leq u\}$

lemma *lfp-lowerbound*: $f\ A \leq A$ ==> *lfp* $f \leq A$
 by (*auto simp add*: *lfp-def intro*: *Inf-lower*)

lemma *lfp-greatest*: (!!u. $f\ u \leq u$ ==> $A \leq u$) ==> $A \leq$ *lfp f*
 by (*auto simp add*: *lfp-def intro*: *Inf-greatest*)

lemma *lfp-lemma2*: *mono f* ==> f (*lfp f*) $\leq$ *lfp f*
 by (*iprover intro*: *lfp-greatest order-trans monoD lfp-lowerbound*)

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

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

lemma *lfp-const*: *lfp* ($\lambda x.\ t$) = t
 by (*rule lfp-unfold*) (*simp add*:*mono-def*)

9.3 General induction rules for least fixed points

theorem *lfp-induct*:
 assumes *mono*: *mono f* **and** *ind*: f (*inf* (*lfp f*) P) <= P
 shows *lfp f* <= P
proof −
 have *inf* (*lfp f*) P <= *lfp f* **by** (*rule inf-le1*)
 with *mono* **have** f (*inf* (*lfp f*) P) <= f (*lfp f*) **..**
 also from *mono* **have** f (*lfp f*) = *lfp f* **by** (*rule lfp-unfold* [*symmetric*])
 finally have f (*inf* (*lfp f*) P) <= *lfp f* **.**
 from *this* **and** *ind* **have** f (*inf* (*lfp f*) P) <= *inf* (*lfp f*) P **by** (*rule le-infI*)
 hence *lfp f* <= *inf* (*lfp f*) P **by** (*rule lfp-lowerbound*)
 also have *inf* (*lfp f*) P <= P **by** (*rule inf-le2*)
 finally show *?thesis* **.**
qed

lemma *lfp-induct-set*:
 assumes *lfp*: a: *lfp*(f)
  and *mono*: *mono*(f)
  and *indhyp*: !!x. [| x: f(*lfp*(f) *Int* {x. $P(x)$}) |] ==> $P(x)$
 shows $P(a)$
 by (*rule lfp-induct* [*THEN subsetD*, *THEN CollectD*, *OF mono - lfp*])
  (*auto simp*: *inf-set-eq intro*: *indhyp*)

lemma *lfp-ordinal-induct*:
 assumes *mono*: *mono f*
 and *P-f*: !!S. $P\ S$ ==> $P(f\ S)$
 and *P-Union*: !!M. !S:M. $P\ S$ ==> P(*Union M*)

```
  shows P(lfp f)
proof -
  let ?M = {S. S ⊆ lfp f & P S}
  have P (Union ?M) using P-Union by simp
  also have Union ?M = lfp f
  proof
    show Union ?M ⊆ lfp f by blast
    hence f (Union ?M) ⊆ f (lfp f) by (rule mono [THEN monoD])
    hence f (Union ?M) ⊆ lfp f using mono [THEN lfp-unfold] by simp
    hence f (Union ?M) ∈ ?M using P-f P-Union by simp
    hence f (Union ?M) ⊆ Union ?M by (rule Union-upper)
    thus lfp f ⊆ Union ?M by (rule lfp-lowerbound)
  qed
  finally show ?thesis .
qed
```

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

```
lemma def-lfp-unfold: [| h==lfp(f);  mono(f) |] ==> h = f(h)
by (auto intro!: lfp-unfold)

lemma def-lfp-induct:
    [| A == lfp(f); mono(f);
       f (inf A P) ≤ P
    |] ==> A ≤ P
  by (blast intro: lfp-induct)

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


lemma lfp-mono: (!!Z. f Z ≤ g Z) ==> lfp f ≤ lfp g
  by (rule lfp-lowerbound [THEN lfp-greatest], blast intro: order-trans)
```

9.4 Proof of Knaster-Tarski Theorem using *gfp*

gfp f is the greatest lower bound of the set $\{u.\ u \leq f\ u\}$

```
lemma gfp-upperbound: X ≤ f X ==> X ≤ gfp f
  by (auto simp add: gfp-def intro: Sup-upper)

lemma gfp-least: (!!u. u ≤ f u ==> u ≤ X) ==> gfp f ≤ X
  by (auto simp add: gfp-def intro: Sup-least)

lemma gfp-lemma2: mono f ==> gfp f ≤ f (gfp f)
  by (iprover intro: gfp-least order-trans monoD gfp-upperbound)

lemma gfp-lemma3: mono f ==> f (gfp f) ≤ gfp f
```

by (*iprover intro*: *gfp-lemma2 monoD gfp-upperbound*)

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

9.5 Coinduction rules for greatest fixed points

weak version

lemma *weak-coinduct*: [| *a*: *X*; $X \subseteq f(X)$ |] ==> *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* $\leq$ *f* (*sup X* (*gfp f*)); *mono f* |] ==> *sup X* (*gfp f*) $\leq$ *f* (*sup X* (*gfp f*))
apply (*frule gfp-lemma2*)
apply (*drule mono-sup*)
apply (*rule le-supI*)
apply *assumption*
apply (*rule order-trans*)
apply (*rule order-trans*)
apply *assumption*
apply (*rule sup-ge2*)
apply *assumption*
done

strong version, thanks to Coen and Frost

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

lemma *coinduct*: [| *mono*(*f*); *X* $\leq$ *f* (*sup X* (*gfp f*)) |] ==> *X* $\leq$ *gfp*(*f*)
apply (*rule order-trans*)
apply (*rule sup-ge1*)
apply (*erule gfp-upperbound* [*OF coinduct-lemma*])
apply *assumption*
done

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

9.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition $X \subseteq f\ X$ 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*:
  [| $mono(f)$; $a{:}X$; $X \subseteq f(lfp(\%x.\ f(x)\ Un\ X\ Un\ gfp(f)))$ |] ==> $a : gfp(f)$
apply (*rule coinduct3-lemma* [*THEN* [*2*] *weak-coinduct*])
apply (*rule coinduct3-mono-lemma* [*THEN lfp-unfold*, *THEN ssubst*], *auto*)
done

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding

lemma *def-gfp-unfold*: [| $A{=}{=}gfp(f)$; $mono(f)$ |] ==> $A = f(A)$
by (*auto intro*!: *gfp-unfold*)

lemma *def-coinduct*:
  [| $A{=}{=}gfp(f)$; $mono(f)$; $X \leq f(sup\ X\ A)$ |] ==> $X \leq A$
by (*iprover intro*!: *coinduct*)

lemma *def-coinduct-set*:
  [| $A{=}{=}gfp(f)$; $mono(f)$; $a{:}X$; $X \subseteq f(X\ Un\ A)$ |] ==> $a{:}\ A$
by (*auto intro*!: *coinduct-set*)

lemma *def-Collect-coinduct*:
  [| $A == gfp(\%w.\ Collect(P(w)))$; $mono(\%w.\ Collect(P(w)))$;
    $a{:}\ X$; !!z. $z{:}\ X$ ==> $P\ (X\ Un\ A)\ z$ |] ==>
  $a : A$
apply (*erule def-coinduct-set*, *auto*)
done

lemma *def-coinduct3*:
  [| $A{=}{=}gfp(f)$; $mono(f)$; $a{:}X$; $X \subseteq f(lfp(\%x.\ f(x)\ Un\ X\ Un\ A))$ |] ==> $a{:}\ A$
by (*auto intro*!: *coinduct3*)

Monotonicity of *gfp*!

lemma *gfp-mono*: (!!Z. $f\ Z \leq g\ Z$) ==> $gfp\ f \leq gfp\ g$
  by (*rule gfp-upperbound* [*THEN gfp-least*], *blast intro*: *order-trans*)

9.7 Inductive predicates and sets

Inversion of injective functions.

constdefs
  myinv :: ($'a$ => $'b$) => ($'b$ => $'a$)
  myinv (f :: $'a$ => $'b$) == λy. *THE* x. $f\ x = y$

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

lemma *f-myinv-f*: *inj* f ==> $y \in$ *range* f ==> 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.

theorems *basic-monos* =
  subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
  Collect-mono in-mono vimage-mono
  imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
  not-all not-ex
  Ball-def Bex-def
  induct-rulify-fallback

ML ⟪
val def-lfp-unfold = @{*thm def-lfp-unfold*}
val def-gfp-unfold = @{*thm def-gfp-unfold*}
val def-lfp-induct = @{*thm def-lfp-induct*}
val def-coinduct = @{*thm def-coinduct*}
val inf-bool-eq = @{*thm inf-bool-eq*}
val inf-fun-eq = @{*thm inf-fun-eq*}
val le-boolI = @{*thm le-boolI*}

val le-boolI′ = @{thm le-boolI′}
val le-funI = @{thm le-funI}
val le-boolE = @{thm le-boolE}
val le-funE = @{thm le-funE}
val le-boolD = @{thm le-boolD}
val le-funD = @{thm le-funD}
val le-bool-def = @{thm le-bool-def}
val le-fun-def = @{thm le-fun-def}
⟫

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

theorems [*mono*] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
not-all not-ex
Ball-def Bex-def
induct-rulify-fallback

9.8 Inductive datatypes and primitive recursion

Package setup.

use *Tools/datatype-aux.ML*
use *Tools/datatype-prop.ML*
use *Tools/datatype-rep-proofs.ML*
use *Tools/datatype-abs-proofs.ML*
use *Tools/datatype-case.ML*
use *Tools/datatype-package.ML*
setup *DatatypePackage.setup*
use *Tools/primrec-package.ML*

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

Lambda-abstractions with pattern matching:

syntax
-lam-pats-syntax :: *cases-syn* => ′*a* => ′*b* ((%-) *10*)
syntax (*xsymbols*)
-lam-pats-syntax :: *cases-syn* => ′*a* => ′*b* ((λ-) *10*)

parse-translation (**advanced**) ⟪
let
fun fun-tr ctxt [*cs*] =
let
val x = *Free* (*Name.variant* (*add-term-free-names* (*cs*, [])) *x*, *dummyT*);
val ft = *DatatypeCase.case-tr true DatatypePackage.datatype-of-constr*
ctxt [*x*, *cs*]
in lambda x ft end

```
in [(-lam-pats-syntax, fun-tr)] end
⟫
```

end

10 Product-Type: Cartesian products

theory *Product-Type*
imports *Inductive*
uses
  (*Tools/split-rule.ML*)
  (*Tools/inductive-set-package.ML*)
  (*Tools/inductive-realizer.ML*)
  (*Tools/datatype-realizer.ML*)
begin

10.1 *bool* is a datatype

rep-datatype *bool*
  distinct *True-not-False False-not-True*
  induction *bool-induct*

declare *case-split* [*cases type*: *bool*]
  — prefer plain propositional version

10.2 Unit

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

definition
  Unity :: *unit* ('('))
where
  () = *Abs-unit True*

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

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

ML-setup ⟪
```
  val unit-eq-proc =
    let val unit-meta-eq = mk-meta-eq @{thm unit-eq} in
      Simplifier.simproc @{theory} unit-eq [x::unit]
        (fn - => fn - => fn t => if HOLogic.is-unit t then NONE else SOME
unit-meta-eq)
```

```
    end;

  Addsimprocs [unit-eq-proc];
»
```

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

rep-datatype *unit*
  induction *unit-induct*

lemma *unit-all-eq1*: (!!*x*::*unit*. *PROP P x*) == *PROP P* ()
  by *simp*

lemma *unit-all-eq2*: (!!*x*::*unit*. *PROP P*) == *PROP P*
  by (*rule triv-forall-equality*)

This rewrite counters the effect of *unit-eq-proc* on %*u*::*unit*. *f u*, replacing it by *f* rather than by %*u*. *f* ().

lemma *unit-abs-eta-conv* [*simp*,*noatp*]: (%*u*::*unit*. *f* ()) = *f*
  by (*rule ext*) *simp*

10.3 Pairs

10.3.1 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

10.3.2 Definitions

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

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

abbreviation
  Times :: ['a set, 'b set] => ('a * 'b) set
   (**infixr** <*> *80*) **where**
  A <*> B == Sigma A (%-. B)

notation (*xsymbols*)
  Times (**infixr** × *80*)

notation (*HTML* **output**)
  Times (**infixr** × *80*)

10.3.3 Concrete syntax

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 -:-./ -) [0, 0, 10] 10)

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)
```

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
⟫
```

10.3.4 Lemmas and proof tool setup

lemma *ProdI*: *Pair-Rep a b* : *Prod*
  unfolding *Prod-def* **by** *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*:
  assumes (*a*, *b*) = (*a'*, *b'*)
    and *a* = *a'* ==> *b* = *b'* ==> *R*
  shows *R*
  apply (*insert prems* [*unfolded Pair-def*])
  apply (*rule inj-on-Abs-Prod* [*THEN inj-onD*, *THEN Pair-Rep-inject*, *THEN conjE*])
  apply (*assumption* | *rule ProdI*)+
  done

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

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

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

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*
  using *PairE-lemma* [*of p*] **by** *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*
  then show *PROP P* (*a*, *b*) **.**
next
  fix *x*
  assume !!*a b*. *PROP P* (*a*, *b*)
  from ⟨*PROP P* (*fst x*, *snd x*)⟩ **show** *PROP P x* **by** *simp*
qed

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;

change-claset (fn cs => cs addSbefore (split-all-tac, split-all-tac));
⟫
```

lemma *split-paired-All* [*simp*]: (*ALL x. P x*) = (*ALL a b. P* (*a*, *b*))
  — [*iff*] is not a good idea because it makes *blast* loop
  by *fast*

lemma *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*, *code func*]: *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*

rep-datatype *prod*
inject *Pair-eq*
induction *prod-induct*

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

lemma *split-conv* [*simp*, *code func*]: *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-ss = HOL-basic-ss addsimps [thm cond-split-eta]
  fun  Pair-pat k 0 (Bound m) = (m = k)
  |    Pair-pat k i (Const (Pair,  -) $ Bound m $ t) = i > 0 andalso
                     m = k+i andalso Pair-pat k (i−1) t
  |    Pair-pat - - - = false;
  fun no-args k i (Abs (-, -, t)) = no-args (k+1) i t
  |   no-args k i (t $ u) = no-args k i t andalso no-args k i u
  |   no-args k i (Bound m) = m < k orelse m > k+i
  |   no-args - - - = true;
  fun split-pat tp i (Abs  (-,-,t)) = if tp 0 i t then SOME (i,t) else NONE
  |   split-pat tp i (Const (split, -) $ Abs (-, -, t)) = split-pat tp (i+1) t
  |   split-pat tp i - = NONE;
  fun metaeq ss lhs rhs = mk-meta-eq (Goal.prove (Simplifier.the-context ss) [] []
        (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs,rhs)))
        (K (simp-tac (Simplifier.inherit-context ss cond-split-eta-ss) 1)));

  fun beta-term-pat k i (Abs (-, -, t)) = beta-term-pat (k+1) i t
  |   beta-term-pat k i (t $ u) = Pair-pat k i (t $ u) orelse
                     (beta-term-pat k i t andalso beta-term-pat k i u)
  |   beta-term-pat k i t = no-args k i t;
  fun  eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
  |    eta-term-pat - - - = false;
  fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
  |   subst arg k i (t $ u) = if Pair-pat k i (t $ u) then incr-boundvars k arg
                          else (subst arg k i t $ subst arg k i u)
  |   subst arg k i t = t;
  fun beta-proc ss (s as Const (split, -) $ Abs (-, -, t) $ arg) =
        (case split-pat beta-term-pat 1 t of
        SOME (i,f) => SOME (metaeq ss s (subst arg 0 i f))
        | NONE => NONE)
  |   beta-proc - - = NONE;
  fun eta-proc ss (s as Const (split, -) $ Abs (-, -, t)) =
        (case split-pat eta-term-pat 1 t of
          SOME (-,ft) => SOME (metaeq ss s (let val (f $ arg) = ft in f end))
        | NONE => NONE)
  |   eta-proc - - = NONE;
in
 val split-beta-proc = Simplifier.simproc @{theory} split-beta [split f z] (K beta-proc);
 val split-eta-proc = Simplifier.simproc @{theory} split-eta [split f] (K 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* [*noatp*]: *R*(*split c p*) = (*ALL x y*. *p* = (*x*, *y*) --> *R*(*c x y*))
  — For use with *split* and the Simplifier.
  by (*insert surj-pair* [*of p*], *clarify*, *simp*)

split-split could be declared as [*split*] done after the Splitter has been speeded up significantly; precompute the constants involved and don't do anything unless the current goal contains one of those constants.

lemma *split-split-asm* [*noatp*]: *R* (*split c p*) = (~(*EX x y*. *p* = (*x*, *y*) & (~*R* (*c x y*))))
by (*subst split-split*, *simp*)

split used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *split* as rewrite.

lemma *splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *split c p*
  apply (*simp only*: *split-tupled-all*)
  apply (*simp* (*no-asm-simp*))
  done

lemma *splitI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *split c p x*
  apply (*simp only*: *split-tupled-all*)
  apply (*simp* (*no-asm-simp*))
  done

lemma *splitE*: *split c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
  by (*induct p*) (*auto simp add*: *split-def*)

lemma *splitE'*: *split c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
  by (*induct p*) (*auto simp add*: *split-def*)

lemma *splitE2*:
  [| *Q* (*split P z*); !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R* |] ==> *R*
proof –
  assume *q*: *Q* (*split P z*)
  assume *r*: !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R*
  show *R*
    apply (*rule r surjective-pairing*)+
    apply (*rule split-beta* [*THEN subst*], *rule q*)
    done
qed

lemma *splitD'*: *split R* (*a*,*b*) *c* ==> *R a b c*
  by *simp*

lemma *mem-splitI*: *z*: *c a b* ==> *z*: *split c* (*a*, *b*)
  by *simp*

lemma *mem-splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *z*: *c a b* |] ==> *z*: *split c p*
by (*simp only*: *split-tupled-all*, *simp*)

lemma *mem-splitE*:
  assumes *major*: *z*: *split c p*
    and *cases*: !!*x y*. [| *p* = (*x*,*y*); *z*: *c x y* |] ==> *Q*
  shows *Q*
  by (*rule major* [*unfolded split-def*] *cases surjective-pairing*)+

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

ML-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*) ∗)
change-claset (*fn cs* => *cs addSbefore* (*split-conv-tac*, *split-conv-tac*));
⟫

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

lemma *split-eta-SetCompr2* [*simp*,*noatp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P x y*) = *split P*
  by (*rule ext*) *fast*

lemma *split-part* [*simp*]: (%(*a*,*b*). *P* & *Q a b*) = (%*ab*. *P* & *split Q ab*)
  — Allows simplifications of nested splits in case of independent predicates.
  by (*rule ext*) *blast*

lemma *split-comp-eq*:
  fixes *f* :: *'a* => *'b* => *'c* **and** *g* :: *'d* => *'a*
  shows (%*u*. *f* (*g* (*fst u*)) (*snd u*)) = (*split* (%*x*. *f* (*g x*)))
  by (*rule ext*) *auto*

lemma *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*, *code func*]: *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*!]:
  assumes *major*: *c*: (*prod-fun f g*)‘*r*
    and *cases*: !!x y. [| *c*=(*f*(x),*g*(y)); (x,y):r |] ==> *P*
  shows *P*
  apply (*rule major* [*THEN imageE*])
  apply (*rule-tac* $p = x$ **in** *PairE*)
  apply (*rule cases*)
   apply (*blast intro*: *prod-fun*)
  apply *blast*
  done

definition
  upd-fst :: ($'a \Rightarrow 'c$) $\Rightarrow$ $'a \times 'b \Rightarrow 'c \times 'b$
where
  [*code func del*]: *upd-fst f* = *prod-fun f id*

definition
  upd-snd :: ($'b \Rightarrow 'c$) $\Rightarrow$ $'a \times 'b \Rightarrow 'a \times 'c$

where
  [*code func del*]: *upd-snd f* = *prod-fun id f*

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

lemma *upd-snd-conv* [*simp*, *code*]:
  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*,*noatp*]:
(*ALL z*: *Sigma A B*. *P z*) = (*ALL x*:*A*. *ALL y*: *B x*. *P*(*x*,*y*))
by *blast*

lemma *split-paired-Bex-Sigma* [*simp*,*noatp*]:
(*EX z*: *Sigma A B*. *P z*) = (*EX x*:*A*. *EX y*: *B x*. *P*(*x*,*y*))
by *blast*

lemma *Sigma-Un-distrib1*: (*SIGMA i*:*I Un J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) *Un* (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Un-distrib2*: (*SIGMA i*:*I*. *A*(*i*) *Un B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) *Un* (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Int-distrib1*: (*SIGMA i*:*I Int J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) *Int* (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Int-distrib2*: (*SIGMA i*:*I*. *A*(*i*) *Int B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) *Int* (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Diff-distrib1*: (*SIGMA i*:*I* − *J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) − (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Diff-distrib2*: (*SIGMA i*:*I*. *A*(*i*) − *B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) − (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Union*: *Sigma* (*Union X*) *B* = (*UN A*:*X*. *Sigma A B*)
by *blast*

Non-dependent versions are needed to avoid the need for higher-order 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*

lemmas [*code func del*] = *internal-split-def*

lemma *internal-split-conv*: *internal-split c* (*a*, *b*) = *c a b*
by (*simp only*: *internal-split-def split-conv*)

hide *const internal-split*

use *Tools/split-rule.ML*
setup *SplitRule.setup*

lemmas *prod-caseI* = *prod.cases* [*THEN iffD2*, *standard*]

lemma *prod-caseI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *prod-case c p*
  by *auto*

lemma *prod-caseI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *prod-case c p x*
  by (*auto simp*: *split-tupled-all*)

lemma *prod-caseE*: *prod-case c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
  by (*induct p*) *auto*

lemma *prod-caseE'*: *prod-case c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
  by (*induct p*) *auto*

lemma *prod-case-unfold*: *prod-case* = (%*c p*. *c* (*fst p*) (*snd p*))
  by (*simp add*: *expand-fun-eq*)

declare *prod-caseI2'* [*intro*!] *prod-caseI2* [*intro*!] *prod-caseI* [*intro*!]
declare *prod-caseE'* [*elim*!] *prod-caseE* [*elim*!]

lemma *prod-case-split*:
  prod-case = *split*
  by (*auto simp add*: *expand-fun-eq*)

10.4 Further cases/induct rules for tuples

lemma *prod-cases3* [*cases type*]:
  obtains (*fields*) *a b c* **where** *y* = (*a*, *b*, *c*)
  by (*cases y*, *case-tac b*) *blast*

lemma *prod-induct3* [*case-names fields*, *induct type*]:
    (!!*a b c*. *P* (*a*, *b*, *c*)) ==> *P x*
  by (*cases x*) *blast*

lemma *prod-cases4* [*cases type*]:
  obtains (*fields*) *a b c d* **where** *y* = (*a*, *b*, *c*, *d*)
  by (*cases y*, *case-tac c*) *blast*

lemma *prod-induct4* [*case-names fields*, *induct type*]:
    (!!*a b c d*. *P* (*a*, *b*, *c*, *d*)) ==> *P x*
  by (*cases x*) *blast*

lemma *prod-cases5* [*cases type*]:
obtains (*fields*) $a\ b\ c\ d\ e$ **where** $y = (a,\ b,\ c,\ d,\ e)$
by (*cases* y, *case-tac* d) *blast*

lemma *prod-induct5* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e.\ P\ (a,\ b,\ c,\ d,\ e)) ==> P\ x$
by (*cases* x) *blast*

lemma *prod-cases6* [*cases type*]:
obtains (*fields*) $a\ b\ c\ d\ e\ f$ **where** $y = (a,\ b,\ c,\ d,\ e,\ f)$
by (*cases* y, *case-tac* e) *blast*

lemma *prod-induct6* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f.\ P\ (a,\ b,\ c,\ d,\ e,\ f)) ==> P\ x$
by (*cases* x) *blast*

lemma *prod-cases7* [*cases type*]:
obtains (*fields*) $a\ b\ c\ d\ e\ f\ g$ **where** $y = (a,\ b,\ c,\ d,\ e,\ f,\ g)$
by (*cases* y, *case-tac* f) *blast*

lemma *prod-induct7* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f\ g.\ P\ (a,\ b,\ c,\ d,\ e,\ f,\ g)) ==> P\ x$
by (*cases* x) *blast*

10.5 Further lemmas

lemma
split-Pair: *split Pair* $x = x$
unfolding *split-def* **by** *auto*

lemma
split-comp: *split* $(f \circ g)\ x = f\ (g\ (\mathit{fst}\ x))\ (\mathit{snd}\ x)$
by (*cases* x, *simp*)

10.6 Code generator setup

instance *unit* :: *eq* **..**

lemma [*code func*]:
$(u{::}\mathit{unit}) = v \longleftrightarrow \mathit{True}$ **unfolding** *unit-eq* [*of* u] *unit-eq* [*of* v] **by** *rule+*

code-type *unit*
(*SML unit*)
(*OCaml unit*)
(*Haskell* ())

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

code-const *op* = :: *unit* ⇒ *unit* ⇒ *bool*

(*Haskell* **infixl** *4* ==)

code-const *Unity*
(*SML* ())
(*OCaml* ())
(*Haskell* ())

code-reserved *SML*
unit

code-reserved *OCaml*
unit

instance ∗ :: (*eq*, *eq*) *eq* **..**

lemma [*code func*]:
$(x1::'a::eq,\ y1::'b::eq) = (x2,\ y2) \longleftrightarrow x1 = x2 \wedge y1 = y2$ **by** *auto*

lemma *split-case-cert*:
assumes *CASE* ≡ *split f*
shows *CASE* (*a*, *b*) ≡ *f a b*
using *assms* **by** *simp*

setup ⟪
Code.add-case @{*thm split-case-cert*}
⟫

code-type ∗
(*SML* **infix** *2* ∗)
(*OCaml* **infix** *2* ∗)
(*Haskell* !((-),/ (-)))

code-instance ∗ :: *eq*
(*Haskell* −)

code-const *op* = :: $'a::eq \times 'b::eq \Rightarrow 'a \times 'b \Rightarrow bool$
(*Haskell* **infixl** *4* ==)

code-const *Pair*
(*SML* !((-),/ (-)))
(*OCaml* !((-),/ (-)))
(*Haskell* !((-),/ (-)))

code-const *fst* **and** *snd*
(*Haskell fst* **and** *snd*)

types-code
∗ ((- ∗/ -))
attach (*term-of*) ⟪

```
fun term-of-id-42 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    ((-,/ -))

setup ⟪

let

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

  Codegen.add-codegen let-codegen let-codegen
  #> Codegen.add-codegen split-codegen split-codegen

end
⟫
```

10.7 Legacy bindings

ML ⟪
val Collect-split = thm Collect-split;
val Compl-Times-UNIV1 = thm Compl-Times-UNIV1;
val Compl-Times-UNIV2 = thm Compl-Times-UNIV2;
val PairE = thm PairE;
val 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;
⟫

10.8 Further inductive packages

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

use *Tools/inductive-set-package.ML*
setup *InductiveSetPackage.setup*

use *Tools/datatype-realizer.ML*
setup *DatatypeRealizer.setup*

end

11 Relation: Relations

theory *Relation*
imports *Product-Type*
begin

11.1 Definitions

definition
converse :: $('a * 'b)$ *set* => $('b * 'a)$ *set*
((-^−*1*) [*1000*] *999*) **where**
r^−*1* == {(*y*, *x*). (*x*, *y*) : *r*}

notation (*xsymbols*)
converse ((-$^{-1}$) [*1000*] *999*)

definition
rel-comp :: [$('b * 'c)$ *set*, $('a * 'b)$ *set*] => $('a * 'c)$ *set*
(**infixr** *O 75*) **where**
r O s == {(*x*,*z*). *EX y*. (*x*, *y*) : *s* & (*y*, *z*) : *r*}

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

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

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

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

definition
Range :: $('a * 'b)$ *set* => *'b set* **where**
Range r == *Domain*(*r*^−*1*)

definition
Field :: $('a * 'a)$ *set* => *'a set* **where**
Field r == *Domain r* ∪ *Range r*

definition
refl :: ['a *set*, ('a * 'a) *set*] => *bool* **where** — reflexivity over a set
refl A r == *r* ⊆ *A* × *A* & (*ALL x*: *A*. (*x*,*x*) : *r*)

definition
sym :: ('a * 'a) *set* => *bool* **where** — symmetry predicate
sym r == *ALL x y*. (*x*,*y*): *r* --> (*y*,*x*): *r*

definition
antisym :: ('a * 'a) *set* => *bool* **where** — antisymmetry predicate
antisym r == *ALL x y*. (*x*,*y*):*r* --> (*y*,*x*):*r* --> *x*=*y*

definition
trans :: ('a * 'a) *set* => *bool* **where** — transitivity predicate
trans r == (*ALL x y z*. (*x*,*y*):*r* --> (*y*,*z*):*r* --> (*x*,*z*):*r*)

definition
single-valued :: ('a * 'b) *set* => *bool* **where**
single-valued r == *ALL x y*. (*x*,*y*):*r* --> (*ALL z*. (*x*,*z*):*r* --> *y*=*z*)

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

abbreviation
reflexive :: ('a * 'a) *set* => *bool* **where** — reflexivity over a type
reflexive == *refl UNIV*

11.2 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 *sym-Id*: *sym Id*
by (*simp add*: *sym-def*)

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

11.3 Diagonal: identity over a set

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

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

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

lemma *diagE* [*elim*!]:
  $c : diag\ A$ ==> (!!x. $x : A$ ==> $c = (x, x)$ ==> P) ==> 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*

11.4 Composition of two relations

lemma *rel-compI* [*intro*]:
  $(a, b) : s$ ==> $(b, c) : r$ ==> $(a, c) : r\ O\ s$
  by (*unfold rel-comp-def*) *blast*

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s$ ==>
  (!!$x\ y\ z$. $xz = (x, z)$ ==> $(x, y) : s$ ==> $(y, z) : r$ ==> P) ==> P
  by (*unfold rel-comp-def*) (*iprover elim*!: *CollectE splitE exE conjE*)

lemma *rel-compEpair*:
  $(a, c) : r\ O\ s$ ==> (!!y. $(a, y) : s$ ==> $(y, c) : r$ ==> P) ==> P
  by (*iprover elim*: *rel-compE Pair-inject ssubst*)

lemma *R-O-Id* [*simp*]: $R\ O\ Id = R$
  by *fast*

lemma *Id-O-R* [*simp*]: $Id\ O\ R = R$
  by *fast*

lemma *rel-comp-empty1*[*simp*]: {} $O\ R$ = {}
  by *blast*

lemma *rel-comp-empty2*[*simp*]: $R\ O$ {} = {}
  by *blast*

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
by *blast*

lemma *trans-O-subset*: *trans* r ==> $r\ O\ r \subseteq r$
by (*unfold trans-def*) *blast*

lemma *rel-comp-mono*: $r' \subseteq r$ ==> $s' \subseteq s$ ==> $(r'\ O\ s') \subseteq (r\ O\ s)$
by *blast*

lemma *rel-comp-subset-Sigma*:
$s \subseteq A \times B$ ==> $r \subseteq B \times C$ ==> $(r\ O\ s) \subseteq A \times C$
by *blast*

11.5 Reflexivity

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

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

lemma *reflD1*: *refl* $A\ r$ ==> $(x, y) : r$ ==> $x : A$
by (*unfold refl-def*) *blast*

lemma *reflD2*: *refl* $A\ r$ ==> $(x, y) : r$ ==> $y : A$
by (*unfold refl-def*) *blast*

lemma *refl-Int*: *refl* $A\ r$ ==> *refl* $B\ s$ ==> *refl* $(A \cap B)\ (r \cap s)$
by (*unfold refl-def*) *blast*

lemma *refl-Un*: *refl* $A\ r$ ==> *refl* $B\ s$ ==> *refl* $(A \cup B)\ (r \cup s)$
by (*unfold refl-def*) *blast*

lemma *refl-INTER*:
ALL x:S. *refl* $(A\ x)\ (r\ x)$ ==> *refl* (*INTER* S A) (*INTER* S r)
by (*unfold refl-def*) *fast*

lemma *refl-UNION*:
ALL x:S. *refl* $(A\ x)\ (r\ x) \Longrightarrow$ *refl* (*UNION* S A) (*UNION* S r)
by (*unfold refl-def*) *blast*

lemma *refl-diag*: *refl* A (*diag* A)
by (*rule reflI* [*OF diag-subset-Times diagI*])

11.6 Antisymmetry

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

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

lemma *antisym-subset*: *r* ⊆ *s* ==> *antisym s* ==> *antisym r*
by (*unfold antisym-def*) *blast*

lemma *antisym-empty* [*simp*]: *antisym* {}
by (*unfold antisym-def*) *blast*

lemma *antisym-diag* [*simp*]: *antisym* (*diag A*)
by (*unfold antisym-def*) *blast*

11.7 Symmetry

lemma *symI*: (!!*a b*. (*a*, *b*) : *r* ==> (*b*, *a*) : *r*) ==> *sym r*
by (*unfold sym-def*) *iprover*

lemma *symD*: *sym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r*
by (*unfold sym-def*, *blast*)

lemma *sym-Int*: *sym r* ==> *sym s* ==> *sym* (*r* ∩ *s*)
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-Un*: *sym r* ==> *sym s* ==> *sym* (*r* ∪ *s*)
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-INTER*: *ALL x*:*S*. *sym* (*r x*) ==> *sym* (*INTER S r*)
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-UNION*: *ALL x*:*S*. *sym* (*r x*) ==> *sym* (*UNION S r*)
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-diag* [*simp*]: *sym* (*diag A*)
by (*rule symI*) *clarify*

11.8 Transitivity

lemma *transI*:
(!!*x y z*. (*x*, *y*) : *r* ==> (*y*, *z*) : *r* ==> (*x*, *z*) : *r*) ==> *trans r*
by (*unfold trans-def*) *iprover*

lemma *transD*: *trans r* ==> (*a*, *b*) : *r* ==> (*b*, *c*) : *r* ==> (*a*, *c*) : *r*
by (*unfold trans-def*) *iprover*

lemma *trans-Int*: *trans r* ==> *trans s* ==> *trans* (*r* ∩ *s*)
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-INTER*: *ALL x*:*S*. *trans* (*r x*) ==> *trans* (*INTER S r*)
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-diag* [*simp*]: *trans* (*diag A*)
by (*fast intro*: *transI elim*: *transD*)

11.9 Converse

lemma *converse-iff* [*iff*]: ((*a*,*b*): *r*^−*1*) = ((*b*,*a*) : *r*)
by (*simp add*: *converse-def*)

lemma *converseI*[*sym*]: (*a*, *b*) : *r* ==> (*b*, *a*) : *r*^−*1*
by (*simp add*: *converse-def*)

lemma *converseD*[*sym*]: (*a*,*b*) : *r*^−*1* ==> (*b*, *a*) : *r*
by (*simp add*: *converse-def*)

lemma *converseE* [*elim*!]:
yx : *r*^−*1* ==> (!!*x y*. *yx* = (*y*, *x*) ==> (*x*, *y*) : *r* ==> *P*) ==> *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*^−*1*)^−*1* = *r*
by (*unfold converse-def*) *blast*

lemma *converse-rel-comp*: (*r O s*)^−*1* = *s*^−*1 O r*^−*1*
by *blast*

lemma *converse-Int*: (*r* ∩ *s*)^−*1* = *r*^−*1* ∩ *s*^−*1*
by *blast*

lemma *converse-Un*: (*r* ∪ *s*)^−*1* = *r*^−*1* ∪ *s*^−*1*
by *blast*

lemma *converse-INTER*: (*INTER S r*)^−*1* = (*INT x*:*S*. (*r x*)^−*1*)
by *fast*

lemma *converse-UNION*: (*UNION S r*)^−*1* = (*UN x*:*S*. (*r x*)^−*1*)
by *blast*

lemma *converse-Id* [*simp*]: *Id*^−*1* = *Id*
by *blast*

lemma *converse-diag* [*simp*]: (*diag A*)^−*1* = *diag A*
by *blast*

lemma *refl-converse* [*simp*]: *refl A* (*converse r*) = *refl A r*
by (*unfold refl-def*) *auto*

lemma *sym-converse* [*simp*]: *sym* (*converse r*) = *sym r*
by (*unfold sym-def*) *blast*

lemma *antisym-converse* [*simp*]: *antisym* (*converse r*) = *antisym r*
by (*unfold antisym-def*) *blast*

lemma *trans-converse* [*simp*]: *trans* (*converse r*) = *trans r*
by (*unfold trans-def*) *blast*

lemma *sym-conv-converse-eq*: *sym r* = (*r*ˆ−*1* = *r*)
by (*unfold sym-def*) *fast*

lemma *sym-Un-converse*: *sym* (*r* ∪ *r*ˆ−*1*)
by (*unfold sym-def*) *blast*

lemma *sym-Int-converse*: *sym* (*r* ∩ *r*ˆ−*1*)
by (*unfold sym-def*) *blast*

11.10 Domain

declare *Domain-def* [*noatp*]

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

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

lemma *DomainE* [*elim*!]:
a : *Domain r* ==> (!!*y*. (*a*, *y*) : *r* ==> *P*) ==> *P*
by (*iprover dest*!: *iffD1* [*OF Domain-iff*])

lemma *Domain-empty* [*simp*]: *Domain* {} = {}
by *blast*

lemma *Domain-insert*: *Domain* (*insert* (*a*, *b*) *r*) = *insert a* (*Domain r*)
by *blast*

lemma *Domain-Id* [*simp*]: *Domain Id* = *UNIV*
by *blast*

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

lemma *Domain-mono*: $r \subseteq s$ ==> *Domain* $r \subseteq$ *Domain* s
by *blast*

lemma *fst-eq-Domain*: *fst* ' R = *Domain* R
apply *auto*
apply (*rule image-eqI*, *auto*)
done

11.11 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*

lemma *snd-eq-Range*: *snd* ' R = *Range* R

apply *auto*
apply (*rule image-eqI*, *auto*)
done

11.12 Image of a set under a relation

declare *Image-def* [*noatp*]

lemma *Image-iff*: $(b : r``A) = (EX\ x{:}A.\ (x,\ b) : r)$
by (*simp add*: *Image-def*)

lemma *Image-singleton*: $r``\{a\} = \{b.\ (a,\ b) : r\}$
by (*simp add*: *Image-def*)

lemma *Image-singleton-iff* [*iff*]: $(b : r``\{a\}) = ((a,\ b) : r)$
by (*rule Image-iff* [*THEN trans*]) *simp*

lemma *ImageI* [*intro*,*noatp*]: $(a,\ b) : r ==> a : A ==> b : r``A$
by (*unfold Image-def*) *blast*

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

lemma *rev-ImageI*: $a : A ==> (a,\ b) : r ==> b : r\ ``\ A$
— This version's more effective when we already have the required a
by *blast*

lemma *Image-empty* [*simp*]: $R``\{\} = \{\}$
by *blast*

lemma *Image-Id* [*simp*]: $Id\ ``\ A = A$
by *blast*

lemma *Image-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-valued (*converse* R) $==> R\ ``\ (A \cap B) = R\ ``\ A \cap R\ ``\ B$
by (*simp add*: *single-valued-def*, *blast*)

lemma *Image-Un*: $R\ ``\ (A \cup B) = R\ ``\ A \cup R\ ``\ B$
by *blast*

lemma *Un-Image*: $(R \cup S)\ ``\ A = R\ ``\ A \cup S\ ``\ A$
by *blast*

lemma *Image-subset*: $r \subseteq A \times B ==> r``C \subseteq B$
  by (*iprover intro*!: *subsetI elim*!: *ImageE dest*!: *subsetD SigmaD2*)

lemma *Image-eq-UN*: $r``B = (\bigcup y\in B.\ r``\{y\})$
  — NOT suitable for rewriting
  by *blast*

lemma *Image-mono*: $r' \subseteq r ==> A' \subseteq A ==> (r'\ ``\ A') \subseteq (r\ ``\ A)$
  by *blast*

lemma *Image-UN*: $(r\ ``\ (UNION\ A\ B)) = (\bigcup x\in A.\ r\ ``\ (B\ x))$
  by *blast*

lemma *Image-INT-subset*: $(r\ ``\ INTER\ A\ B) \subseteq (\bigcap x\in A.\ r\ ``\ (B\ x))$
  by *blast*

Converse inclusion requires some assumptions

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

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

11.13 Single valued relations

lemma *single-valuedI*:
  $ALL\ x\ y.\ (x,y){:}r\ -->\ (ALL\ z.\ (x,z){:}r\ -->\ y{=}z) ==> single\text{-}valued\ r$
  by (*unfold single-valued-def*)

lemma *single-valuedD*:
  $single\text{-}valued\ r ==> (x,\ y) : r ==> (x,\ z) : r ==> y = z$
  by (*simp add*: *single-valued-def*)

lemma *single-valued-rel-comp*:
  $single\text{-}valued\ r ==> single\text{-}valued\ s ==> single\text{-}valued\ (r\ O\ s)$
  by (*unfold single-valued-def*) *blast*

lemma *single-valued-subset*:
  $r \subseteq s ==> single\text{-}valued\ s ==> single\text{-}valued\ r$
  by (*unfold single-valued-def*) *blast*

lemma *single-valued-Id* [*simp*]: *single-valued Id*
  by (*unfold single-valued-def*) *blast*

lemma *single-valued-diag* [*simp*]: *single-valued* (*diag A*)
by (*unfold single-valued-def*) *blast*

11.14 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*

11.15 Inverse image

lemma *sym-inv-image*: *sym r* ==> *sym* (*inv-image r f*)
by (*unfold sym-def inv-image-def*) *blast*

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

11.16 Version of *lfp-induct* for binary relations

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

end

12 Predicate: Predicates

theory *Predicate*
imports *Inductive Relation*
begin

12.1 Equality and Subsets

lemma *pred-equals-eq* [*pred-set-conv*]: $((\lambda x.\ x \in R) = (\lambda x.\ x \in S)) = (R = S)$
by (*auto simp add*: *expand-fun-eq*)

lemma *pred-equals-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) = (\lambda x\ y.\ (x,\ y) \in S))$
$= (R = S)$
by (*auto simp add*: *expand-fun-eq*)

lemma *pred-subset-eq* [*pred-set-conv*]: $((\lambda x.\ x \in R) <= (\lambda x.\ x \in S)) = (R <= S)$
by *fast*

lemma *pred-subset-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) <= (\lambda x\ y.\ (x,\ y) \in S)) = (R <= S)$
by *fast*

12.2 Top and bottom elements

lemma *top1I* [*intro*!]: *top x*
by (*simp add*: *top-fun-eq top-bool-eq*)

lemma *top2I* [*intro*!]: *top x y*
by (*simp add*: *top-fun-eq top-bool-eq*)

lemma *bot1E* [*elim*!]: *bot x* $\Longrightarrow$ *P*
by (*simp add*: *bot-fun-eq bot-bool-eq*)

lemma *bot2E* [*elim*!]: *bot x y* $\Longrightarrow$ *P*
by (*simp add*: *bot-fun-eq bot-bool-eq*)

12.3 The empty set

lemma *bot-empty-eq*: $bot = (\lambda x.\ x \in \{\})$
by (*auto simp add*: *expand-fun-eq*)

lemma *bot-empty-eq2*: $bot = (\lambda x\ y.\ (x,\ y) \in \{\})$
by (*auto simp add*: *expand-fun-eq*)

12.4 Binary union

lemma *sup1-iff* [*simp*]: $sup\ A\ B\ x \longleftrightarrow A\ x\ |\ B\ x$
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup2-iff* [*simp*]: $sup\ A\ B\ x\ y \longleftrightarrow A\ x\ y\ |\ B\ x\ y$
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup-Un-eq* [*pred-set-conv*]: $sup\ (\lambda x.\ x \in R)\ (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cup S)$
by (*simp add*: *expand-fun-eq*)

lemma *sup-Un-eq2* [*pred-set-conv*]: $sup\ (\lambda x\ y.\ (x,\ y) \in R)\ (\lambda x\ y.\ (x,\ y) \in S) = (\lambda x\ y.\ (x,\ y) \in R \cup S)$
by (*simp add*: *expand-fun-eq*)

lemma *sup1I1* [*elim?*]: $A\ x \Longrightarrow sup\ A\ B\ x$
by *simp*

lemma *sup2I1* [*elim?*]: $A\ x\ y \Longrightarrow sup\ A\ B\ x\ y$

by *simp*

lemma *sup1I2* [*elim?*]: $B\ x \Longrightarrow sup\ A\ B\ x$
by *simp*

lemma *sup2I2* [*elim?*]: $B\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
by *simp*

Classical introduction rule: no commitment to A vs B.

lemma *sup1CI* [*intro!*]: $(\sim\ B\ x ==> A\ x) ==> sup\ A\ B\ x$
by *auto*

lemma *sup2CI* [*intro!*]: $(\sim\ B\ x\ y ==> A\ x\ y) ==> sup\ A\ B\ x\ y$
by *auto*

lemma *sup1E* [*elim!*]: $sup\ A\ B\ x ==> (A\ x ==> P) ==> (B\ x ==> P) ==> P$
by *simp iprover*

lemma *sup2E* [*elim!*]: $sup\ A\ B\ x\ y ==> (A\ x\ y ==> P) ==> (B\ x\ y ==> P) ==> P$
by *simp iprover*

12.5 Binary intersection

lemma *inf1-iff* [*simp*]: $inf\ A\ B\ x \longleftrightarrow A\ x \wedge B\ x$
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf2-iff* [*simp*]: $inf\ A\ B\ x\ y \longleftrightarrow A\ x\ y \wedge B\ x\ y$
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf-Int-eq* [*pred-set-conv*]: $inf\ (\lambda x.\ x \in R)\ (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cap S)$
by (*simp add*: *expand-fun-eq*)

lemma *inf-Int-eq2* [*pred-set-conv*]: $inf\ (\lambda x\ y.\ (x,\ y) \in R)\ (\lambda x\ y.\ (x,\ y) \in S) = (\lambda x\ y.\ (x,\ y) \in R \cap S)$
by (*simp add*: *expand-fun-eq*)

lemma *inf1I* [*intro!*]: $A\ x ==> B\ x ==> inf\ A\ B\ x$
by *simp*

lemma *inf2I* [*intro!*]: $A\ x\ y ==> B\ x\ y ==> inf\ A\ B\ x\ y$
by *simp*

lemma *inf1D1*: $inf\ A\ B\ x ==> A\ x$
by *simp*

lemma *inf2D1*: $inf\ A\ B\ x\ y ==> A\ x\ y$
by *simp*

lemma *inf1D2*: *inf A B x ==> B x*
by *simp*

lemma *inf2D2*: *inf A B x y ==> B x y*
by *simp*

lemma *inf1E* [*elim*!]: *inf A B x ==> (A x ==> B x ==> P) ==> P*
by *simp*

lemma *inf2E* [*elim*!]: *inf A B x y ==> (A x y ==> B x y ==> P) ==> P*
by *simp*

12.6 Unions of families

lemma *SUP1-iff* [*simp*]: (*SUP x*:*A*. *B x*) *b* = (*EX x*:*A*. *B x b*)
by (*simp add*: *SUPR-def Sup-fun-def Sup-bool-def*) *blast*

lemma *SUP2-iff* [*simp*]: (*SUP x*:*A*. *B x*) *b c* = (*EX x*:*A*. *B x b c*)
by (*simp add*: *SUPR-def Sup-fun-def Sup-bool-def*) *blast*

lemma *SUP1-I* [*intro*]: *a* : *A* ==> *B a b* ==> (*SUP x*:*A*. *B x*) *b*
by *auto*

lemma *SUP2-I* [*intro*]: *a* : *A* ==> *B a b c* ==> (*SUP x*:*A*. *B x*) *b c*
by *auto*

lemma *SUP1-E* [*elim*!]: (*SUP x*:*A*. *B x*) *b* ==> (!!*x*. *x* : *A* ==> *B x b* ==> *R*) ==> *R*
by *auto*

lemma *SUP2-E* [*elim*!]: (*SUP x*:*A*. *B x*) *b c* ==> (!!*x*. *x* : *A* ==> *B x b c* ==> *R*) ==> *R*
by *auto*

lemma *SUP-UN-eq*: (*SUP i*. (λ*x*. *x* ∈ *r i*)) = (λ*x*. *x* ∈ (*UN i*. *r i*))
by (*simp add*: *expand-fun-eq*)

lemma *SUP-UN-eq2*: (*SUP i*. (λ*x y*. (*x*, *y*) ∈ *r i*)) = (λ*x y*. (*x*, *y*) ∈ (*UN i*. *r i*))
by (*simp add*: *expand-fun-eq*)

12.7 Intersections of families

lemma *INF1-iff* [*simp*]: (*INF x*:*A*. *B x*) *b* = (*ALL x*:*A*. *B x b*)
by (*simp add*: *INFI-def Inf-fun-def Inf-bool-def*) *blast*

lemma *INF2-iff* [*simp*]: (*INF x*:*A*. *B x*) *b c* = (*ALL x*:*A*. *B x b c*)
by (*simp add*: *INFI-def Inf-fun-def Inf-bool-def*) *blast*

lemma *INF1-I* [*intro*!]: (!!*x*. *x* : *A* ==> *B x b*) ==> (*INF x*:*A*. *B x*) *b*

by *auto*

lemma *INF2-I* [*intro*!]: (!!*x*. *x* : *A* ==> *B x b c*) ==> (*INF x*:*A*. *B x*) *b c*
by *auto*

lemma *INF1-D* [*elim*]: (*INF x*:*A*. *B x*) *b* ==> *a* : *A* ==> *B a b*
by *auto*

lemma *INF2-D* [*elim*]: (*INF x*:*A*. *B x*) *b c* ==> *a* : *A* ==> *B a b c*
by *auto*

lemma *INF1-E* [*elim*]: (*INF x*:*A*. *B x*) *b* ==> (*B a b* ==> *R*) ==> (*a* ~: *A* ==> *R*) ==> *R*
by *auto*

lemma *INF2-E* [*elim*]: (*INF x*:*A*. *B x*) *b c* ==> (*B a b c* ==> *R*) ==> (*a* ~: *A* ==> *R*) ==> *R*
by *auto*

lemma *INF-INT-eq*: (*INF i*. (λ*x*. *x* ∈ *r i*)) = (λ*x*. *x* ∈ (*INT i*. *r i*))
by (*simp add*: *expand-fun-eq*)

lemma *INF-INT-eq2*: (*INF i*. (λ*x y*. (*x*, *y*) ∈ *r i*)) = (λ*x y*. (*x*, *y*) ∈ (*INT i*. *r i*))
by (*simp add*: *expand-fun-eq*)

12.8 Composition of two relations

inductive
pred-comp :: ['*b* => '*c* => *bool*, '*a* => '*b* => *bool*] => '*a* => '*c* => *bool*
(**infixr** *OO 75*)
for *r* :: '*b* => '*c* => *bool* **and** *s* :: '*a* => '*b* => *bool*
where
pred-compI [*intro*]: *s a b* ==> *r b c* ==> (*r OO s*) *a c*

inductive-cases *pred-compE* [*elim*!]: (*r OO s*) *a c*

lemma *pred-comp-rel-comp-eq* [*pred-set-conv*]:
((λ*x y*. (*x*, *y*) ∈ *r*) *OO* (λ*x y*. (*x*, *y*) ∈ *s*)) = (λ*x y*. (*x*, *y*) ∈ *r O s*)
by (*auto simp add*: *expand-fun-eq elim*: *pred-compE*)

12.9 Converse

inductive
conversep :: ('*a* => '*b* => *bool*) => '*b* => '*a* => *bool*
((-^--*1*) [*1000*] *1000*)
for *r* :: '*a* => '*b* => *bool*
where
conversepI: *r a b* ==> *r*^--*1 b a*

notation (*xsymbols*)

conversep $((\text{-}^{-1-1})\ [1000]\ 1000)$

lemma *conversepD*:
assumes *ab*: *r ^−−1 a b*
shows *r b a* **using** *ab*
by *cases simp*

lemma *conversep-iff* [*iff*]: *r ^−−1 a b = r b a*
by (*iprover intro*: *conversepI dest*: *conversepD*)

lemma *conversep-converse-eq* [*pred-set-conv*]:
(λ*x y*. (*x*, *y*) ∈ *r*) ^−−1 = (λ*x y*. (*x*, *y*) ∈ *r* ^−1)
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-conversep* [*simp*]: (*r ^−−1*) ^−−1 = *r*
by (*iprover intro*: *order-antisym conversepI dest*: *conversepD*)

lemma *converse-pred-comp*: (*r OO s*) ^−−1 = *s ^−−1 OO r ^−−1*
by (*iprover intro*: *order-antisym conversepI pred-compI*
elim: *pred-compE dest*: *conversepD*)

lemma *converse-meet*: (*inf r s*) ^−−1 = *inf r ^−−1 s ^−−1*
by (*simp add*: *inf-fun-eq inf-bool-eq*)
(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *converse-join*: (*sup r s*) ^−−1 = *sup r ^−−1 s ^−−1*
by (*simp add*: *sup-fun-eq sup-bool-eq*)
(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *conversep-noteq* [*simp*]: (*op* ~=) ^−−1 = *op* ~=
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-eq* [*simp*]: (*op* =) ^−−1 = *op* =
by (*auto simp add*: *expand-fun-eq*)

12.10 Domain

inductive
DomainP :: ('*a* => '*b* => *bool*) => '*a* => *bool*
for *r* :: '*a* => '*b* => *bool*
where
DomainPI [*intro*]: *r a b* ==> *DomainP r a*

inductive-cases *DomainPE* [*elim*!]: *DomainP r a*

lemma *DomainP-Domain-eq* [*pred-set-conv*]: *DomainP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Domain r*)
by (*blast intro*!: *Orderings.order-antisym*)

12.11 Range

inductive
RangeP :: $('a => 'b => bool) => 'b => bool$
for *r* :: $'a => 'b => bool$
where
RangePI [*intro*]: *r a b ==> RangeP r b*

inductive-cases *RangePE* [*elim*!]: *RangeP r b*

lemma *RangeP-Range-eq* [*pred-set-conv*]: *RangeP* $(\lambda x\ y.\ (x, y) \in r) = (\lambda x.\ x \in$ *Range r*$)$
by (*blast intro*!: *Orderings.order-antisym*)

12.12 Inverse image

definition
inv-imagep :: $('b => 'b => bool) => ('a => 'b) => 'a => 'a => bool$ **where**
inv-imagep r f == %x y. r (f x) (f y)

lemma [*pred-set-conv*]: *inv-imagep* $(\lambda x\ y.\ (x, y) \in r)\ f = (\lambda x\ y.\ (x, y) \in$ *inv-image r f*$)$
by (*simp add*: *inv-image-def inv-imagep-def*)

lemma *in-inv-imagep* [*simp*]: *inv-imagep r f x y = r (f x) (f y)*
by (*simp add*: *inv-imagep-def*)

12.13 The Powerset operator

definition *Powp* :: $('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$ **where**
Powp A $== \lambda B.\ \forall x \in B.\ A\ x$

lemma *Powp-Pow-eq* [*pred-set-conv*]: *Powp* $(\lambda x.\ x \in A) = (\lambda x.\ x \in Pow\ A)$
by (*auto simp add*: *Powp-def expand-fun-eq*)

12.14 Properties of relations - predicate versions

abbreviation *antisymP* :: $('a => 'a => bool) => bool$ **where**
antisymP r == antisym $\{(x, y).\ r\ x\ y\}$

abbreviation *transP* :: $('a => 'a => bool) => bool$ **where**
transP r == trans $\{(x, y).\ r\ x\ y\}$

abbreviation *single-valuedP* :: $('a => 'b => bool) => bool$ **where**
single-valuedP r == single-valued $\{(x, y).\ r\ x\ y\}$

end

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

theory *Transitive-Closure*
imports *Predicate*
uses ~~/*src*/*Provers*/*trancl.ML*
begin

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

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

inductive-set
 $rtrancl :: ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set\ \ ((\text{-}\hat{}*)\ [1000]\ 999)$
 for $r :: ('a \times 'a)\ set$
where
 $rtrancl\text{-}refl\ [intro!,\ Pure.intro!,\ simp]:\ (a,\ a) : r\hat{}*$
 $|\ rtrancl\text{-}into\text{-}rtrancl\ [Pure.intro]:\ (a,\ b) : r\hat{}* \Longrightarrow (b,\ c) : r \Longrightarrow (a,\ c) : r\hat{}*$

inductive-set
 $trancl :: ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set\ \ ((\text{-}\hat{}+)\ [1000]\ 999)$
 for $r :: ('a \times 'a)\ set$
where
 $r\text{-}into\text{-}trancl\ [intro,\ Pure.intro]:\ (a,\ b) : r \Longrightarrow (a,\ b) : r\hat{}+$
 $|\ trancl\text{-}into\text{-}trancl\ [Pure.intro]:\ (a,\ b) : r\hat{}+ \Longrightarrow (b,\ c) : r \Longrightarrow (a,\ c) : r\hat{}+$

notation
 $rtranclp\ \ ((\text{-}\hat{}**)\ [1000]\ 1000)$ **and**
 $tranclp\ \ ((\text{-}\hat{}++)\ [1000]\ 1000)$

abbreviation
 $reflclp :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool\ \ ((\text{-}\hat{}==)\ [1000]\ 1000)$
where
 $r\hat{}== == sup\ r\ op =$

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

notation (*xsymbols*)
 $rtranclp\ \ ((\text{-}^{**})\ [1000]\ 1000)$ **and**
 $tranclp\ \ ((\text{-}^{++})\ [1000]\ 1000)$ **and**
 $reflclp\ \ ((\text{-}^{==})\ [1000]\ 1000)$ **and**
 $rtrancl\ \ ((\text{-}^{*})\ [1000]\ 999)$ **and**
 $trancl\ \ ((\text{-}^{+})\ [1000]\ 999)$ **and**
 $reflcl\ \ ((\text{-}^{=})\ [1000]\ 999)$

notation (*HTML* **output**)

rtranclp ((-**) [1000] 1000) **and**
tranclp ((-$^{++}$) [1000] 1000) **and**
reflclp ((-$^{==}$) [1000] 1000) **and**
rtrancl ((-*) [1000] 999) **and**
trancl ((-$^{+}$) [1000] 999) **and**
reflcl ((-$^{=}$) [1000] 999)

13.1 Reflexive-transitive closure

lemma reflcl-set-eq [pred-set-conv]: (sup (λx y. (x, y) ∈ r) op =) = (λx y. (x, y) ∈ r Un Id)
by (simp add: expand-fun-eq)

lemma r-into-rtrancl [intro]: !!p. p ∈ r ==> p ∈ r^*
— rtrancl of r contains r
apply (simp only: split-tupled-all)
apply (erule rtrancl-refl [THEN rtrancl-into-rtrancl])
done

lemma r-into-rtranclp [intro]: r x y ==> r^** x y
— rtrancl of r contains r
by (erule rtranclp.rtrancl-refl [THEN rtranclp.rtrancl-into-rtrancl])

lemma rtranclp-mono: r ≤ s ==> r^** ≤ s^**
— monotonicity of rtrancl
apply (rule predicate2I)
apply (erule rtranclp.induct)
apply (rule-tac [2] rtranclp.rtrancl-into-rtrancl, blast+)
done

lemmas rtrancl-mono = rtranclp-mono [to-set]

theorem rtranclp-induct [consumes 1, induct set: rtranclp]:
assumes a: r^** a b
and cases: P a !!y z. [| r^** a y; r y z; P y |] ==> P z
shows P b
proof −
from a **have** a = a −−> P b
by (induct %x y. x = a −−> P y a b) (iprover intro: cases)+
thus ?thesis **by** iprover
qed

lemmas rtrancl-induct [induct set: rtrancl] = rtranclp-induct [to-set]

lemmas rtranclp-induct2 =
rtranclp-induct[of - (ax,ay) (bx,by), split-rule,
consumes 1, case-names refl step]

lemmas rtrancl-induct2 =

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

lemma *reflexive-rtrancl*: *reflexive* (*r*^*)
by (*unfold refl-def*) *fast*

lemma *trans-rtrancl*: *trans*(*r*^*)
— 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 *rtranclp-trans*:
assumes *xy*: *r*^** *x y*
and *yz*: *r*^** *y z*
shows *r*^** *x z* **using** *yz xy*
by *induct iprover*+

lemma *rtranclE*:
assumes *major*: (*a*::'*a*,*b*) : *r*^*
and *cases*: (*a* = *b*) ==> *P*
!!*y*. [| (*a*,*y*) : *r*^*; (*y*,*b*) : *r* |] ==> *P*
shows *P*
— elimination of *rtrancl* – by induction on a special formula
apply (*subgoal-tac* (*a*::'*a*) = *b* | (*EX y*. (*a*,*y*) : *r*^* & (*y*,*b*) : *r*))
apply (*rule-tac* [*2*] *major* [*THEN rtrancl-induct*])
prefer *2* **apply** *blast*
prefer *2* **apply** *blast*
apply (*erule asm-rl exE disjE conjE cases*)+
done

lemma *rtrancl-Int-subset*: [| *Id* ⊆ *s*; *r O* (*r*^* ∩ *s*) ⊆ *s*|] ==> *r*^* ⊆ *s*
apply (*rule subsetI*)
apply (*rule-tac p*=*x* **in** *PairE*, *clarify*)
apply (*erule rtrancl-induct*, *auto*)
done

lemma *converse-rtranclp-into-rtranclp*:
$r\ a\ b \Longrightarrow r^{**}\ b\ c \Longrightarrow r^{**}\ a\ c$
by (*rule rtranclp-trans*) *iprover*+

lemmas *converse-rtrancl-into-rtrancl* = *converse-rtranclp-into-rtranclp* [*to-set*]

More r^* equations and inclusions.

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

lemmas *rtrancl-idemp* [*simp*] = *rtranclp-idemp* [*to-set*]

lemma *rtrancl-idemp-self-comp* [*simp*]: $R\hat{}* \ O \ R\hat{}* = R\hat{}*$
apply (*rule set-ext*)
apply (*simp only*: *split-tupled-all*)
apply (*blast intro*: *rtrancl-trans*)
done

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

lemma *rtranclp-subset*: $R \leq S ==> S \leq R\hat{}** ==> S\hat{}** = R\hat{}**$
apply (*drule rtranclp-mono*)
apply (*drule rtranclp-mono*, *simp*)
done

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: (*sup* $(R\hat{}**)$ $(S\hat{}**)$)$\hat{}**$ = (*sup R S*)$\hat{}**$
by (*blast intro*!: *rtranclp-subset intro*: *rtranclp-mono* [*THEN predicate2D*])

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

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

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflcl* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - Id)\hat{}* = r\hat{}*$
apply (*rule sym*)
apply (*rule rtrancl-subset*, *blast*, *clarify*)
apply (*rename-tac a b*)
apply (*case-tac a = b*, *blast*)
apply (*blast intro*!: *r-into-rtrancl*)
done

lemma *rtranclp-r-diff-Id*: (*inf r op* $\sim$=)$\hat{}** = r\hat{}**$
apply (*rule sym*)
apply (*rule rtranclp-subset*)
apply *blast*+
done

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

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

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

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

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

lemma *sym-rtrancl*: *sym r* ==> *sym* $(r\hat{}{*})$
  by (*simp only*: *sym-conv-converse-eq rtrancl-converse* [*symmetric*])

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

lemmas *converse-rtrancl-induct* = *converse-rtranclp-induct* [*to-set*]

lemmas *converse-rtranclp-induct2* =
  converse-rtranclp-induct[*of* - (*ax*,*ay*) (*bx*,*by*), *split-rule*,
        consumes 1, *case-names refl step*]

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

lemma *converse-rtranclpE*:
  assumes *major*: $r\hat{}{**}\ x\ z$
    and *cases*: *x*=*z* ==> *P*

```
    !!y. [| r x y; r^** y z |] ==> P
  shows P
  apply (subgoal-tac x = z | (EX y. r x y & r^** y z))
   apply (rule-tac [2] major [THEN converse-rtranclp-induct])
    prefer 2 apply iprover
   prefer 2 apply iprover
  apply (erule asm-rl exE disjE conjE cases)+
  done

lemmas converse-rtranclE = converse-rtranclpE [to-set]

lemmas converse-rtranclpE2 = converse-rtranclpE [of - (xa,xb) (za,zb), split-rule]

lemmas converse-rtranclE2 = converse-rtranclE [of (xa,xb) (za,zb), split-rule]

lemma r-comp-rtrancl-eq: r O r^* = r^* O r
  by (blast elim: rtranclE converse-rtranclE
    intro: rtrancl-into-rtrancl converse-rtrancl-into-rtrancl)

lemma rtrancl-unfold: r^* = Id Un r O r^*
  by (auto intro: rtrancl-into-rtrancl elim: rtranclE)
```

13.2 Transitive closure

```
lemma trancl-mono: !!p. p ∈ r^+ ==> r ⊆ s ==> p ∈ s^+
  apply (simp add: split-tupled-all)
  apply (erule trancl.induct)
  apply (iprover dest: subsetD)+
  done

lemma r-into-trancl': !!p. p : r ==> p : r^+
  by (simp only: split-tupled-all) (erule r-into-trancl)
```

Conversions between *trancl* and *rtrancl*.

```
lemma tranclp-into-rtranclp: r^++ a b ==> r^** a b
  by (erule tranclp.induct) iprover+

lemmas trancl-into-rtrancl = tranclp-into-rtranclp [to-set]

lemma rtranclp-into-tranclp1: assumes r: r^** a b
  shows !!c. r b c ==> r^++ a c using r
  by induct iprover+

lemmas rtrancl-into-trancl1 = rtranclp-into-tranclp1 [to-set]

lemma rtranclp-into-tranclp2: [| r a b; r^** b c |] ==> r^++ a c
  — intro rule from r and rtrancl
  apply (erule rtranclp.cases, iprover)
  apply (rule rtranclp-trans [THEN rtranclp-into-tranclp1])
```

apply (*simp* | *rule r-into-rtranclp*)+
done

lemmas *rtrancl-into-trancl2* = *rtranclp-into-tranclp2* [*to-set*]

lemma *tranclp-induct* [*consumes 1*, *induct set*: *tranclp*]:
assumes *a*: *r*ˆ++ *a b*
and *cases*: !!*y*. *r a y* ==> *P y*
!!*y z*. *r*ˆ++ *a y* ==> *r y z* ==> *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

lemmas *trancl-induct* [*induct set*: *trancl*] = *tranclp-induct* [*to-set*]

lemmas *tranclp-induct2* =
tranclp-induct[*of* - (*ax*,*ay*) (*bx*,*by*), *split-rule*,
consumes 1, *case-names base step*]

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

lemma *tranclp-trans-induct*:
assumes *major*: *r*ˆ++ *x y*
and *cases*: !!*x y*. *r x y* ==> *P x y*
!!*x y z*. [| *r*ˆ++ *x y*; *P x y*; *r*ˆ++ *y z*; *P y z* |] ==> *P x z*
shows *P x y*
— Another induction rule for trancl, incorporating transitivity
by (*iprover intro*: *major* [*THEN tranclp-induct*] *cases*)

lemmas *trancl-trans-induct* = *tranclp-trans-induct* [*to-set*]

inductive-cases *tranclE*: (*a*, *b*) : *r*ˆ+

lemma *trancl-Int-subset*: [| *r* ⊆ *s*; *r O* (*r*ˆ+ ∩ *s*) ⊆ *s*|] ==> *r*ˆ+ ⊆ *s*
apply (*rule subsetI*)
apply (*rule-tac p*=*x* **in** *PairE*, *clarify*)
apply (*erule trancl-induct*, *auto*)
done

lemma *trancl-unfold*: *r*ˆ+ = *r Un r O r*ˆ+
by (*auto intro*: *trancl-into-trancl elim*: *tranclE*)

lemma *trans-trancl*[*simp*]: *trans*(*r*ˆ+)

— Transitivity of r^+
proof (*rule transI*)
  fix *x y z*
  assume *xy*: (*x*, *y*) ∈ *r*ˆ+
  assume (*y*, *z*) ∈ *r*ˆ+
  thus (*x*, *z*) ∈ *r*ˆ+ **by** *induct* (*insert xy*, *iprover*)+
qed

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

lemma *tranclp-trans*:
  assumes *xy*: *r*ˆ++ *x y*
  and *yz*: *r*ˆ++ *y z*
  shows *r*ˆ++ *x z* **using** *yz xy*
  by *induct iprover*+

lemma *trancl-id*[*simp*]: *trans r* ⟹ *r*ˆ+ = *r*
apply(*auto*)
apply(*erule trancl-induct*)
apply *assumption*
apply(*unfold trans-def*)
apply(*blast*)
done

lemma *rtranclp-tranclp-tranclp*: **assumes** *r*: *r*ˆ∗∗ *x y*
  shows !!*z*. *r*ˆ++ *y z* ==> *r*ˆ++ *x z* **using** *r*
  by *induct* (*iprover intro*: *tranclp-trans*)+

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: *r a b* ==> *r*ˆ++ *b c* ==> *r*ˆ++ *a c*
  by (*erule tranclp-trans* [*OF tranclp.r-into-trancl*])

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *trancl-insert*:
  (*insert* (*y*, *x*) *r*)ˆ+ = *r*ˆ+ ∪ {(*a*, *b*). (*a*, *y*) ∈ *r*ˆ∗ ∧ (*x*, *b*) ∈ *r*ˆ∗}
  — primitive recursion for *trancl* over finite relations
  apply (*rule equalityI*)
   apply (*rule subsetI*)
   apply (*simp only*: *split-tupled-all*)
   apply (*erule trancl-induct*, *blast*)
  apply (*blast intro*: *rtrancl-into-trancl1 trancl-into-rtrancl r-into-trancl trancl-trans*)
  apply (*rule subsetI*)
  apply (*blast intro*: *trancl-mono rtrancl-mono*
    [*THEN* [*2*] *rev-subsetD*] *rtrancl-trancl-trancl rtrancl-into-trancl2*)
  done

lemma *tranclp-converseI*: (*r*ˆ++)ˆ−−*1 x y* ==> (*r*ˆ−−*1*)ˆ++ *x y*

apply (*drule conversepD*)
apply (*erule tranclp-induct*)
apply (*iprover intro*: *conversepI tranclp-trans*)+
done

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

lemma *tranclp-converseD*: $(r\hat{\ }{--}1)\hat{\ }{++}\ x\ y \implies (r\hat{\ }{++})\hat{\ }{--}1\ x\ y$
apply (*rule conversepI*)
apply (*erule tranclp-induct*)
apply (*iprover dest*: *conversepD intro*: *tranclp-trans*)+
done

lemmas *trancl-converseD* = *tranclp-converseD* [*to-set*]

lemma *tranclp-converse*: $(r\hat{\ }{--}1)\hat{\ }{++} = (r\hat{\ }{++})\hat{\ }{--}1$
by (*fastsimp simp add*: *expand-fun-eq*
intro!: *tranclp-converseI dest*!: *tranclp-converseD*)

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

lemma *sym-trancl*: *sym r* ==> *sym* $(r\hat{\ }{+})$
by (*simp only*: *sym-conv-converse-eq trancl-converse* [*symmetric*])

lemma *converse-tranclp-induct*:
assumes *major*: $r\hat{\ }{++}\ a\ b$
and *cases*: !!*y*. *r y b* ==> $P(y)$
!!*y z*.[| *r y z*; $r\hat{\ }{++}\ z\ b$; $P(z)$ |] ==> $P(y)$
shows *P a*
apply (*rule tranclp-induct* [*OF tranclp-converseI*, *OF conversepI*, *OF major*])
apply (*rule cases*)
apply (*erule conversepD*)
apply (*blast intro*: *prems dest*!: *tranclp-converseD conversepD*)
done

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

lemma *tranclpD*: $R\hat{\ }{++}\ x\ y \implies EX\ z.\ R\ x\ z \wedge R\hat{\ }{**}\ z\ y$
apply (*erule converse-tranclp-induct*, *auto*)
apply (*blast intro*: *rtranclp-trans*)
done

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *tranclD2*:
$(x, y) \in R^+ \Longrightarrow \exists z.\ (x, z) \in R^* \wedge (z, y) \in R$
by (*blast elim*: *tranclE intro*: *trancl-into-rtrancl*)

lemma *irrefl-tranclI*: $r\hat{\ }{-}1 \cap r\hat{\ }{*} = \{\} \implies (x, x) \notin r\hat{\ }{+}$

by (*blast elim*: *tranclE dest*: *trancl-into-rtrancl*)

lemma *irrefl-trancl-rD*: !!X. ALL x. $(x, x) \notin r\hat{}+$ ==> $(x, y) \in r$ ==> $x \neq y$
by (*blast dest*: *r-into-trancl*)

lemma *trancl-subset-Sigma-aux*:
$(a, b) \in r\hat{}*$ ==> $r \subseteq A \times A$ ==> $a = b \vee a \in A$
by (*induct rule*: *rtrancl-induct*) *auto*

lemma *trancl-subset-Sigma*: $r \subseteq A \times A$ ==> $r\hat{}+ \subseteq A \times A$
apply (*rule subsetI*)
apply (*simp only*: *split-tupled-all*)
apply (*erule tranclE*)
apply (*blast dest*!: *trancl-into-rtrancl trancl-subset-Sigma-aux*)+
done

lemma *reflcl-tranclp* [*simp*]: $(r\hat{}++)\hat{}== = r\hat{}**$
apply (*safe intro*!: *order-antisym*)
apply (*erule tranclp-into-rtranclp*)
apply (*blast elim*: *rtranclp.cases dest*: *rtranclp-into-tranclp1*)
done

lemmas *reflcl-trancl* [*simp*] = *reflcl-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r\hat{}=)\hat{}+ = r\hat{}*$
apply *safe*
apply (*drule trancl-into-rtrancl*, *simp*)
apply (*erule rtranclE*, *safe*)
apply (*rule r-into-trancl*, *simp*)
apply (*rule rtrancl-into-trancl1*)
apply (*erule rtrancl-reflcl* [*THEN equalityD2*, *THEN subsetD*], *fast*)
done

lemma *trancl-empty* [*simp*]: $\{\}\hat{}+ = \{\}$
by (*auto elim*: *trancl-induct*)

lemma *rtrancl-empty* [*simp*]: $\{\}\hat{}* = Id$
by (*rule subst* [*OF reflcl-trancl*]) *simp*

lemma *rtranclpD*: $R\hat{}**\ a\ b$ ==> $a = b \vee a \neq b \wedge R\hat{}++\ a\ b$
by (*force simp add*: *reflcl-tranclp* [*symmetric*] *simp del*: *reflcl-tranclp*)

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*:
$(x,y) \in R^* = (x=y \vee x \neq y \wedge (x,y) \in R^+)$
by (*fast elim*: *trancl-into-rtrancl dest*: *rtranclD*)

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: *Domain* (R ˆ∗) = *UNIV*
by *blast*

lemma *Range-rtrancl* [*simp*]: *Range* (R ˆ∗) = *UNIV*
by *blast*

lemma *rtrancl-Un-subset*: (R ˆ∗ ∪ S ˆ∗) ⊆ (R *Un* S) ˆ∗
by (*rule rtrancl-Un-rtrancl* [*THEN subst*]) *fast*

lemma *in-rtrancl-UnI*: x ∈ R ˆ∗ ∨ x ∈ S ˆ∗ ==> x ∈ (R ∪ S) ˆ∗
by (*blast intro*: *subsetD* [*OF rtrancl-Un-subset*])

lemma *trancl-domain* [*simp*]: *Domain* (r ˆ+) = *Domain* r
by (*unfold Domain-def*) (*blast dest*: *tranclD*)

lemma *trancl-range* [*simp*]: *Range* (r ˆ+) = *Range* r
by (*simp add*: *Range-def trancl-converse* [*symmetric*])

lemma *Not-Domain-rtrancl*:
x ~: *Domain* R ==> ((x, y) : R ˆ∗) = (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 ˆ∗; (x,z) ∈ r ˆ∗ ⟧
⟹ (y,z) ∈ r ˆ∗ ∨ (z,y) ∈ r ˆ∗
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 ˆ+
by (*fast intro*: *trancl-trans*)

lemma *trancl-into-trancl* [*rule-format*]:
(a, b) ∈ r^+ ==> (b, c) ∈ r −−> (a,c) ∈ r^+
apply (*erule trancl-induct*)
apply (*fast intro*: *r-r-into-trancl*)
apply (*fast intro*: *r-r-into-trancl trancl-trans*)
done

lemma *tranclp-rtranclp-tranclp*:
r^{++} a b ==> r^{**} b c ==> r^{++} a c
apply (*drule tranclpD*)
apply (*erule exE*, *erule conjE*)
apply (*drule rtranclp-trans*, *assumption*)

apply (*drule rtranclp-into-tranclp2*, *assumption*, *assumption*)
done

lemmas *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl.trancl-into-trancl trancl-into-trancl2
rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

declare *rtranclE* [*cases set*: *rtrancl*]
declare *tranclE* [*cases set*: *trancl*]

13.3 Setup of transitivity reasoner

ML-setup ⟪

```
structure Trancl-Tac = Trancl-Tac-Fun (
  struct
    val r-into-trancl = thm trancl.r-into-trancl;
    val trancl-trans  = thm trancl-trans;
    val rtrancl-refl = thm rtrancl.rtrancl-refl;
    val r-into-rtrancl = thm r-into-rtrancl;
    val trancl-into-rtrancl = thm trancl-into-rtrancl;
    val rtrancl-trancl-trancl = thm rtrancl-trancl-trancl;
    val trancl-rtrancl-trancl = thm trancl-rtrancl-trancl;
    val rtrancl-trans = thm rtrancl-trans;

  fun decomp (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);
```

```
structure Tranclp-Tac = Trancl-Tac-Fun (
  struct
    val r-into-trancl = thm tranclp.r-into-trancl;
    val trancl-trans  = thm tranclp-trans;
    val rtrancl-refl = thm rtranclp.rtrancl-refl;
    val r-into-rtrancl = thm r-into-rtranclp;
    val trancl-into-rtrancl = thm tranclp-into-rtranclp;
    val rtrancl-trancl-trancl = thm rtranclp-tranclp-tranclp;
    val trancl-rtrancl-trancl = thm tranclp-rtranclp-tranclp;
    val rtrancl-trans = thm rtranclp-trans;

  fun decomp (Trueprop $ t) =
    let fun dec (rel $ a $ b) =
        let fun decr (Const (Transitive-Closure.rtranclp, - ) $ r) = (r,r*)
              | decr (Const (Transitive-Closure.tranclp, - ) $ r)  = (r,r+)
              | decr r = (r,r);
            val (rel,r) = decr rel;
        in SOME (a, b, Envir.beta-eta-contract rel, r) end
      | dec - =  NONE
    in dec t end;

  end);

change-simpset (fn ss => ss
  addSolver (mk-solver Trancl (fn - => Trancl-Tac.trancl-tac))
  addSolver (mk-solver Rtrancl (fn - => Trancl-Tac.rtrancl-tac))
  addSolver (mk-solver Tranclp (fn - => Tranclp-Tac.trancl-tac))
  addSolver (mk-solver Rtranclp (fn - => Tranclp-Tac.rtrancl-tac)));

⟩⟩
```

method-setup *trancl* =
⟨⟨ *Method.no-args (Method.SIMPLE-METHOD′ Trancl-Tac.trancl-tac)* ⟩⟩
⟨⟨ *simple transitivity reasoner* ⟩⟩
method-setup *rtrancl* =
⟨⟨ *Method.no-args (Method.SIMPLE-METHOD′ Trancl-Tac.rtrancl-tac)* ⟩⟩
⟨⟨ *simple transitivity reasoner* ⟩⟩
method-setup *tranclp* =
⟨⟨ *Method.no-args (Method.SIMPLE-METHOD′ Tranclp-Tac.trancl-tac)* ⟩⟩
⟨⟨ *simple transitivity reasoner (predicate version)* ⟩⟩
method-setup *rtranclp* =
⟨⟨ *Method.no-args (Method.SIMPLE-METHOD′ Tranclp-Tac.rtrancl-tac)* ⟩⟩
⟨⟨ *simple transitivity reasoner (predicate version)* ⟩⟩

end

14 Wellfounded-Recursion: Well-founded Recursion

theory *Wellfounded-Recursion*
imports *Transitive-Closure*
begin

inductive
  wfrec-rel :: $('a * 'a)$ *set* => $(('a => 'b) => 'a => 'b) => 'a => 'b =>$ *bool*
  for R :: $('a * 'a)$ *set*
  and F :: $('a => 'b) => 'a => 'b$
where
  wfrecI: *ALL* z. $(z, x) : R$ --> *wfrec-rel* R F z $(g\ z)$ ==>
      wfrec-rel R F x $(F\ g\ x)$

constdefs
  wf   :: $('a * 'a)$*set* => *bool*
  wf(r) == (!P. (!x. (!y. (y,x):r --> $P(y)$) --> $P(x)$) --> (!x. $P(x)$))

  wfP :: $('a => 'a =>$ *bool*$)$ => *bool*
  wfP r == *wf* $\{(x, y).\ r\ x\ y\}$

  acyclic :: $('a*'a)$*set* => *bool*
  acyclic r == !x. (x,x) ~: r^+

  cut   :: $('a => 'b) => ('a * 'a)$*set* => $'a => 'a => 'b$
  cut f r x == (%y. *if* (y,x):r *then* f y *else 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* => $(('a => 'b) => 'a => 'b) => 'a => 'b$
  [*code func del*]: *wfrec* R F == %x. *THE* y. *wfrec-rel* R (%f x. F (*cut* f R x) x)
x y

abbreviation *acyclicP* :: $('a => 'a =>$ *bool*$)$ => *bool* **where**
  acyclicP r == *acyclic* $\{(x, y).\ r\ x\ y\}$

class *wellorder* = *linorder* +
  assumes *wf*: *wf* $\{(x, y).\ x < y\}$

lemma *wfP-wf-eq* [*pred-set-conv*]: *wfP* $(\lambda x\ y.\ (x, y) \in r)$ = *wf* r
  by (*simp add*: *wfP-def*)

lemma *wfUNIVI*:
   (!!P x. (*ALL* x. (*ALL* y. (y,x) : r --> $P(y)$) --> $P(x)$) ==> $P(x)$) ==>
wf(r)

by (*unfold wf-def*, *blast*)

lemmas *wfPUNIVI* = *wfUNIVI* [*to-pred*]

Restriction to domain A and range B. If r is well-founded over their intersection, then *wf r*

lemma *wfI*:
 [| $r \subseteq A <*> B$;
 !!x P. [|$\forall x$. ($\forall y$. (y,x) : r --> P y) --> P x; x : A; x : B |] ==> 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 *wfP-induct* = *wf-induct* [*to-pred*]

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *consumes 1*, *case-names less*, *induct set*: *wf*]

lemmas *wfP-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set*: *wfP*]

lemma *wf-not-sym* [*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*: *tranclE*)
done

lemmas *wfP-trancl* = *wf-trancl* [*to-pred*]

lemma *wf-converse-trancl*: $wf\ (r\hat{}{-1}) ==> wf\ ((r\hat{}+)\hat{}{-1})$
apply (*subst trancl-converse* [*symmetric*])
apply (*erule wf-trancl*)
done

14.0.1 Other simple well-foundedness results

Minimal-element characterization of well-foundedness

lemma *wf-eq-minimal*: $wf\ r = (\forall\, Q\ x.\ x{\in}Q\ {-}{-}{>}\ (\exists\, z{\in}Q.\ \forall\, y.\ (y{,}z){\in}r\ {-}{-}{>}\ y{\notin}Q))$
proof (*intro iffI strip*)
  fix $Q{::}'a\ set$ **and** x
  assume $wf\ r$ **and** $x \in Q$
  thus $\exists\, z{\in}Q.\ \forall\, y.\ (y,\ z) \in r \longrightarrow y \notin Q$
    by (*unfold wf-def*,
        blast dest: *spec* [*of* - $\%x.\ x{\in}Q \longrightarrow (\exists\, z{\in}Q.\ \forall\, y.\ (y{,}z) \in r \longrightarrow y{\notin}Q)$])
next
  assume *1*: $\forall\, Q\ x.\ x \in Q \longrightarrow (\exists\, z{\in}Q.\ \forall\, y.\ (y,\ z) \in r \longrightarrow y \notin Q)$
  show $wf\ r$
  proof (*rule wfUNIVI*)
    fix $P :: 'a \Rightarrow bool$ **and** x
    assume *2*: $\forall\, x.\ (\forall\, y.\ (y,\ x) \in r \longrightarrow P\ y) \longrightarrow P\ x$
    let $?Q = \{x.\ \neg\ P\ x\}$
    have $x \in ?Q \longrightarrow (\exists\, z{\in}?Q.\ \forall\, y.\ (y,\ z) \in r \longrightarrow y \notin ?Q)$
      by (*rule 1* [*THEN spec*, *THEN spec*])
    hence $\neg\ P\ x \longrightarrow (\exists\, z.\ \neg\ P\ z \wedge (\forall\, y.\ (y,\ z) \in r \longrightarrow P\ y))$ **by** *simp*
    with *2* **have** $\neg\ P\ x \longrightarrow (\exists\, z.\ \neg\ P\ z \wedge P\ z)$ **by** *fast*
    thus $P\ x$ **by** *simp*
  qed
qed

lemma *wfE-min*:
  assumes $p{:}wf\ R\ x \in Q$
  obtains z **where** $z \in Q\ \bigwedge y.\ (y,\ z) \in R \Longrightarrow y \notin Q$
  using p
  unfolding *wf-eq-minimal*
  by *blast*

lemma *wfI-min*:
  $(\bigwedge x\ Q.\ x \in Q \Longrightarrow \exists\, z{\in}Q.\ \forall\, y.\ (y,\ z) \in R \longrightarrow y \notin Q)$
  $\Longrightarrow wf\ R$
  unfolding *wf-eq-minimal*
  by *blast*

lemmas *wfP-eq-minimal* = *wf-eq-minimal* [*to-pred*]

Well-foundedness of subsets

lemma *wf-subset*: $[|\ wf(r);\ \ p{<}{=}r\ |] ==> wf(p)$
apply (*simp* (*no-asm-use*) *add*: *wf-eq-minimal*)

apply *fast*
done

lemmas *wfP-subset* = *wf-subset* [*to-pred*]

Well-foundedness of the empty relation

lemma *wf-empty* [*iff*]: *wf*({})
by (*simp add*: *wf-def*)

lemmas *wfP-empty* [*iff*] =
wf-empty [*to-pred bot-empty-eq2*, *simplified bot-fun-eq bot-bool-eq*]

lemma *wf-Int1*: *wf r* ==> *wf* (*r Int r'*)
by (*erule wf-subset*, *rule Int-lower1*)

lemma *wf-Int2*: *wf r* ==> *wf* (*r' Int r*)
by (*erule wf-subset*, *rule Int-lower2*)

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

14.0.2 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

lemmas *wfP-SUP* = *wf-UN* [**where** *I*=*UNIV* **and** *r*=λ*i*. {(*x*, *y*). *r i x y*},
 to-pred SUP-UN-eq2 bot-empty-eq, *simplified*, *standard*]

lemma *wf-Union*:
 [| *ALL r*:*R*. *wf r*;
   ALL r:*R*. *ALL s*:*R*. *r* ~= *s* --> *Domain r Int Range s* = {}
 |] ==> *wf*(*Union R*)
apply (*simp add*: *Union-def*)
apply (*blast intro*: *wf-UN*)
done

lemma *wf-Un*:
  [| *wf r*; *wf s*; *Domain r Int Range s* = {} |] ==> *wf*(*r Un s*)
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

lemma *wf-union-merge*:
 wf (*R* ∪ *S*) = *wf* (*R O R* ∪ *R O S* ∪ *S*) (**is** *wf ?A* = *wf ?B*)
proof
 assume *wf ?A*
 with *wf-trancl* **have** *wfT*: *wf* (*?A* ^+) **.**
 moreover have *?B* ⊆ *?A* ^+
  by (*subst trancl-unfold*, *subst trancl-unfold*) *blast*
 ultimately show *wf ?B* **by** (*rule wf-subset*)

next
  assume *wf ?B*

  show *wf ?A*
  proof (*rule wfI-min*)
    fix Q :: $'a$ *set* **and** x
    assume $x \in Q$

    with ⟨*wf ?B*⟩
    obtain z **where** $z \in Q$ **and** $\bigwedge y.\ (y, z) \in ?B \Longrightarrow y \notin Q$
      by (*erule wfE-min*)
    hence *A1*: $\bigwedge y.\ (y, z) \in R\ O\ R \Longrightarrow y \notin Q$
      and *A2*: $\bigwedge y.\ (y, z) \in R\ O\ S \Longrightarrow y \notin Q$
      and *A3*: $\bigwedge y.\ (y, z) \in S \Longrightarrow y \notin Q$
      by *auto*

    show $\exists z{\in}Q.\ \forall y.\ (y, z) \in ?A \longrightarrow y \notin Q$
    proof (*cases* $\forall y.\ (y, z) \in R \longrightarrow y \notin Q$)
      case *True*
      with ⟨$z \in Q$⟩ *A3* **show** *?thesis* **by** *blast*
    next
      case *False*
      then obtain z' **where** $z'{\in}Q$ $(z', z) \in R$ **by** *blast*

      have $\forall y.\ (y, z') \in ?A \longrightarrow y \notin Q$
      proof (*intro allI impI*)
        fix y **assume** $(y, z') \in ?A$
        thus $y \notin Q$
        proof
          assume $(y, z') \in R$
          hence $(y, z) \in R\ O\ R$ **using** ⟨$(z', z) \in R$⟩ **..**
          with *A1* **show** $y \notin Q$ **.**
        next
          assume $(y, z') \in S$
          hence $(y, z) \in R\ O\ S$ **using** ⟨$(z', z) \in R$⟩ **..**
          with *A2* **show** $y \notin Q$ **.**
        qed
      qed
      with ⟨$z' \in Q$⟩ **show** *?thesis* **..**
    qed
  qed
qed

lemma *wf-comp-self*: *wf* R = *wf* $(R\ O\ R)$
  by (*fact wf-union-merge*[**where** $S = \{\}$, *simplified*])

14.0.3 acyclic

lemma *acyclicI*: *ALL* $x.\ (x, x)$ ~: r^+ ==> *acyclic* r

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

lemma *wf-acyclic*: *wf r ==> acyclic r*
apply (*simp add*: *acyclic-def*)
apply (*blast elim*: *wf-trancl* [*THEN wf-irrefl*])
done

lemmas *wfP-acyclicP* = *wf-acyclic* [*to-pred*]

lemma *acyclic-insert* [*iff*]:
  acyclic(*insert* (*y*,*x*) *r*) = (*acyclic r* & (*x*,*y*) ~: *r*^*)
apply (*simp add*: *acyclic-def trancl-insert*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *acyclic-converse* [*iff*]: *acyclic*(*r*^−*1*) = *acyclic r*
by (*simp add*: *acyclic-def trancl-converse*)

lemmas *acyclicP-converse* [*iff*] = *acyclic-converse* [*to-pred*]

lemma *acyclic-impl-antisym-rtrancl*: *acyclic r ==> antisym*(*r*^*)
apply (*simp add*: *acyclic-def antisym-def*)
apply (*blast elim*: *rtranclE intro*: *rtrancl-into-trancl1 rtrancl-trancl-trancl*)
done

lemma *acyclic-subset*: [| *acyclic s*; *r <= s* |] ==> *acyclic r*
apply (*simp add*: *acyclic-def*)
apply (*blast intro*: *trancl-mono*)
done

14.1 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*. *wfrec-rel R F x y*
apply (*simp add*: *adm-wf-def*)
apply (*erule-tac a*=*x* **in** *wf-induct*)
apply (*rule ex1I*)
apply (*rule-tac g* = %*x*. *THE y*. *wfrec-rel R F x y* **in** *wfrec-rel.wfrecI*)

apply (*fast dest*!: *theI'*)
apply (*erule wfrec-rel.cases*, *simp*)
apply (*erule allE*, *erule allE*, *erule allE*, *erule mp*)
apply (*fast intro*: *the-equality* [*symmetric*])
done

lemma *adm-lemma*: *adm-wf R* (%*f x*. *F* (*cut f R x*) *x*)
apply (*simp add*: *adm-wf-def*)
apply (*intro strip*)
apply (*rule cuts-eq* [*THEN iffD2*, *THEN subst*], *assumption*)
apply (*rule refl*)
done

lemma *wfrec*: *wf*(*r*) ==> *wfrec r H a* = *H* (*cut* (*wfrec r H*) *r a*) *a*
apply (*simp add*: *wfrec-def*)
apply (*rule adm-lemma* [*THEN wfrec-unique*, *THEN the1-equality*], *assumption*)
apply (*rule wfrec-rel.wfrecI*)
apply (*intro strip*)
apply (*erule adm-lemma* [*THEN wfrec-unique*, *THEN theI'*])
done

* 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

14.2 Code generator setup

consts-code
  wfrec (⟨**module**⟩*wfrec?*)
attach ⟪
fun wfrec f x = *f* (*wfrec f*) *x*;
⟫

14.3 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

14.4 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) --> 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 ==> P (Least P)*
apply (*erule exE*)
apply (*erule LeastI*)
done

lemma *LeastI2*:
[| *P (a::'a::wellorder)*; !!*x. P x ==> Q x* |] *==> Q (Least P)*
by (*blast intro*: *LeastI*)

lemma *LeastI2-ex*:
[| *EX a::'a::wellorder. P a*; !!*x. P x ==> Q x* |] *==> Q (Least P)*
by (*blast intro*: *LeastI-ex*)

lemma *not-less-Least*: [| *k < (LEAST x. P x)* |] *==> ~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

15 OrderedGroup: Ordered Groups

theory *OrderedGroup*
imports *Lattices*
uses ~~/*src/Provers/Arith/abel-cancel.ML*
begin

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

15.1 Semigroups and Monoids

class *semigroup-add* = *plus* +
  assumes *add-assoc*: $(a + b) + c = a + (b + c)$

class *ab-semigroup-add* = *semigroup-add* +
  assumes *add-commute*: $a + b = b + a$
begin

lemma *add-left-commute*: $a + (b + c) = b + (a + c)$
  by (*rule mk-left-commute* [*of plus*, *OF add-assoc add-commute*])

theorems *add-ac* = *add-assoc add-commute add-left-commute*

end

theorems *add-ac* = *add-assoc add-commute add-left-commute*

class *semigroup-mult* = *times* +
  assumes *mult-assoc*: $(a * b) * c = a * (b * c)$

class *ab-semigroup-mult* = *semigroup-mult* +
  assumes *mult-commute*: $a * b = b * a$
begin

lemma *mult-left-commute*: $a * (b * c) = b * (a * c)$
  by (*rule mk-left-commute* [*of times*, *OF mult-assoc mult-commute*])

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

end

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

class *monoid-add* = *zero* + *semigroup-add* +
  assumes *add-0-left* [*simp*]: $0 + a = a$
    and *add-0-right* [*simp*]: $a + 0 = a$

class *comm-monoid-add* = *zero* + *ab-semigroup-add* +
  assumes *add-0*: $0 + a = a$
begin

subclass *monoid-add*
  by *unfold-locales* (*insert add-0*, *simp-all add*: *add-commute*)

end

class *monoid-mult* = *one* + *semigroup-mult* +
  assumes *mult-1-left* [*simp*]: $1 * a = a$
  assumes *mult-1-right* [*simp*]: $a * 1 = a$

class *comm-monoid-mult* = *one* + *ab-semigroup-mult* +
  assumes *mult-1*: $1 * a = a$
begin

subclass *monoid-mult*
  by *unfold-locales* (*insert mult-1*, *simp-all add*: *mult-commute*)

end

class *cancel-semigroup-add* = *semigroup-add* +
  assumes *add-left-imp-eq*: $a + b = a + c \Longrightarrow b = c$
  assumes *add-right-imp-eq*: $b + a = c + a \Longrightarrow b = c$

class *cancel-ab-semigroup-add* = *ab-semigroup-add* +
  assumes *add-imp-eq*: $a + b = a + c \Longrightarrow b = c$
begin

subclass *cancel-semigroup-add*
proof *unfold-locales*
  fix $a\ b\ c :: 'a$
  assume $a + b = a + c$
  then show $b = c$ **by** (*rule add-imp-eq*)
next
  fix $a\ b\ c :: 'a$
  assume $b + a = c + a$
  then have $a + b = a + c$ **by** (*simp only*: *add-commute*)
  then show $b = c$ **by** (*rule add-imp-eq*)
qed

end

context *cancel-ab-semigroup-add*
begin

lemma *add-left-cancel* [*simp*]:
  $a + b = a + c \longleftrightarrow b = c$
  by (*blast dest*: *add-left-imp-eq*)

lemma *add-right-cancel* [*simp*]:
  $b + a = c + a \longleftrightarrow b = c$
  by (*blast dest*: *add-right-imp-eq*)

end

15.2 Groups

class *group-add* = *minus* + *monoid-add* +
  assumes *left-minus* [*simp*]: $- a + a = 0$
  assumes *diff-minus*: $a - b = a + (- b)$
begin

lemma *minus-add-cancel*: $- a + (a + b) = b$
  by (*simp add*: *add-assoc*[*symmetric*])

lemma *minus-zero* [*simp*]: $- 0 = 0$
proof −
  have $- 0 = - 0 + (0 + 0)$ **by** (*simp only*: *add-0-right*)
  also have $\ldots = 0$ **by** (*rule minus-add-cancel*)
  finally show *?thesis* **.**
qed

lemma *minus-minus* [*simp*]: $- (- a) = a$
proof −
  have $- (- a) = - (- a) + (- a + a)$ **by** *simp*
  also have $\ldots = a$ **by** (*rule minus-add-cancel*)
  finally show *?thesis* **.**
qed

lemma *right-minus* [*simp*]: $a + - a = 0$
proof −
  have $a + - a = - (- a) + - a$ **by** *simp*
  also have $\ldots = 0$ **by** (*rule left-minus*)
  finally show *?thesis* **.**
qed

lemma *right-minus-eq*: $a - b = 0 \longleftrightarrow a = b$
proof
  assume $a - b = 0$
  have $a = (a - b) + b$ **by** (*simp add*:*diff-minus add-assoc*)
  also have $\ldots = b$ **using** $\langle a - b = 0\rangle$ **by** *simp*
  finally show $a = b$ **.**
next
  assume $a = b$ **thus** $a - b = 0$ **by** (*simp add*: *diff-minus*)
qed

lemma *equals-zero-I*:
  assumes $a + b = 0$
  shows $- a = b$
proof −
  have $- a = - a + (a + b)$ **using** *assms* **by** *simp*
  also have $\ldots = b$ **by** (*simp add*: *add-assoc*[*symmetric*])
  finally show *?thesis* **.**
qed

lemma *diff-self* [*simp*]: $a - a = 0$
  by (*simp add*: *diff-minus*)

lemma *diff-0* [*simp*]: $0 - a = - a$
  by (*simp add*: *diff-minus*)

lemma *diff-0-right* [*simp*]: $a - 0 = a$
  by (*simp add*: *diff-minus*)

lemma *diff-minus-eq-add* [*simp*]: $a - - b = a + b$
  by (*simp add*: *diff-minus*)

lemma *neg-equal-iff-equal* [*simp*]:
  $- a = - b \longleftrightarrow a = b$
proof
  assume $- a = - b$
  hence $- (- a) = - (- b)$
    by *simp*
  thus $a = b$ **by** *simp*
next
  assume $a = b$
  thus $- a = - b$ **by** *simp*
qed

lemma *neg-equal-0-iff-equal* [*simp*]:
  $- a = 0 \longleftrightarrow a = 0$
  by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

lemma *neg-0-equal-iff-equal* [*simp*]:
  $0 = - a \longleftrightarrow 0 = a$
  by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*:
  $a = - b \longleftrightarrow b = - a$
proof −
  have $- (- a) = - b \longleftrightarrow - a = b$ **by** (*rule neg-equal-iff-equal*)
  thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

lemma *minus-equation-iff*:
  $- a = b \longleftrightarrow - b = a$
proof −
  have $- a = - (- b) \longleftrightarrow a = -b$ **by** (*rule neg-equal-iff-equal*)
  thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

end

class *ab-group-add* = *minus* + *comm-monoid-add* +
  assumes *ab-left-minus*: $- a + a = 0$
  assumes *ab-diff-minus*: $a - b = a + (- b)$
begin

subclass *group-add*
  by *unfold-locales* (*simp-all add*: *ab-left-minus ab-diff-minus*)

subclass *cancel-ab-semigroup-add*
proof *unfold-locales*
  fix $a\ b\ c$ :: $'a$
  assume $a + b = a + c$
  then have $- a + a + b = - a + a + c$
    unfolding *add-assoc* **by** *simp*
  then show $b = c$ **by** *simp*
qed

lemma *uminus-add-conv-diff*:
  $- a + b = b - a$
  by (*simp add*:*diff-minus add-commute*)

lemma *minus-add-distrib* [*simp*]:
  $- (a + b) = - a + - b$
  by (*rule equals-zero-I*) (*simp add*: *add-ac*)

lemma *minus-diff-eq* [*simp*]:
  $- (a - b) = b - a$
  by (*simp add*: *diff-minus add-commute*)

lemma *add-diff-eq*: $a + (b - c) = (a + b) - c$
  by (*simp add*: *diff-minus add-ac*)

lemma *diff-add-eq*: $(a - b) + c = (a + c) - b$
  by (*simp add*: *diff-minus add-ac*)

lemma *diff-eq-eq*: $a - b = c \longleftrightarrow a = c + b$
  by (*auto simp add*: *diff-minus add-assoc*)

lemma *eq-diff-eq*: $a = c - b \longleftrightarrow a + b = c$
  by (*auto simp add*: *diff-minus add-assoc*)

lemma *diff-diff-eq*: $(a - b) - c = a - (b + c)$
  by (*simp add*: *diff-minus add-ac*)

lemma *diff-diff-eq2*: $a - (b - c) = (a + c) - b$
  by (*simp add*: *diff-minus add-ac*)

lemma *diff-add-cancel*: $a - b + b = a$
  by (*simp add*: *diff-minus add-ac*)

lemma *add-diff-cancel*: $a + b - b = a$
by (*simp add*: *diff-minus add-ac*)

lemmas *compare-rls* =
diff-minus [*symmetric*]
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-eq-eq eq-diff-eq

lemma *eq-iff-diff-eq-0*: $a = b \longleftrightarrow a - b = 0$
by (*simp add*: *compare-rls*)

end

15.3 (Partially) Ordered Groups

class *pordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +
assumes *add-left-mono*: $a \le b \Longrightarrow c + a \le c + b$
begin

lemma *add-right-mono*:
$a \le b \Longrightarrow a + c \le b + c$
by (*simp add*: *add-commute* [*of* - *c*] *add-left-mono*)

non-strict, in both arguments

lemma *add-mono*:
$a \le b \Longrightarrow c \le d \Longrightarrow a + c \le b + d$
apply (*erule add-right-mono* [*THEN order-trans*])
apply (*simp add*: *add-commute add-left-mono*)
done

end

class *pordered-cancel-ab-semigroup-add* =
pordered-ab-semigroup-add + *cancel-ab-semigroup-add*
begin

lemma *add-strict-left-mono*:
$a < b \Longrightarrow c + a < c + b$
by (*auto simp add*: *less-le add-left-mono*)

lemma *add-strict-right-mono*:
$a < b \Longrightarrow a + c < b + c$
by (*simp add*: *add-commute* [*of* - *c*] *add-strict-left-mono*)

Strict monotonicity in both arguments

lemma *add-strict-mono*:
$a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$
apply (*erule add-strict-right-mono* [*THEN less-trans*])

apply (*erule add-strict-left-mono*)
done

lemma *add-less-le-mono*:
$a < b \Longrightarrow c \leq d \Longrightarrow a + c < b + d$
apply (*erule add-strict-right-mono* [*THEN less-le-trans*])
apply (*erule add-left-mono*)
done

lemma *add-le-less-mono*:
$a \leq b \Longrightarrow c < d \Longrightarrow a + c < b + d$
apply (*erule add-right-mono* [*THEN le-less-trans*])
apply (*erule add-strict-left-mono*)
done

end

class *pordered-ab-semigroup-add-imp-le* =
pordered-cancel-ab-semigroup-add +
assumes *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
begin

lemma *add-less-imp-less-left*:
assumes *less*: $c + a < c + b$
shows $a < b$
proof −
from *less* **have** *le*: $c + a <= c + b$ **by** (*simp add*: *order-le-less*)
have $a <= b$
apply (*insert le*)
apply (*drule add-le-imp-le-left*)
by (*insert le*, *drule add-le-imp-le-left*, *assumption*)
moreover have $a \neq b$
proof (*rule ccontr*)
assume $\sim(a \neq b)$
then have $a = b$ **by** *simp*
then have $c + a = c + b$ **by** *simp*
with *less* **show** *False***by** *simp*
qed
ultimately show $a < b$ **by** (*simp add*: *order-le-less*)
qed

lemma *add-less-imp-less-right*:
$a + c < b + c \Longrightarrow a < b$
apply (*rule add-less-imp-less-left* [*of c*])
apply (*simp add*: *add-commute*)
done

lemma *add-less-cancel-left* [*simp*]:
$c + a < c + b \longleftrightarrow a < b$

by (*blast intro*: *add-less-imp-less-left add-strict-left-mono*)

lemma *add-less-cancel-right* [*simp*]:
$a + c < b + c \longleftrightarrow a < b$
by (*blast intro*: *add-less-imp-less-right add-strict-right-mono*)

lemma *add-le-cancel-left* [*simp*]:
$c + a \leq c + b \longleftrightarrow a \leq b$
by (*auto*, *drule add-le-imp-le-left*, *simp-all add*: *add-left-mono*)

lemma *add-le-cancel-right* [*simp*]:
$a + c \leq b + c \longleftrightarrow a \leq b$
by (*simp add*: *add-commute* [*of a c*] *add-commute* [*of b c*])

lemma *add-le-imp-le-right*:
$a + c \leq b + c \Longrightarrow a \leq b$
by *simp*

lemma *max-add-distrib-left*:
max $x\ y + z =$ *max* $(x + z)\ (y + z)$
unfolding *max-def* **by** *auto*

lemma *min-add-distrib-left*:
min $x\ y + z =$ *min* $(x + z)\ (y + z)$
unfolding *min-def* **by** *auto*

end

15.4 Support for reasoning about signs

class *pordered-comm-monoid-add* =
pordered-cancel-ab-semigroup-add + *comm-monoid-add*
begin

lemma *add-pos-nonneg*:
assumes $0 < a$ **and** $0 \leq b$
shows $0 < a + b$
proof −
have $0 + 0 < a + b$
using *assms* **by** (*rule add-less-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-pos-pos*:
assumes $0 < a$ **and** $0 < b$
shows $0 < a + b$
by (*rule add-pos-nonneg*) (*insert assms*, *auto*)

lemma *add-nonneg-pos*:

assumes $0 \leq a$ **and** $0 < b$
shows $0 < a + b$
proof –
have $0 + 0 < a + b$
using *assms* **by** (*rule add-le-less-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonneg-nonneg*:
assumes $0 \leq a$ **and** $0 \leq b$
shows $0 \leq a + b$
proof –
have $0 + 0 \leq a + b$
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-neg-nonpos*:
assumes $a < 0$ **and** $b \leq 0$
shows $a + b < 0$
proof –
have $a + b < 0 + 0$
using *assms* **by** (*rule add-less-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-neg-neg*:
assumes $a < 0$ **and** $b < 0$
shows $a + b < 0$
by (*rule add-neg-nonpos*) (*insert assms*, *auto*)

lemma *add-nonpos-neg*:
assumes $a \leq 0$ **and** $b < 0$
shows $a + b < 0$
proof –
have $a + b < 0 + 0$
using *assms* **by** (*rule add-le-less-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonpos-nonpos*:
assumes $a \leq 0$ **and** $b \leq 0$
shows $a + b \leq 0$
proof –
have $a + b \leq 0 + 0$
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

end

class *pordered-ab-group-add* =
  ab-group-add + *pordered-ab-semigroup-add*
begin

subclass *pordered-cancel-ab-semigroup-add*
  by *unfold-locales*

subclass *pordered-ab-semigroup-add-imp-le*
proof *unfold-locales*
  fix $a\ b\ c :: 'a$
  assume $c + a \leq c + b$
  hence $(-c) + (c + a) \leq (-c) + (c + b)$ **by** (*rule add-left-mono*)
  hence $((-c) + c) + a \leq ((-c) + c) + b$ **by** (*simp only*: *add-assoc*)
  thus $a \leq b$ **by** *simp*
qed

subclass *pordered-comm-monoid-add*
  by *unfold-locales*

lemma *max-diff-distrib-left*:
  shows $max\ x\ y - z = max\ (x - z)\ (y - z)$
  by (*simp add*: *diff-minus*, *rule max-add-distrib-left*)

lemma *min-diff-distrib-left*:
  shows $min\ x\ y - z = min\ (x - z)\ (y - z)$
  by (*simp add*: *diff-minus*, *rule min-add-distrib-left*)

lemma *le-imp-neg-le*:
  assumes $a \leq b$
  shows $-b \leq -a$
proof −
  have $-a+a \leq -a+b$
    using ‹$a \leq b$› **by** (*rule add-left-mono*)
  hence $0 \leq -a+b$
    by *simp*
  hence $0 + (-b) \leq (-a + b) + (-b)$
    by (*rule add-right-mono*)
  thus *?thesis*
    by (*simp add*: *add-assoc*)
qed

lemma *neg-le-iff-le* [*simp*]: $-\ b \leq -\ a \longleftrightarrow a \leq b$
proof
  assume $-\ b \leq -\ a$
  hence $-\ (-\ a) \leq -\ (-\ b)$
    by (*rule le-imp-neg-le*)
  thus $a \leq b$ **by** *simp*

next
  assume $a \le b$
  thus $-b \le -a$ **by** (*rule le-imp-neg-le*)
qed

lemma *neg-le-0-iff-le* [*simp*]: $- a \le 0 \longleftrightarrow 0 \le a$
  by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-0-le-iff-le* [*simp*]: $0 \le - a \longleftrightarrow a \le 0$
  by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-less-iff-less* [*simp*]: $- b < - a \longleftrightarrow a < b$
  by (*force simp add*: *less-le*)

lemma *neg-less-0-iff-less* [*simp*]: $- a < 0 \longleftrightarrow 0 < a$
  by (*subst neg-less-iff-less* [*symmetric*], *simp*)

lemma *neg-0-less-iff-less* [*simp*]: $0 < - a \longleftrightarrow a < 0$
  by (*subst neg-less-iff-less* [*symmetric*], *simp*)

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < - b \longleftrightarrow b < - a$
proof −
  have $(- (-a) < - b) = (b < - a)$ **by** (*rule neg-less-iff-less*)
  thus *?thesis* **by** *simp*
qed

lemma *minus-less-iff*: $- a < b \longleftrightarrow - b < a$
proof −
  have $(- a < - (-b)) = (- b < a)$ **by** (*rule neg-less-iff-less*)
  thus *?thesis* **by** *simp*
qed

lemma *le-minus-iff*: $a \le - b \longleftrightarrow b \le - a$
proof −
  have *mm*: !! a (b::$'a$). $(-(-a)) < -b \Longrightarrow -(-b) < -a$ **by** (*simp only*: *minus-less-iff*)
  have $(- (- a) <= -b) = (b <= - a)$
    apply (*auto simp only*: *le-less*)
    apply (*drule mm*)
    apply (*simp-all*)
    apply (*drule mm*[*simplified*], *assumption*)
    done
  then show *?thesis* **by** *simp*
qed

lemma *minus-le-iff*: $- a \le b \longleftrightarrow - b \le a$
  by (*auto simp add*: *le-less minus-less-iff*)

lemma *less-iff-diff-less-0*: $a < b \longleftrightarrow a - b < 0$

proof –
  have $(a < b) = (a + (- b) < b + (-b))$
    by (*simp only*: *add-less-cancel-right*)
  also have ... = $(a - b < 0)$ **by** (*simp add*: *diff-minus*)
  finally show *?thesis* **.**
qed

lemma *diff-less-eq*: $a - b < c \longleftrightarrow a < c + b$
apply (*subst less-iff-diff-less-0* [*of a*])
apply (*rule less-iff-diff-less-0* [*of - c*, *THEN ssubst*])
apply (*simp add*: *diff-minus add-ac*)
done

lemma *less-diff-eq*: $a < c - b \longleftrightarrow a + b < c$
apply (*subst less-iff-diff-less-0* [*of plus a b*])
apply (*subst less-iff-diff-less-0* [*of a*])
apply (*simp add*: *diff-minus add-ac*)
done

lemma *diff-le-eq*: $a - b \leq c \longleftrightarrow a \leq c + b$
  by (*auto simp add*: *le-less diff-less-eq diff-add-cancel add-diff-cancel*)

lemma *le-diff-eq*: $a \leq c - b \longleftrightarrow a + b \leq c$
  by (*auto simp add*: *le-less less-diff-eq diff-add-cancel add-diff-cancel*)

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

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 (**in** –) *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

lemma *le-iff-diff-le-0*: $a \leq b \longleftrightarrow a - b \leq 0$
  by (*simp add*: *compare-rls*)

lemmas *group-simps* =
  add-ac
  add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
  diff-eq-eq eq-diff-eq diff-minus [*symmetric*] *uminus-add-conv-diff*
  diff-less-eq less-diff-eq diff-le-eq le-diff-eq

end

```
lemmas group-simps =
  mult-ac
  add-ac
  add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
  diff-eq-eq eq-diff-eq diff-minus [symmetric] uminus-add-conv-diff
  diff-less-eq less-diff-eq diff-le-eq le-diff-eq

class ordered-ab-semigroup-add =
  linorder + pordered-ab-semigroup-add

class ordered-cancel-ab-semigroup-add =
  linorder + pordered-cancel-ab-semigroup-add
begin

subclass ordered-ab-semigroup-add
  by unfold-locales

subclass pordered-ab-semigroup-add-imp-le
proof unfold-locales
  fix a b c :: 'a
  assume le: c + a <= c + b
  show a <= b
  proof (rule ccontr)
    assume w: ~ a ≤ b
    hence b <= a by (simp add: linorder-not-le)
    hence le2: c + b <= c + a by (rule add-left-mono)
    have a = b
      apply (insert le)
      apply (insert le2)
      apply (drule antisym, simp-all)
      done
    with w show False
      by (simp add: linorder-not-le [symmetric])
  qed
qed

end

class ordered-ab-group-add =
  linorder + pordered-ab-group-add
begin

subclass ordered-cancel-ab-semigroup-add
  by unfold-locales

lemma neg-less-eq-nonneg:
  - a ≤ a ⟷ 0 ≤ a
proof
```

```
  assume A: − a ≤ a show 0 ≤ a
  proof (rule classical)
    assume ¬ 0 ≤ a
    then have a < 0 by auto
    with A have − a < 0 by (rule le-less-trans)
    then show ?thesis by auto
  qed
next
  assume A: 0 ≤ a show − a ≤ a
  proof (rule order-trans)
    show − a ≤ 0 using A by (simp add: minus-le-iff)
  next
    show 0 ≤ a using A .
  qed
qed

lemma less-eq-neg-nonpos:
  a ≤ − a ⟷ a ≤ 0
proof
  assume A: a ≤ − a show a ≤ 0
  proof (rule classical)
    assume ¬ a ≤ 0
    then have 0 < a by auto
    then have 0 < − a using A by (rule less-le-trans)
    then show ?thesis by auto
  qed
next
  assume A: a ≤ 0 show a ≤ − a
  proof (rule order-trans)
    show 0 ≤ − a using A by (simp add: minus-le-iff)
  next
    show a ≤ 0 using A .
  qed
qed

lemma equal-neg-zero:
  a = − a ⟷ a = 0
proof
  assume a = 0 then show a = − a by simp
next
  assume A: a = − a show a = 0
  proof (cases 0 ≤ a)
    case True with A have 0 ≤ − a by auto
    with le-minus-iff have a ≤ 0 by simp
    with True show ?thesis by (auto intro: order-trans)
  next
    case False then have B: a ≤ 0 by auto
    with A have − a ≤ 0 by auto
    with B show ?thesis by (auto intro: order-trans)
```

qed
qed

lemma *neg-equal-zero*:
$- a = a \longleftrightarrow a = 0$
unfolding *equal-neg-zero* [*symmetric*] **by** *auto*

end

— FIXME localize the following

lemma *add-increasing*:
fixes c :: $'a$::{*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*}
shows $[|0 \leq a;\ b \leq c|] ==> b \leq a + c$
by (*insert add-mono* [*of 0 a b c*], *simp*)

lemma *add-increasing2*:
fixes c :: $'a$::{*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*}
shows $[|0 \leq c;\ b \leq a|] ==> b \leq a + c$
by (*simp add*:*add-increasing add-commute*[*of a*])

lemma *add-strict-increasing*:
fixes c :: $'a$::{*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*}
shows $[|0 < a;\ b \leq c|] ==> b < a + c$
by (*insert add-less-le-mono* [*of 0 a b c*], *simp*)

lemma *add-strict-increasing2*:
fixes c :: $'a$::{*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*}
shows $[|0 \leq a;\ b < c|] ==> b < a + c$
by (*insert add-le-less-mono* [*of 0 a b c*], *simp*)

class *pordered-ab-group-add-abs* = *pordered-ab-group-add* + *abs* +
assumes *abs-ge-zero* [*simp*]: $|a| \geq 0$
and *abs-ge-self*: $a \leq |a|$
and *abs-leI*: $a \leq b \Longrightarrow - a \leq b \Longrightarrow |a| \leq b$
and *abs-minus-cancel* [*simp*]: $|-a| = |a|$
and *abs-triangle-ineq*: $|a + b| \leq |a| + |b|$
begin

lemma *abs-minus-le-zero*: $- |a| \leq 0$
unfolding *neg-le-0-iff-le* **by** *simp*

lemma *abs-of-nonneg* [*simp*]:
assumes *nonneg*: $0 \leq a$
shows $|a| = a$
proof (*rule antisym*)
from *nonneg le-imp-neg-le* **have** $- a \leq 0$ **by** *simp*
from *this nonneg* **have** $- a \leq a$ **by** (*rule order-trans*)

then show $|a| \leq a$ **by** (*auto intro*: *abs-leI*)
qed (*rule abs-ge-self*)

lemma *abs-idempotent* [*simp*]: $||a|| = |a|$
by (*rule antisym*)
(*auto intro*!: *abs-ge-self abs-leI order-trans* [*of uminus* (*abs a*) *zero abs a*])

lemma *abs-eq-0* [*simp*]: $|a| = 0 \longleftrightarrow a = 0$
proof −
have $|a| = 0 \Longrightarrow a = 0$
proof (*rule antisym*)
assume *zero*: $|a| = 0$
with *abs-ge-self* **show** $a \leq 0$ **by** *auto*
from *zero* **have** $|-a| = 0$ **by** *simp*
with *abs-ge-self* [*of uminus a*] **have** $- a \leq 0$ **by** *auto*
with *neg-le-0-iff-le* **show** $0 \leq a$ **by** *auto*
qed
then show *?thesis* **by** *auto*
qed

lemma *abs-zero* [*simp*]: $|0| = 0$
by *simp*

lemma *abs-0-eq* [*simp*, *noatp*]: $0 = |a| \longleftrightarrow a = 0$
proof −
have $0 = |a| \longleftrightarrow |a| = 0$ **by** (*simp only*: *eq-ac*)
thus *?thesis* **by** *simp*
qed

lemma *abs-le-zero-iff* [*simp*]: $|a| \leq 0 \longleftrightarrow a = 0$
proof
assume $|a| \leq 0$
then have $|a| = 0$ **by** (*rule antisym*) *simp*
thus $a = 0$ **by** *simp*
next
assume $a = 0$
thus $|a| \leq 0$ **by** *simp*
qed

lemma *zero-less-abs-iff* [*simp*]: $0 < |a| \longleftrightarrow a \neq 0$
by (*simp add*: *less-le*)

lemma *abs-not-less-zero* [*simp*]: $\neg |a| < 0$
proof −
have *a*: $\bigwedge x\ y.\ x \leq y \Longrightarrow \neg\ y < x$ **by** *auto*
show *?thesis* **by** (*simp add*: *a*)
qed

lemma *abs-ge-minus-self*: $- a \leq |a|$

proof −
  have $- a \leq |-a|$ **by** (*rule abs-ge-self*)
  then show *?thesis* **by** *simp*
qed

lemma *abs-minus-commute*:
  $|a - b| = |b - a|$
proof −
  have $|a - b| = |- (a - b)|$ **by** (*simp only*: *abs-minus-cancel*)
  also have ... $= |b - a|$ **by** *simp*
  finally show *?thesis* **.**
qed

lemma *abs-of-pos*: $0 < a \Longrightarrow |a| = a$
  by (*rule abs-of-nonneg*, *rule less-imp-le*)

lemma *abs-of-nonpos* [*simp*]:
  assumes $a \leq 0$
  shows $|a| = - a$
proof −
  let $?b = - a$
  have $- ?b \leq 0 \Longrightarrow |- ?b| = - (- ?b)$
  unfolding *abs-minus-cancel* [*of ?b*]
  unfolding *neg-le-0-iff-le* [*of ?b*]
  unfolding *minus-minus* **by** (*erule abs-of-nonneg*)
  then show *?thesis* **using** *assms* **by** *auto*
qed

lemma *abs-of-neg*: $a < 0 \Longrightarrow |a| = - a$
  by (*rule abs-of-nonpos*, *rule less-imp-le*)

lemma *abs-le-D1*: $|a| \leq b \Longrightarrow a \leq b$
  by (*insert abs-ge-self*, *blast intro*: *order-trans*)

lemma *abs-le-D2*: $|a| \leq b \Longrightarrow - a \leq b$
  by (*insert abs-le-D1* [*of uminus a*], *simp*)

lemma *abs-le-iff*: $|a| \leq b \longleftrightarrow a \leq b \wedge - a \leq b$
  by (*blast intro*: *abs-leI dest*: *abs-le-D1 abs-le-D2*)

lemma *abs-triangle-ineq2*: $|a| - |b| \leq |a - b|$
  apply (*simp add*: *compare-rls*)
  apply (*subgoal-tac abs a = abs* (*plus* (*minus 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*: $||a| - |b|| \leq |a - b|$
  apply (*subst abs-le-iff*)
  apply *auto*
  apply (*rule abs-triangle-ineq2*)
  apply (*subst abs-minus-commute*)
  apply (*rule abs-triangle-ineq2*)
done

lemma *abs-triangle-ineq4*: $|a - b| \leq |a| + |b|$
proof −
  have $abs(a - b) = abs(a + - b)$
    by (*subst diff-minus*, *rule refl*)
  also have ... <= $abs\ a + abs\ (- b)$
    by (*rule abs-triangle-ineq*)
  finally show *?thesis*
    by *simp*
qed

lemma *abs-diff-triangle-ineq*: $|a + b - (c + d)| \leq |a - c| + |b - d|$
proof −
  have $|a + b - (c+d)| = |(a-c) + (b-d)|$ **by** (*simp add*: *diff-minus add-ac*)
  also have ... $\leq |a-c| + |b-d|$ **by** (*rule abs-triangle-ineq*)
  finally show *?thesis* **.**
qed

lemma *abs-add-abs* [*simp*]:
  $||a| + |b|| = |a| + |b|$ (**is** *?L* = *?R*)
proof (*rule antisym*)
  show *?L* $\geq$ *?R* **by**(*rule abs-ge-self*)
next
  have *?L* $\leq ||a|| + ||b||$ **by**(*rule abs-triangle-ineq*)
  also have ... = *?R* **by** *simp*
  finally show *?L* $\leq$ *?R* **.**
qed

end

15.5 Lattice Ordered (Abelian) Groups

class *lordered-ab-group-add-meet* = *pordered-ab-group-add* + *lower-semilattice*
begin

lemma *add-inf-distrib-left*:
  $a + inf\ b\ c = inf\ (a + b)\ (a + c)$
apply (*rule antisym*)
apply (*simp-all add*: *le-infI*)
apply (*rule add-le-imp-le-left* [*of uminus a*])
apply (*simp only*: *add-assoc* [*symmetric*], *simp*)
apply *rule*

apply (*rule add-le-imp-le-left*[*of a*], *simp only*: *add-assoc*[*symmetric*], *simp*)+
done

lemma *add-inf-distrib-right*:
inf a b + *c* = *inf* (*a* + *c*) (*b* + *c*)
proof −
have *c* + *inf a b* = *inf* (*c*+*a*) (*c*+*b*) **by** (*simp add*: *add-inf-distrib-left*)
thus *?thesis* **by** (*simp add*: *add-commute*)
qed

end

class *lordered-ab-group-add-join* = *pordered-ab-group-add* + *upper-semilattice*
begin

lemma *add-sup-distrib-left*:
a + *sup b c* = *sup* (*a* + *b*) (*a* + *c*)
apply (*rule antisym*)
apply (*rule add-le-imp-le-left* [*of uminus a*])
apply (*simp only*: *add-assoc*[*symmetric*], *simp*)
apply *rule*
apply (*rule add-le-imp-le-left* [*of a*], *simp only*: *add-assoc*[*symmetric*], *simp*)+
apply (*rule le-supI*)
apply (*simp-all*)
done

lemma *add-sup-distrib-right*:
sup a b + *c* = *sup* (*a*+*c*) (*b*+*c*)
proof −
have *c* + *sup a b* = *sup* (*c*+*a*) (*c*+*b*) **by** (*simp add*: *add-sup-distrib-left*)
thus *?thesis* **by** (*simp add*: *add-commute*)
qed

end

class *lordered-ab-group-add* = *pordered-ab-group-add* + *lattice*
begin

subclass *lordered-ab-group-add-meet* **by** *unfold-locales*
subclass *lordered-ab-group-add-join* **by** *unfold-locales*

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

lemma *inf-eq-neg-sup*: *inf a b* = − *sup* (−*a*) (−*b*)
proof (*rule inf-unique*)
fix *a b* :: *'a*
show − *sup* (−*a*) (−*b*) ≤ *a*
by (*rule add-le-imp-le-right* [*of - sup* (*uminus a*) (*uminus b*)])

(*simp*, *simp add*: *add-sup-distrib-left*)
next
fix *a b* :: *'a*
show $-$ *sup* $(-a)$ $(-b) \leq b$
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
next
fix *a b c* :: *'a*
assume $a \leq b$ $a \leq c$
then show $a \leq -$ *sup* $(-b)$ $(-c)$ **by** (*subst neg-le-iff-le* [*symmetric*])
(*simp add*: *le-supI*)
qed

lemma *sup-eq-neg-inf*: *sup a b* $= -$ *inf* $(-a)$ $(-b)$
proof (*rule sup-unique*)
fix *a b* :: *'a*
show $a \leq -$ *inf* $(-a)$ $(-b)$
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
next
fix *a b* :: *'a*
show $b \leq -$ *inf* $(-a)$ $(-b)$
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
next
fix *a b c* :: *'a*
assume $a \leq c$ $b \leq c$
then show $-$ *inf* $(-a)$ $(-b) \leq c$ **by** (*subst neg-le-iff-le* [*symmetric*])
(*simp add*: *le-infI*)
qed

lemma *neg-inf-eq-sup*: $-$ *inf a b* $=$ *sup* $(-a)$ $(-b)$
by (*simp add*: *inf-eq-neg-sup*)

lemma *neg-sup-eq-inf*: $-$ *sup a b* $=$ *inf* $(-a)$ $(-b)$
by (*simp add*: *sup-eq-neg-inf*)

lemma *add-eq-inf-sup*: $a + b =$ *sup a b* $+$ *inf a b*
proof $-$
have $0 = -$ *inf* 0 $(a-b)$ $+$ *inf* $(a-b)$ 0 **by** (*simp add*: *inf-commute*)
hence $0 =$ *sup* 0 $(b-a)$ $+$ *inf* $(a-b)$ 0 **by** (*simp add*: *inf-eq-neg-sup*)
hence $0 = (-a +$ *sup a b*$) + ($*inf a b* $+ (-b))$
apply (*simp add*: *add-sup-distrib-left add-inf-distrib-right*)
by (*simp add*: *diff-minus add-commute*)
thus *?thesis*
apply (*simp add*: *compare-rls*)
apply (*subst add-left-cancel* [*symmetric*, *of plus a b plus* (*sup a b*) (*inf a b*) *uminus a*])
apply (*simp only*: *add-assoc*, *simp add*: *add-assoc*[*symmetric*])

done
qed

15.6 Positive Part, Negative Part, Absolute Value

definition
nprt :: *'a* ⇒ *'a* **where**
nprt x = *inf x 0*

definition
pprt :: *'a* ⇒ *'a* **where**
pprt x = *sup x 0*

lemma *pprt-neg*: *pprt* (− *x*) = − *nprt x*
proof −
have *sup* (− *x*) *0* = *sup* (− *x*) (− *0*) **unfolding** *minus-zero* **..**
also have ... = − *inf x 0* **unfolding** *neg-inf-eq-sup* **..**
finally have *sup* (− *x*) *0* = − *inf x 0* **.**
then show *?thesis* **unfolding** *pprt-def nprt-def* **.**
qed

lemma *nprt-neg*: *nprt* (− *x*) = − *pprt x*
proof −
from *pprt-neg* **have** *pprt* (− (− *x*)) = − *nprt* (− *x*) **.**
then have *pprt x* = − *nprt* (− *x*) **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *prts*: *a* = *pprt a* + *nprt a*
by (*simp add*: *pprt-def nprt-def add-eq-inf-sup*[*symmetric*])

lemma *zero-le-pprt*[*simp*]: *0* ≤ *pprt a*
by (*simp add*: *pprt-def*)

lemma *nprt-le-zero*[*simp*]: *nprt a* ≤ *0*
by (*simp add*: *nprt-def*)

lemma *le-eq-neg*: *a* ≤ − *b* ⟷ *a* + *b* ≤ *0* (**is** *?l* = *?r*)
proof −
have *a*: *?l* ⟶ *?r*
apply (*auto*)
apply (*rule add-le-imp-le-right*[*of* - *uminus b* -])
apply (*simp add*: *add-assoc*)
done
have *b*: *?r* ⟶ *?l*
apply (*auto*)
apply (*rule add-le-imp-le-right*[*of* - *b* -])
apply (*simp*)
done

```
  from a b show ?thesis by blast
qed

lemma pprt-0[simp]: pprt 0 = 0 by (simp add: pprt-def)
lemma nprt-0[simp]: nprt 0 = 0 by (simp add: nprt-def)

lemma pprt-eq-id [simp, noatp]: 0 ≤ x ⟹ pprt x = x
  by (simp add: pprt-def le-iff-sup sup-ACI)

lemma nprt-eq-id [simp, noatp]: x ≤ 0 ⟹ nprt x = x
  by (simp add: nprt-def le-iff-inf inf-ACI)

lemma pprt-eq-0 [simp, noatp]: x ≤ 0 ⟹ pprt x = 0
  by (simp add: pprt-def le-iff-sup sup-ACI)

lemma nprt-eq-0 [simp, noatp]: 0 ≤ x ⟹ nprt x = 0
  by (simp add: nprt-def le-iff-inf inf-ACI)

lemma sup-0-imp-0: sup a (− a) = 0 ⟹ a = 0
proof −
  {
    fix a::'a
    assume hyp: sup a (−a) = 0
    hence sup a (−a) + a = a by (simp)
    hence sup (a+a) 0 = a by (simp add: add-sup-distrib-right)
    hence sup (a+a) 0 <= a by (simp)
    hence 0 <= a by (blast intro: order-trans inf-sup-ord)
  }
  note p = this
  assume hyp:sup a (−a) = 0
  hence hyp2:sup (−a) (−(−a)) = 0 by (simp add: sup-commute)
  from p[OF hyp] p[OF hyp2] show a = 0 by simp
qed

lemma inf-0-imp-0: inf a (−a) = 0 ⟹ a = 0
apply (simp add: inf-eq-neg-sup)
apply (simp add: sup-commute)
apply (erule sup-0-imp-0)
done

lemma inf-0-eq-0 [simp, noatp]: inf a (− a) = 0 ⟷ a = 0
  by (rule, erule inf-0-imp-0) simp

lemma sup-0-eq-0 [simp, noatp]: sup a (− a) = 0 ⟷ a = 0
  by (rule, erule sup-0-imp-0) simp

lemma zero-le-double-add-iff-zero-le-single-add [simp]:
  0 ≤ a + a ⟷ 0 ≤ a
proof
```

assume *0 <= a + a*
hence *a*:*inf* (*a+a*) *0* = *0* **by** (*simp add*: *le-iff-inf inf-commute*)
have (*inf a 0*)+(*inf a 0*) = *inf* (*inf* (*a+a*) *0*) *a* (**is** *?l*=-)
by (*simp add*: *add-sup-inf-distribs inf-ACI*)
hence *?l* = *0* + *inf a 0* **by** (*simp add*: *a*, *simp add*: *inf-commute*)
hence *inf a 0* = *0* **by** (*simp only*: *add-right-cancel*)
then show *0* <= *a* **by** (*simp add*: *le-iff-inf inf-commute*)
next
assume *a*: *0* <= *a*
show *0* <= *a* + *a* **by** (*simp add*: *add-mono*[*OF a a*, *simplified*])
qed

lemma *double-zero*: *a* + *a* = *0* ⟷ *a* = *0*
proof
assume *assm*: *a* + *a* = *0*
then have *a* + *a* + − *a* = − *a* **by** *simp*
then have *a* + (*a* + − *a*) = − *a* **by** (*simp only*: *add-assoc*)
then have *a*: − *a* = *a* **by** *simp*
show *a* = *0* **apply** (*rule antisym*)
apply (*unfold neg-le-iff-le* [*symmetric*, *of a*])
unfolding *a* **apply** *simp*
unfolding *zero-le-double-add-iff-zero-le-single-add* [*symmetric*, *of a*]
unfolding *assm* **unfolding** *le-less* **apply** *simp-all* **done**
next
assume *a* = *0* **then show** *a* + *a* = *0* **by** *simp*
qed

lemma *zero-less-double-add-iff-zero-less-single-add*:
0 < *a* + *a* ⟷ *0* < *a*
proof (*cases a* = *0*)
case *True* **then show** *?thesis* **by** *auto*
next
case *False* **then show** *?thesis*
unfolding *less-le* **apply** *simp* **apply** *rule*
apply *clarify*
apply *rule*
apply *assumption*
apply (*rule notI*)
unfolding *double-zero* [*symmetric*, *of a*] **apply** *simp*
done
qed

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]:
a + *a* ≤ *0* ⟷ *a* ≤ *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*]:
$a + a < 0 \longleftrightarrow a < 0$
proof –
have $a + a < 0 \longleftrightarrow 0 < - (a + a)$ **by** (*subst less-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a < 0$ **by** (*simp add*: *zero-less-double-add-iff-zero-less-single-add*)
ultimately show *?thesis* **by** *blast*
qed

declare *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq - a \longleftrightarrow a \leq 0$
proof –
from *add-le-cancel-left* [*of uminus a plus a a zero*]
have $(a <= -a) = (a+a <= 0)$
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *minus-le-self-iff*: $- a \leq a \longleftrightarrow 0 \leq a$
proof –
from *add-le-cancel-left* [*of uminus a zero plus a a*]
have $(-a <= a) = (0 <= a+a)$
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
by (*simp add*: *le-iff-inf nprt-def inf-commute*)

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
by (*simp add*: *le-iff-sup pprt-def sup-commute*)

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$
by (*simp add*: *le-iff-sup pprt-def sup-commute*)

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
by (*simp add*: *le-iff-inf nprt-def inf-commute*)

lemma *pprt-mono* [*simp*, *noatp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$
by (*simp add*: *le-iff-sup pprt-def sup-ACI sup-assoc* [*symmetric*, *of a*])

lemma *nprt-mono* [*simp*, *noatp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
by (*simp add*: *le-iff-inf nprt-def inf-ACI inf-assoc* [*symmetric*, *of a*])

end

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

class *lordered-ab-group-add-abs* = *lordered-ab-group-add* + *abs* +
  assumes *abs-lattice*: $|a| = sup\ a\ (-\ a)$
begin

lemma *abs-prts*: $|a| = pprt\ a - nprt\ a$
proof −
  have $0 \le |a|$
  proof −
    have a: $a \le |a|$ **and** b: $-\ a \le |a|$ **by** (*auto simp add*: *abs-lattice*)
    show *?thesis* **by** (*rule add-mono* [*OF a b*, *simplified*])
  qed
  then have $0 \le sup\ a\ (-\ a)$ **unfolding** *abs-lattice* **.**
  then have $sup\ (sup\ a\ (-\ a))\ 0 = sup\ a\ (-\ a)$ **by** (*rule sup-absorb1*)
  then show *?thesis*
    by (*simp add*: *add-sup-inf-distribs sup-ACI*
      pprt-def nprt-def diff-minus abs-lattice)
qed

subclass *pordered-ab-group-add-abs*
proof −
  have *abs-ge-zero* [*simp*]: $\bigwedge a.\ 0 \le |a|$
  proof −
    fix $a\ b$
    have a: $a \le |a|$ **and** b: $-\ a \le |a|$ **by** (*auto simp add*: *abs-lattice*)
    show $0 \le |a|$ **by** (*rule add-mono* [*OF a b*, *simplified*])
  qed
  have *abs-leI*: $\bigwedge a\ b.\ a \le b \Longrightarrow -\ a \le b \Longrightarrow |a| \le b$
    by (*simp add*: *abs-lattice le-supI*)
  show *?thesis*
  proof *unfold-locales*
    fix a
    show $0 \le |a|$ **by** *simp*
  next
    fix a
    show $a \le |a|$
      by (*auto simp add*: *abs-lattice*)
  next
    fix a
    show $|-a| = |a|$
      by (*simp add*: *abs-lattice sup-commute*)
  next
    fix $a\ b$
    show $a \le b \Longrightarrow -\ a \le b \Longrightarrow |a| \le b$ **by** (*erule abs-leI*)
  next
    fix $a\ b$
    show $|a + b| \le |a| + |b|$
    proof −

have *g*:*abs a* + *abs b* = *sup* (*a*+*b*) (*sup* (−*a*−*b*) (*sup* (−*a*+*b*) (*a* + (−*b*))))
(**is** -=*sup ?m ?n*)
by (*simp add*: *abs-lattice add-sup-inf-distribs sup-ACI diff-minus*)
have *a*:*a*+*b* <= *sup ?m ?n* **by** (*simp*)
have *b*:−*a*−*b* <= *?n* **by** (*simp*)
have *c*:*?n* <= *sup ?m ?n* **by** (*simp*)
from *b c* **have** *d*: −*a*−*b* <= *sup ?m ?n* **by**(*rule order-trans*)
have *e*:−*a*−*b* = −(*a*+*b*) **by** (*simp add*: *diff-minus*)
from *a d e* **have** *abs*(*a*+*b*) <= *sup ?m ?n*
by (*drule-tac abs-leI*, *auto*)
with *g*[*symmetric*] **show** *?thesis* **by** *simp*
qed
qed *auto*
qed

end

lemma *sup-eq-if*:
fixes *a* :: *'a*::{*lordered-ab-group-add*, *linorder*}
shows *sup a* (− *a*) = (*if a* < *0 then* − *a else a*)
proof −
note *add-le-cancel-right* [*of a a* − *a*, *symmetric*, *simplified*]
moreover note *add-le-cancel-right* [*of* −*a a a*, *symmetric*, *simplified*]
then show *?thesis* **by** (*auto simp*: *sup-max max-def*)
qed

lemma *abs-if-lattice*:
fixes *a* :: *'a*::{*lordered-ab-group-add-abs*, *linorder*}
shows |*a*| = (*if a* < *0 then* − *a else a*)
by *auto*

Needed for abelian cancellation simprocs:

lemma *add-cancel-21*: ((*x*::*'a*::*ab-group-add*) + (*y* + *z*) = *y* + *u*) = (*x* + *z* = *u*)
apply (*subst add-left-commute*)
apply (*subst add-left-cancel*)
apply *simp*
done

lemma *add-cancel-end*: (*x* + (*y* + *z*) = *y*) = (*x* = − (*z*::*'a*::*ab-group-add*))
apply (*subst add-cancel-21*[*of - - - 0*, *simplified*])
apply (*simp add*: *add-right-cancel*[*symmetric*, *of x* −*z z*, *simplified*])
done

lemma *less-eqI*: (*x*::*'a*::*pordered-ab-group-add*) − *y* = *x'* − *y'* ⟹ (*x* < *y*) = (*x'* < *y'*)
by (*simp add*: *less-iff-diff-less-0*[*of x y*] *less-iff-diff-less-0*[*of x' y'*])

lemma *le-eqI*: (*x*::*'a*::*pordered-ab-group-add*) − *y* = *x'* − *y'* ⟹ (*y* <= *x*) = (*y'* <= *x'*)

apply (*simp add*: *le-iff-diff-le-0*[*of y x*] *le-iff-diff-le-0*[*of* y' x'])
apply (*simp add*: *neg-le-iff-le*[*symmetric*, *of* $y-x$ *0*] *neg-le-iff-le*[*symmetric*, *of* $y'-x'$ *0*])
done

lemma *eq-eqI*: $(x::'a::ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (x = y) = (x' = y')$
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 *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

lemma *estimate-by-abs*:
  $a + b <= (c::'a::lordered\text{-}ab\text{-}group\text{-}add\text{-}abs) \Longrightarrow a <= c + abs\ b$
proof −
  assume $a+b <= c$
  hence *2*: $a <= c+(-b)$ **by** (*simp add*: *group-simps*)
  have *3*: $(-b) <= abs\ b$ **by** (*rule abs-ge-minus-self*)
  show *?thesis* **by** (*rule le-add-right-mono*[*OF 2 3*])
qed

15.7 Tools setup

lemma *add-mono-thms-ordered-semiring* [*noatp*]:
  fixes *i j k* :: *'a::pordered-ab-semigroup-add*
  shows $i \leq j \wedge k \leq l \Longrightarrow i + k \leq j + l$
    and $i = j \wedge k \leq l \Longrightarrow i + k \leq j + l$
    and $i \leq j \wedge k = l \Longrightarrow i + k \leq j + l$
    and $i = j \wedge k = l \Longrightarrow i + k = j + l$
by (*rule add-mono*, *clarify*+)+

lemma *add-mono-thms-ordered-field* [*noatp*]:
  fixes *i j k* :: *'a::pordered-cancel-ab-semigroup-add*
  shows $i < j \wedge k = l \Longrightarrow i + k < j + l$
    and $i = j \wedge k < l \Longrightarrow i + k < j + l$
    and $i < j \wedge k \leq l \Longrightarrow i + k < j + l$
    and $i \leq j \wedge k < l \Longrightarrow i + k < j + l$
    and $i < j \wedge k < l \Longrightarrow i + k < j + l$

by (*auto intro*: *add-strict-right-mono add-strict-left-mono add-less-le-mono add-le-less-mono add-strict-mono*)

Simplification of $x - y < (0::'a)$, etc.

lemmas *diff-less-0-iff-less* [*simp*] = *less-iff-diff-less-0* [*symmetric*]
lemmas *diff-eq-0-iff-eq* [*simp*, *noatp*] = *eq-iff-diff-eq-0* [*symmetric*]
lemmas *diff-le-0-iff-le* [*simp*] = *le-iff-diff-le-0* [*symmetric*]

ML ⟪
```
structure ab-group-add-cancel = Abel-Cancel(
struct

(* term order for abelian groups *)

fun agrp-ord (Const (a, -)) = find-index (fn a' => a = a')
      [@{const-name HOL.zero}, @{const-name HOL.plus},
        @{const-name HOL.uminus}, @{const-name HOL.minus}]
  | agrp-ord - = ~1;

fun termless-agrp (a, b) = (Term.term-lpo agrp-ord (a, b) = LESS);

local
  val ac1 = mk-meta-eq @{thm add-assoc};
  val ac2 = mk-meta-eq @{thm add-commute};
  val ac3 = mk-meta-eq @{thm add-left-commute};
  fun solve-add-ac thy - (- $ (Const (@{const-name HOL.plus},-) $ - $ -) $ -) =
        SOME ac1
    | solve-add-ac thy - (- $ x $ (Const (@{const-name HOL.plus},-) $ y $ z)) =
        if termless-agrp (y, x) then SOME ac3 else NONE
    | solve-add-ac thy - (- $ x $ y) =
        if termless-agrp (y, x) then SOME ac2 else NONE
    | solve-add-ac thy - - = NONE
in
  val add-ac-proc = Simplifier.simproc @{theory}
    add-ac-proc [x + y::'a::ab-semigroup-add] solve-add-ac;
end;

val cancel-ss = HOL-basic-ss settermless termless-agrp
  addsimprocs [add-ac-proc] addsimps
  [@{thm add-0-left}, @{thm add-0-right}, @{thm diff-def},
   @{thm minus-add-distrib}, @{thm minus-minus}, @{thm minus-zero},
   @{thm right-minus}, @{thm left-minus}, @{thm add-minus-cancel},
   @{thm minus-add-cancel}];

val eq-reflection = @{thm eq-reflection};

val thy-ref = Theory.check-thy @{theory};

val T = @{typ 'a::ab-group-add};
```

val eqI-rules = [@{*thm less-eqI*}, @{*thm le-eqI*}, @{*thm eq-eqI*}];

val dest-eqI =
 fst o HOLogic.dest-bin op = *HOLogic.boolT o HOLogic.dest-Trueprop o concl-of*;

end);
⟫

ML-setup ⟪
 Addsimprocs [*ab-group-add-cancel.sum-conv*, *ab-group-add-cancel.rel-conv*];
⟫

end

16 Ring-and-Field: (Ordered) Rings and Fields

theory *Ring-and-Field*
imports *OrderedGroup*
begin

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +
 assumes *left-distrib*: $(a + b) * c = a * c + b * c$
 assumes *right-distrib*: $a * (b + c) = a * b + a * c$
begin

For the *combine-numerals* simproc

lemma *combine-common-factor*:
 $a * e + (b * e + c) = (a + b) * e + c$
 by (*simp add*: *left-distrib add-ac*)

end

class *mult-zero* = *times* + *zero* +
 assumes *mult-zero-left* [*simp*]: $0 * a = 0$

```
  assumes mult-zero-right [simp]: a * 0 = 0

class semiring-0 = semiring + comm-monoid-add + mult-zero

class semiring-0-cancel = semiring + comm-monoid-add + cancel-ab-semigroup-add
begin

subclass semiring-0
proof unfold-locales
  fix a :: 'a
  have 0 * a + 0 * a = 0 * a + 0
    by (simp add: left-distrib [symmetric])
  thus 0 * a = 0
    by (simp only: add-left-cancel)
next
  fix a :: 'a
  have a * 0 + a * 0 = a * 0 + 0
    by (simp add: right-distrib [symmetric])
  thus a * 0 = 0
    by (simp only: add-left-cancel)
qed

end

class comm-semiring = ab-semigroup-add + ab-semigroup-mult +
  assumes distrib: (a + b) * c = a * c + b * c
begin

subclass semiring
proof unfold-locales
  fix a b c :: 'a
  show (a + b) * c = a * c + b * c by (simp add: distrib)
  have a * (b + c) = (b + c) * a by (simp add: mult-ac)
  also have ... = b * a + c * a by (simp only: distrib)
  also have ... = a * b + a * c by (simp add: mult-ac)
  finally show a * (b + c) = a * b + a * c by blast
qed

end

class comm-semiring-0 = comm-semiring + comm-monoid-add + mult-zero
begin

subclass semiring-0 by unfold-locales

end

class comm-semiring-0-cancel = comm-semiring + comm-monoid-add + cancel-ab-semigroup-add
begin
```

subclass *semiring-0-cancel* **by** *unfold-locales*

end

class *zero-neq-one* = *zero* + *one* +
  assumes *zero-neq-one* [*simp*]: $0 \neq 1$

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult*

begin

subclass *semiring-1* **by** *unfold-locales*

end

class *no-zero-divisors* = *zero* + *times* +
  assumes *no-zero-divisors*: $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$

class *semiring-1-cancel* = *semiring* + *comm-monoid-add* + *zero-neq-one*
  + *cancel-ab-semigroup-add* + *monoid-mult*
begin

subclass *semiring-0-cancel* **by** *unfold-locales*

subclass *semiring-1* **by** *unfold-locales*

end

class *comm-semiring-1-cancel* = *comm-semiring* + *comm-monoid-add* + *comm-monoid-mult*
  + *zero-neq-one* + *cancel-ab-semigroup-add*
begin

subclass *semiring-1-cancel* **by** *unfold-locales*
subclass *comm-semiring-0-cancel* **by** *unfold-locales*
subclass *comm-semiring-1* **by** *unfold-locales*

end

class *ring* = *semiring* + *ab-group-add*
begin

subclass *semiring-0-cancel* **by** *unfold-locales*

Distribution rules

lemma *minus-mult-left*: $-(a * b) = - a * b$
  by (*rule equals-zero-I*) (*simp add*: *left-distrib* [*symmetric*])

lemma *minus-mult-right*: $- (a * b) = a * - b$
by (*rule equals-zero-I*) (*simp add*: *right-distrib* [*symmetric*])

lemma *minus-mult-minus* [*simp*]: $- a * - b = a * b$
by (*simp add*: *minus-mult-left* [*symmetric*] *minus-mult-right* [*symmetric*])

lemma *minus-mult-commute*: $- a * b = a * - b$
by (*simp add*: *minus-mult-left* [*symmetric*] *minus-mult-right* [*symmetric*])

lemma *right-diff-distrib*: $a * (b - c) = a * b - a * c$
by (*simp add*: *right-distrib diff-minus*
minus-mult-left [*symmetric*] *minus-mult-right* [*symmetric*])

lemma *left-diff-distrib*: $(a - b) * c = a * c - b * c$
by (*simp add*: *left-distrib diff-minus*
minus-mult-left [*symmetric*] *minus-mult-right* [*symmetric*])

lemmas *ring-distribs* =
right-distrib left-distrib left-diff-distrib right-diff-distrib

lemmas *ring-simps* =
add-ac
add-diff-eq diff-add-eq diff-diff-eq diff-diff-eq2
diff-eq-eq eq-diff-eq diff-minus [*symmetric*] *uminus-add-conv-diff*
ring-distribs

lemma *eq-add-iff1*:
$a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
by (*simp add*: *ring-simps*)

lemma *eq-add-iff2*:
$a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$
by (*simp add*: *ring-simps*)

end

lemmas *ring-distribs* =
right-distrib left-distrib left-diff-distrib right-diff-distrib

class *comm-ring* = *comm-semiring* + *ab-group-add*
begin

subclass *ring* **by** *unfold-locales*
subclass *comm-semiring-0* **by** *unfold-locales*

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*

begin

subclass *semiring-1-cancel* **by** *unfold-locales*

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*

begin

subclass *ring-1* **by** *unfold-locales*
subclass *comm-semiring-1-cancel* **by** *unfold-locales*

end

class *ring-no-zero-divisors* = *ring* + *no-zero-divisors*
begin

lemma *mult-eq-0-iff* [*simp*]:
  shows $a * b = 0 \longleftrightarrow (a = 0 \vee b = 0)$
proof (*cases* $a = 0 \vee b = 0$)
  case *False* **then have** $a \neq 0$ **and** $b \neq 0$ **by** *auto*
    then show *?thesis* **using** *no-zero-divisors* **by** *simp*
next
  case *True* **then show** *?thesis* **by** *auto*
qed

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*

class *idom* = *comm-ring-1* + *no-zero-divisors*
begin

subclass *ring-1-no-zero-divisors* **by** *unfold-locales*

end

class *division-ring* = *ring-1* + *inverse* +
  assumes *left-inverse* [*simp*]: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
  assumes *right-inverse* [*simp*]: $a \neq 0 \Longrightarrow a * inverse\ a = 1$
begin

subclass *ring-1-no-zero-divisors*
proof *unfold-locales*
  fix $a\ b :: 'a$
  assume a: $a \neq 0$ **and** b: $b \neq 0$
  show $a * b \neq 0$
  proof

```
    assume ab: a * b = 0
    hence 0 = inverse a * (a * b) * inverse b
      by simp
    also have ... = (inverse a * a) * (b * inverse b)
      by (simp only: mult-assoc)
    also have ... = 1
      using a b by simp
    finally show False
      by simp
  qed
qed

end

class field = comm-ring-1 + inverse +
  assumes field-inverse:  a ≠ 0 ⟹ inverse a * a = 1
  assumes divide-inverse: a / b = a * inverse b
begin

subclass division-ring
proof unfold-locales
  fix a :: 'a
  assume a ≠ 0
  thus inverse a * a = 1 by (rule field-inverse)
  thus a * inverse a = 1 by (simp only: mult-commute)
qed

subclass idom by unfold-locales

lemma right-inverse-eq: b ≠ 0 ⟹ a / b = 1 ⟷ a = b
proof
  assume neq: b ≠ 0
  {
    hence a = (a / b) * b by (simp add: divide-inverse mult-ac)
    also assume a / b = 1
    finally show a = b by simp
  next
    assume a = b
    with neq show a / b = 1 by (simp add: divide-inverse)
  }
qed

lemma nonzero-inverse-eq-divide: a ≠ 0 ⟹ inverse a = 1 / a
  by (simp add: divide-inverse)

lemma divide-self [simp]: a ≠ 0 ⟹ a / a = 1
  by (simp add: divide-inverse)

lemma divide-zero-left [simp]: 0 / a = 0
```

by (*simp add*: *divide-inverse*)

lemma *inverse-eq-divide*: *inverse a = 1 / a*
by (*simp add*: *divide-inverse*)

lemma *add-divide-distrib*: *(a+b) / c = a/c + b/c*
by (*simp add*: *divide-inverse ring-distribs*)

end

class *division-by-zero = zero + inverse +*
assumes *inverse-zero* [*simp*]: *inverse 0 = 0*

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*)

class *mult-mono = times + zero + ord +*
assumes *mult-left-mono*: $a \le b \Longrightarrow 0 \le c \Longrightarrow c * a \le c * b$
assumes *mult-right-mono*: $a \le b \Longrightarrow 0 \le c \Longrightarrow a * c \le b * c$

class *pordered-semiring = mult-mono + semiring-0 + pordered-ab-semigroup-add*

begin

lemma *mult-mono*:
$a \le b \Longrightarrow c \le d \Longrightarrow 0 \le b \Longrightarrow 0 \le c$
$\Longrightarrow a * c \le b * d$
apply (*erule mult-right-mono* [*THEN order-trans*], *assumption*)
apply (*erule mult-left-mono*, *assumption*)
done

lemma *mult-mono'*:
$a \le b \Longrightarrow c \le d \Longrightarrow 0 \le a \Longrightarrow 0 \le c$
$\Longrightarrow a * c \le b * d$
apply (*rule mult-mono*)
apply (*fast intro*: *order-trans*)+
done

end

class *pordered-cancel-semiring = mult-mono + pordered-ab-semigroup-add*
+ semiring + comm-monoid-add + cancel-ab-semigroup-add
begin

subclass *semiring-0-cancel* **by** *unfold-locales*
subclass *pordered-semiring* **by** *unfold-locales*

lemma *mult-nonneg-nonneg*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * b$
by (*drule mult-left-mono* [*of zero b*], *auto*)

lemma *mult-nonneg-nonpos*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow a * b \leq 0$
by (*drule mult-left-mono* [*of b zero*], *auto*)

lemma *mult-nonneg-nonpos2*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow b * a \leq 0$
by (*drule mult-right-mono* [*of b zero*], *auto*)

lemma *split-mult-neg-le*: $(0 \leq a \ \& \ b \leq 0) \mid (a \leq 0 \ \& \ 0 \leq b) \Longrightarrow a * b \leq (0::-::pordered\text{-}cancel\text{-}semiring)$
by (*auto simp add*: *mult-nonneg-nonpos mult-nonneg-nonpos2*)

end

class *ordered-semiring* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add* + *mult-mono*
begin

subclass *pordered-cancel-semiring* **by** *unfold-locales*

subclass *pordered-comm-monoid-add* **by** *unfold-locales*

lemma *mult-left-less-imp-less*:
$c * a < c * b \Longrightarrow 0 \leq c \Longrightarrow a < b$
by (*force simp add*: *mult-left-mono not-le* [*symmetric*])

lemma *mult-right-less-imp-less*:
$a * c < b * c \Longrightarrow 0 \leq c \Longrightarrow a < b$
by (*force simp add*: *mult-right-mono not-le* [*symmetric*])

end

class *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add* +
assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* **by** *unfold-locales*

subclass *ordered-semiring*
proof *unfold-locales*
fix $a\ b\ c :: 'a$
assume A: $a \leq b\ 0 \leq c$
from A **show** $c * a \leq c * b$

unfolding *le-less*
using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
from *A* **show** $a * c \leq b * c$
unfolding *le-less*
using *mult-strict-right-mono* **by** (*cases* $c = 0$) *auto*
qed

lemma *mult-left-le-imp-le*:
$c * a \leq c * b \Longrightarrow 0 < c \Longrightarrow a \leq b$
by (*force simp add*: *mult-strict-left-mono -not-less* [*symmetric*])

lemma *mult-right-le-imp-le*:
$a * c \leq b * c \Longrightarrow 0 < c \Longrightarrow a \leq b$
by (*force simp add*: *mult-strict-right-mono not-less* [*symmetric*])

lemma *mult-pos-pos*:
$0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
by (*drule mult-strict-left-mono* [*of zero b*], *auto*)

lemma *mult-pos-neg*:
$0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$
by (*drule mult-strict-left-mono* [*of b zero*], *auto*)

lemma *mult-pos-neg2*:
$0 < a \Longrightarrow b < 0 \Longrightarrow b * a < 0$
by (*drule mult-strict-right-mono* [*of b zero*], *auto*)

lemma *zero-less-mult-pos*:
$0 < a * b \Longrightarrow 0 < a \Longrightarrow 0 < b$
apply (*cases* $b \leq 0$)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

lemma *zero-less-mult-pos2*:
$0 < b * a \Longrightarrow 0 < a \Longrightarrow 0 < b$
apply (*cases* $b \leq 0$)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg2* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

end

class *mult-mono1* = *times* + *zero* + *ord* +
assumes *mult-mono1*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$

class *pordered-comm-semiring* = *comm-semiring-0*

+ *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-semiring*
proof *unfold-locales*
fix $a\ b\ c$ :: $'a$
assume $a \leq b$ $0 \leq c$
thus $c * a \leq c * b$ **by** (*rule mult-mono1*)
thus $a * c \leq b * c$ **by** (*simp only*: *mult-commute*)
qed

end

class *pordered-cancel-comm-semiring* = *comm-semiring-0-cancel*
+ *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-comm-semiring* **by** *unfold-locales*
subclass *pordered-cancel-semiring* **by** *unfold-locales*

end

class *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add*
+
assumes *mult-strict-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *ordered-semiring-strict*
proof *unfold-locales*
fix $a\ b\ c$ :: $'a$
assume $a < b$ $0 < c$
thus $c * a < c * b$ **by** (*rule mult-strict-mono*)
thus $a * c < b * c$ **by** (*simp only*: *mult-commute*)
qed

subclass *pordered-cancel-comm-semiring*
proof *unfold-locales*
fix $a\ b\ c$ :: $'a$
assume $a \leq b$ $0 \leq c$
thus $c * a \leq c * b$
unfolding *le-less*
using *mult-strict-mono* **by** (*cases* $c = 0$) *auto*
qed

end

class *pordered-ring* = *ring* + *pordered-cancel-semiring*
begin

subclass *pordered-ab-group-add* **by** *unfold-locales*

lemmas *ring-simps* = *ring-simps group-simps*

lemma *less-add-iff1*:
$a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$
by (*simp add*: *ring-simps*)

lemma *less-add-iff2*:
$a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$
by (*simp add*: *ring-simps*)

lemma *le-add-iff1*:
$a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$
by (*simp add*: *ring-simps*)

lemma *le-add-iff2*:
$a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$
by (*simp add*: *ring-simps*)

lemma *mult-left-mono-neg*:
$b \leq a \Longrightarrow c \leq 0 \Longrightarrow c * a \leq c * b$
apply (*drule mult-left-mono* [*of - - uminus c*])
apply (*simp-all add*: *minus-mult-left* [*symmetric*])
done

lemma *mult-right-mono-neg*:
$b \leq a \Longrightarrow c \leq 0 \Longrightarrow a * c \leq b * c$
apply (*drule mult-right-mono* [*of - - uminus c*])
apply (*simp-all add*: *minus-mult-right* [*symmetric*])
done

lemma *mult-nonpos-nonpos*:
$a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a * b$
by (*drule mult-right-mono-neg* [*of a zero b*]) *auto*

lemma *split-mult-pos-le*:
$(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \Longrightarrow 0 \leq a * b$
by (*auto simp add*: *mult-nonneg-nonneg mult-nonpos-nonpos*)

end

class *abs-if* = *minus* + *ord* + *zero* + *abs* +
assumes *abs-if*: $|a| = (\textit{if } a < 0 \textit{ then } (- a) \textit{ else } a)$

class *sgn-if* = *sgn* + *zero* + *one* + *minus* + *ord* +
assumes *sgn-if*: $sgn\ x = (\textit{if } x = 0 \textit{ then } 0 \textit{ else if } 0 < x \textit{ then } 1 \textit{ else } - 1)$

class *ordered-ring* = *ring* + *ordered-semiring*

$+$ *ordered-ab-group-add* $+$ *abs-if*
begin

subclass *pordered-ring* **by** *unfold-locales*

subclass *pordered-ab-group-add-abs*
proof *unfold-locales*
fix *a b*
show $|a + b| \leq |a| + |b|$
by (*auto simp add*: *abs-if not-less neg-less-eq-nonneg less-eq-neg-nonpos*)
(*auto simp del*: *minus-add-distrib simp add*: *minus-add-distrib* [*symmetric*]
neg-less-eq-nonneg less-eq-neg-nonpos, *auto intro*: *add-nonneg-nonneg*,
auto intro!: *less-imp-le add-neg-neg*)
qed (*auto simp add*: *abs-if less-eq-neg-nonpos neg-equal-zero*)

end

class *ordered-ring-strict* = *ring* $+$ *ordered-semiring-strict*
$+$ *ordered-ab-group-add* $+$ *abs-if*
begin

subclass *ordered-ring* **by** *unfold-locales*

lemma *mult-strict-left-mono-neg*:
$b < a \Longrightarrow c < 0 \Longrightarrow c * a < c * b$
apply (*drule mult-strict-left-mono* [*of* - - *uminus c*])
apply (*simp-all add*: *minus-mult-left* [*symmetric*])
done

lemma *mult-strict-right-mono-neg*:
$b < a \Longrightarrow c < 0 \Longrightarrow a * c < b * c$
apply (*drule mult-strict-right-mono* [*of* - - *uminus c*])
apply (*simp-all add*: *minus-mult-right* [*symmetric*])
done

lemma *mult-neg-neg*:
$a < 0 \Longrightarrow b < 0 \Longrightarrow 0 < a * b$
by (*drule mult-strict-right-mono-neg*, *auto*)

end

instance *ordered-ring-strict* $\subseteq$ *ring-no-zero-divisors*
apply *intro-classes*
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-less-mult-iff*:

fixes a :: $'a$::*ordered-ring-strict*
shows $0 < a * b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
apply (*auto simp add*: *le-less not-less mult-pos-pos mult-neg-neg*)
apply (*blast dest*: *zero-less-mult-pos*)
apply (*blast dest*: *zero-less-mult-pos2*)
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 *zero-le-square*[*simp*]: $(0::'a::ordered\text{-}ring\text{-}strict) \leq a*a$
by (*simp add*: *zero-le-mult-iff linorder-linear*)

lemma *not-square-less-zero*[*simp*]: $\neg\, (a * a < (0::'a::ordered\text{-}ring\text{-}strict))$
by (*simp add*: *not-less*)

This list of rewrites simplifies ring terms by multiplying everything out and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides ring equalities but also helps with inequalities.

lemmas *ring-simps* = *group-simps ring-distribs*

class *pordered-comm-ring* = *comm-ring* + *pordered-comm-semiring*
begin

subclass *pordered-ring* **by** *unfold-locales*
subclass *pordered-cancel-comm-semiring* **by** *unfold-locales*

end

class *ordered-semidom* = *comm-semiring-1-cancel* + *ordered-comm-semiring-strict* +

assumes *zero-less-one* [*simp*]: $0 < 1$
begin

lemma *pos-add-strict*:
shows $0 < a \Longrightarrow b < c \Longrightarrow b < a + c$
using *add-strict-mono* [*of zero a b c*] **by** *simp*

end

class *ordered-idom* =
comm-ring-1 +
ordered-comm-semiring-strict +
ordered-ab-group-add +
abs-if + *sgn-if*

instance *ordered-idom* ⊆ *ordered-ring-strict* **..**

instance *ordered-idom* ⊆ *pordered-comm-ring* **..**

class *ordered-field* = *field* + *ordered-idom*

lemma *linorder-neqE-ordered-idom*:
fixes *x y* :: *'a* :: *ordered-idom*
assumes $x \neq y$ **obtains** $x < y \mid y < x$
using *assms* **by** (*rule linorder-neqE*)

Proving axiom *zero-less-one* makes all *ordered-semidom* theorems available to members of *ordered-idom*

instance *ordered-idom* ⊆ *ordered-semidom*
proof
have (*0*::*'a*) ≤ *1*∗*1* **by** (*rule zero-le-square*)
thus (*0*::*'a*) < *1* **by** (*simp add*: *order-le-less*)
qed

instance *ordered-idom* ⊆ *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-semidom*) ≤ *1*
by (*rule zero-less-one* [*THEN order-less-imp-le*])

lemma *not-one-le-zero* [*simp*]: ~ (*1*::*'a*::*ordered-semidom*) ≤ *0*
by (*simp add*: *linorder-not-le*)

lemma *not-one-less-zero* [*simp*]: ~ (*1*::*'a*::*ordered-semidom*) < *0*
by (*simp add*: *linorder-not-less*)

16.1 More Monotonicity

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
$[|a<b;\ c<d;\ 0<b;\ 0\le c|] ==> a * c < b * (d::'a::ordered\text{-}semiring\text{-}strict)$
apply (*cases 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'*:
$[|\ a<b;\ c<d;\ 0 \le a;\ 0 \le c|] ==> a * c < b * (d::'a::ordered\text{-}semiring\text{-}strict)$
apply (*rule mult-strict-mono*)
apply (*blast intro*: *order-le-less-trans*)+
done

lemma *less-1-mult*: $[|\ 1 < m;\ 1 < n\ |] ==> 1 < m*(n::'a::ordered\text{-}semidom)$
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*: $(a::'a::ordered\text{-}semiring\text{-}strict) < b ==>$
$c <= d ==> 0 <= a ==> 0 < c ==> a * c < b * d$
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*: $(a::'a::ordered\text{-}semiring\text{-}strict) <= b ==>$
$c < d ==> 0 < a ==> 0 <= c ==> a * c < b * d$
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

16.2 Cancellation Laws for Relationships With a Common Factor

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations $\le$ 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 (*cases 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 (*cases 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-ring-strict*
  shows $(a*c < b*c) = ((0 \le c \ {-}{-}{>} \ a < b) \ \& \ (c \le 0 \ {-}{-}{>} \ b < a))$
by (*insert mult-less-cancel-right-disj* [*of a c b*], *auto*)

lemma *mult-less-cancel-left*:
  fixes *c* :: *'a* :: *ordered-ring-strict*
  shows $(c*a < c*b) = ((0 \le c \ {-}{-}{>} \ a < b) \ \& \ (c \le 0 \ {-}{-}{>} \ b < a))$
by (*insert mult-less-cancel-left-disj* [*of c a b*], *auto*)

lemma *mult-le-cancel-right*:
  $(a*c \le b*c) = ((0{<}c \ {-}{-}{>} \ a{\le}b) \ \& \ (c{<}0 \ {-}{-}{>} \ b \le (a::'a::ordered\text{-}ring\text{-}strict)))$
by (*simp add*: *linorder-not-less* [*symmetric*] *mult-less-cancel-right-disj*)

lemma *mult-le-cancel-left*:
  $(c*a \le c*b) = ((0{<}c \ {-}{-}{>} \ a{\le}b) \ \& \ (c{<}0 \ {-}{-}{>} \ b \le (a::'a::ordered\text{-}ring\text{-}strict)))$
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 ≤ c
    shows a < (b::'a::ordered-semiring-strict)
proof (rule ccontr)
  assume ~ a < b
  hence b ≤ a by (simp add: linorder-not-less)
  hence c*b ≤ c*a using nonneg 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-semiring-strict)
proof (rule ccontr)
  assume ~ a < b
  hence b ≤ a by (simp add: linorder-not-less)
  hence b*c ≤ a*c using nonneg 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,noatp]:
  fixes a b c :: 'a::ring-no-zero-divisors
  shows (a * c = b * c) = (c = 0 ∨ a = b)
proof -
  have (a * c = b * c) = ((a - b) * c = 0)
    by (simp add: ring-distribs)
  thus ?thesis
    by (simp add: disj-commute)
qed

lemma mult-cancel-left [simp,noatp]:
  fixes a b c :: 'a::ring-no-zero-divisors
  shows (c * a = c * b) = (c = 0 ∨ a = b)
proof -
  have (c * a = c * b) = (c * (a - b) = 0)
    by (simp add: ring-distribs)
  thus ?thesis
    by simp
qed
```

16.2.1 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-idom
  shows (c ≤ b*c) = ((0<c --> 1≤b) & (c<0 --> b ≤ 1))
by (insert mult-le-cancel-right [of 1 c b], simp)
```

lemma *mult-le-cancel-right2*:
fixes c :: $'a$ :: *ordered-idom*
shows $(a*c \leq c) = ((0<c \dashrightarrow a\leq 1)$ & $(c<0 \dashrightarrow 1 \leq a))$
by (*insert mult-le-cancel-right* [*of a c 1*], *simp*)

lemma *mult-le-cancel-left1*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c \leq c*b) = ((0<c \dashrightarrow 1\leq b)$ & $(c<0 \dashrightarrow b \leq 1))$
by (*insert mult-le-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-le-cancel-left2*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c*a \leq c) = ((0<c \dashrightarrow a\leq 1)$ & $(c<0 \dashrightarrow 1 \leq a))$
by (*insert mult-le-cancel-left* [*of c a 1*], *simp*)

lemma *mult-less-cancel-right1*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c < b*c) = ((0 \leq c \dashrightarrow 1<b)$ & $(c \leq 0 \dashrightarrow b < 1))$
by (*insert mult-less-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-less-cancel-right2*:
fixes c :: $'a$ :: *ordered-idom*
shows $(a*c < c) = ((0 \leq c \dashrightarrow a<1)$ & $(c \leq 0 \dashrightarrow 1 < a))$
by (*insert mult-less-cancel-right* [*of a c 1*], *simp*)

lemma *mult-less-cancel-left1*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c < c*b) = ((0 \leq c \dashrightarrow 1<b)$ & $(c \leq 0 \dashrightarrow b < 1))$
by (*insert mult-less-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-less-cancel-left2*:
fixes c :: $'a$ :: *ordered-idom*
shows $(c*a < c) = ((0 \leq c \dashrightarrow a<1)$ & $(c \leq 0 \dashrightarrow 1 < a))$
by (*insert mult-less-cancel-left* [*of c a 1*], *simp*)

lemma *mult-cancel-right1* [*simp*]:
fixes c :: $'a$ :: *ring-1-no-zero-divisors*
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 :: $'a$ :: *ring-1-no-zero-divisors*
shows $(a*c = c) = (c = 0 \mid a=1)$
by (*insert mult-cancel-right* [*of a c 1*], *simp*)

lemma *mult-cancel-left1* [*simp*]:
fixes c :: $'a$ :: *ring-1-no-zero-divisors*
shows $(c = c*b) = (c = 0 \mid b=1)$

by (*insert mult-cancel-left* [*of c 1 b*], *force*)

lemma *mult-cancel-left2* [*simp*]:
  fixes $c :: 'a :: ring\text{-}1\text{-}no\text{-}zero\text{-}divisors$
  shows $(c*a = c) = (c = 0 \mid a=1)$
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

lemma *nonzero-imp-inverse-nonzero*:
  $a \neq 0 \implies inverse\ a \neq (0::'a::division\text{-}ring)$
proof
  assume *ianz*: $inverse\ a = 0$
  assume $a \neq 0$
  hence $1 = a * inverse\ a$ **by** *simp*
  also have $... = 0$ **by** (*simp add*: *ianz*)
  finally have $1 = (0::'a::division\text{-}ring)$ **.**
  thus *False* **by** (*simp add*: *eq-commute*)
qed

16.3 Basic Properties of *inverse*

lemma *inverse-zero-imp-zero*: $inverse\ a = 0 \implies a = (0::'a::division\text{-}ring)$
apply (*rule ccontr*)
apply (*blast dest*: *nonzero-imp-inverse-nonzero*)
done

lemma *inverse-nonzero-imp-nonzero*:
   $inverse\ a = 0 \implies a = (0::'a::division\text{-}ring)$
apply (*rule ccontr*)
apply (*blast dest*: *nonzero-imp-inverse-nonzero*)
done

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
   $(inverse\ a = 0) = (a = (0::'a::\{division\text{-}ring,division\text{-}by\text{-}zero\}))$
by (*force dest*: *inverse-nonzero-imp-nonzero*)

lemma *nonzero-inverse-minus-eq*:
  assumes [*simp*]: $a \neq 0$
  shows $inverse(-a) = -inverse(a::'a::division\text{-}ring)$
proof −
 have $-a * inverse\ (-\ a) = -a * -\ inverse\ a$
  by *simp*
 thus *?thesis*
  by (*simp only*: *mult-cancel-left*, *simp*)
qed

lemma *inverse-minus-eq* [*simp*]:
 $inverse(-a) = -inverse(a::'a::\{division\text{-}ring,division\text{-}by\text{-}zero\})$
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 $a = (b::'a::division\text{-}ring)$
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*:
 $inverse\ a = inverse\ b\ ==>\ a = (b::'a::\{division\text{-}ring,division\text{-}by\text{-}zero\})$
apply (*cases* $a=0\ |\ b=0$)
 apply (*force dest*!: *inverse-zero-imp-zero*
    simp add: *eq-commute* [*of* $0::'a$])
apply (*force dest*!: *nonzero-inverse-eq-imp-eq*)
done

lemma *inverse-eq-iff-eq* [*simp*]:
 $(inverse\ a = inverse\ b) = (a = (b::'a::\{division\text{-}ring,division\text{-}by\text{-}zero\}))$
by (*force dest*!: *inverse-eq-imp-eq*)

lemma *nonzero-inverse-inverse-eq*:
  assumes [*simp*]: $a \neq 0$
  shows $inverse(inverse\ (a::'a::division\text{-}ring)) = a$

```
  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::{division-ring,division-by-zero})) = a
  proof cases
    assume a=0 thus ?thesis by simp
  next
    assume a≠0
    thus ?thesis by (simp add: nonzero-inverse-inverse-eq)
  qed

lemma inverse-1 [simp]: inverse 1 = (1::'a::division-ring)
  proof -
  have inverse 1 * 1 = (1::'a::division-ring)
    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::division-ring)
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::division-ring)
  proof -
  have inverse(a*b) * (a * b) * inverse(b) = inverse(b)
    by (simp add: 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 $a \neq 0$ & $b \neq 0$
  thus *?thesis*
   by (*simp add*: *nonzero-inverse-mult-distrib mult-commute*)
 next
  assume ~ ($a \neq 0$ & $b \neq 0$)
  thus *?thesis*
   by *force*
 qed

lemma *division-ring-inverse-add*:
 [|(*a*::'*a*::*division-ring*) $\neq 0$; $b \neq 0$|]
  ==> *inverse a* + *inverse b* = *inverse a* * (*a*+*b*) * *inverse b*
by (*simp add*: *ring-simps*)

lemma *division-ring-inverse-diff*:
 [|(*a*::'*a*::*division-ring*) $\neq 0$; $b \neq 0$|]
  ==> *inverse a* − *inverse b* = *inverse a* * (*b*−*a*) * *inverse b*
by (*simp add*: *ring-simps*)

There is no slick version using division by zero.

lemma *inverse-add*:
 [|$a \neq 0$; $b \neq 0$|]
  ==> *inverse a* + *inverse b* = (*a*+*b*) * *inverse a* * *inverse* (*b*::'*a*::*field*)
by (*simp add*: *division-ring-inverse-add mult-ac*)

lemma *inverse-divide* [*simp*]:
 inverse (*a*/*b*) = *b* / (*a*::'*a*::{*field*,*division-by-zero*})
by (*simp add*: *divide-inverse mult-commute*)

16.4 Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

lemma *nonzero-mult-divide-mult-cancel-left*[*simp*,*noatp*]:
assumes [*simp*]: $b \neq 0$ **and** [*simp*]: $c \neq 0$ **shows** (*c**a)/(*c**b) = *a*/(*b*::'*a*::*field*)
proof −
 have (*c**a)/(*c**b) = *c* * *a* * (*inverse b* * *inverse c*)
  by (*simp add*: *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-mult-cancel-left*:
  $c \neq 0$ ==> $(c*a)$ / $(c*b) = a$ / $(b::'a::\{field,division\text{-}by\text{-}zero\})$
apply (*cases* $b = 0$)
apply (*simp-all add*: *nonzero-mult-divide-mult-cancel-left*)
done

lemma *nonzero-mult-divide-mult-cancel-right* [*noatp*]:
  $[|b \neq 0;\ c \neq 0|]$ ==> $(a*c)$ / $(b*c) = a/(b::'a::field)$
by (*simp add*: *mult-commute* [*of - c*] *nonzero-mult-divide-mult-cancel-left*)

lemma *mult-divide-mult-cancel-right*:
  $c \neq 0$ ==> $(a*c)$ / $(b*c) = a$ / $(b::'a::\{field,division\text{-}by\text{-}zero\})$
apply (*cases* $b = 0$)
apply (*simp-all add*: *nonzero-mult-divide-mult-cancel-right*)
done

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*)

lemmas *times-divide-eq* = *times-divide-eq-right times-divide-eq-left*

lemma *divide-divide-eq-right* [*simp*,*noatp*]:
  a / $(b/c) = (a*c)$ / $(b::'a::\{field,division\text{-}by\text{-}zero\})$
by (*simp add*: *divide-inverse mult-ac*)

lemma *divide-divide-eq-left* [*simp*,*noatp*]:
  $(a$ / $b)$ / $(c::'a::\{field,division\text{-}by\text{-}zero\}) = a$ / $(b*c)$
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-mult-cancel-left* [*THEN sym*])
apply *assumption*
apply (*erule nonzero-mult-divide-mult-cancel-right* [*THEN sym*])
apply *assumption*
done

16.4.1 Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if*[*simp*,*noatp*]:
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-mult-cancel-left*)

lemma *nonzero-mult-divide-cancel-right*[*simp*,*noatp*]:
$b \neq 0 \Longrightarrow a * b \ / \ b = (a{::}'a{::}field)$
using *nonzero-mult-divide-mult-cancel-right*[*of 1 b a*] **by** *simp*

lemma *nonzero-mult-divide-cancel-left*[*simp*,*noatp*]:
$a \neq 0 \Longrightarrow a * b \ / \ a = (b{::}'a{::}field)$
using *nonzero-mult-divide-mult-cancel-left*[*of 1 a b*] **by** *simp*

lemma *nonzero-divide-mult-cancel-right*[*simp*,*noatp*]:
$[\![\ a\neq 0; \ b\neq 0 \ ]\!] \Longrightarrow b \ / \ (a * b) = 1/(a{::}'a{::}field)$
using *nonzero-mult-divide-mult-cancel-right*[*of a b 1*] **by** *simp*

lemma *nonzero-divide-mult-cancel-left*[*simp*,*noatp*]:
$[\![\ a\neq 0; \ b\neq 0 \ ]\!] \Longrightarrow a \ / \ (a * b) = 1/(b{::}'a{::}field)$
using *nonzero-mult-divide-mult-cancel-left*[*of b a 1*] **by** *simp*

lemma *nonzero-mult-divide-mult-cancel-left2*[*simp*,*noatp*]:
$[|b\neq 0; \ c\neq 0|]$ ==> $(c*a)$ / $(b*c)$ = $a/(b{::}'a{::}field)$
using *nonzero-mult-divide-mult-cancel-left*[*of b c a*] **by**(*simp add*:*mult-ac*)

lemma *nonzero-mult-divide-mult-cancel-right2*[*simp*,*noatp*]:
$[|b\neq 0; \ c\neq 0|]$ ==> $(a*c)$ / $(c*b)$ = $a/(b{::}'a{::}field)$
using *nonzero-mult-divide-mult-cancel-right*[*of b c a*] **by**(*simp add*:*mult-ac*)

16.5 Division and Unary Minus

lemma *nonzero-minus-divide-left*: $b \neq 0$ ==> $-$ (a/b) = $(-a)$ / $(b{::}'a{::}field)$
by (*simp add*: *divide-inverse minus-mult-left*)

lemma *nonzero-minus-divide-right*: $b \neq 0$ ==> $-$ (a/b) = a / $-(b{::}'a{::}field)$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq minus-mult-right*)

lemma *nonzero-minus-divide-divide*: $b \neq 0$ ==> $(-a)/(-b)$ = 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 (*cases* $b=0$, *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 *add-divide-eq-iff*:
  $(z::'a::field) \neq 0 \Longrightarrow x + y/z = (z*x + y)/z$
by(*simp add*:*add-divide-distrib nonzero-mult-divide-cancel-left*)

lemma *divide-add-eq-iff*:
  $(z::'a::field) \neq 0 \Longrightarrow x/z + y = (x + z*y)/z$
by(*simp add*:*add-divide-distrib nonzero-mult-divide-cancel-left*)

lemma *diff-divide-eq-iff*:
  $(z::'a::field) \neq 0 \Longrightarrow x - y/z = (z*x - y)/z$
by(*simp add*:*diff-divide-distrib nonzero-mult-divide-cancel-left*)

lemma *divide-diff-eq-iff*:
  $(z::'a::field) \neq 0 \Longrightarrow x/z - y = (x - z*y)/z$
by(*simp add*:*diff-divide-distrib nonzero-mult-divide-cancel-left*)

lemma *nonzero-eq-divide-eq*: $c\neq 0$ ==> $((a::'a::field) = b/c) = (a*c = b)$
proof −
  assume [*simp*]: $c\neq 0$
  have $(a = b/c) = (a*c = (b/c)*c)$ **by** *simp*
  also have ... $= (a*c = b)$ **by** (*simp add*: *divide-inverse mult-assoc*)
  finally show *?thesis* **.**
qed

lemma *nonzero-divide-eq-eq*: $c\neq 0$ ==> $(b/c = (a::'a::field)) = (b = a*c)$
proof −
  assume [*simp*]: $c\neq 0$
  have $(b/c = a) = ((b/c)*c = a*c)$ **by** *simp*
  also have ... $= (b = a*c)$ **by** (*simp add*: *divide-inverse mult-assoc*)
  finally show *?thesis* **.**

qed

lemma *eq-divide-eq*:
$((a::'a::\{field,division\text{-}by\text{-}zero\}) = b/c) = (if\ c{\neq}0\ then\ a{*}c = b\ else\ a{=}0)$
by (*simp add*: *nonzero-eq-divide-eq*)

lemma *divide-eq-eq*:
$(b/c = (a::'a::\{field,division\text{-}by\text{-}zero\})) = (if\ c{\neq}0\ then\ b = a{*}c\ else\ a{=}0)$
by (*force simp add*: *nonzero-divide-eq-eq*)

lemma *divide-eq-imp*: $(c::'a::\{division\text{-}by\text{-}zero,field\})$ ~= *0* ==>
$b = a * c$ ==> $b\ /\ c = a$
by (*subst divide-eq-eq*, *simp*)

lemma *eq-divide-imp*: $(c::'a::\{division\text{-}by\text{-}zero,field\})$ ~= *0* ==>
$a * c = b$ ==> $a = b\ /\ c$
by (*subst eq-divide-eq*, *simp*)

lemmas *field-eq-simps* = *ring-simps*

add-divide-eq-iff divide-add-eq-iff
diff-divide-eq-iff divide-diff-eq-iff

nonzero-eq-divide-eq nonzero-divide-eq-eq

An example:

lemma fixes $a\ b\ c\ d\ e\ f$ :: $'a::field$
shows $[\![a{\neq}b;\ c{\neq}d;\ e{\neq}f\ ]\!] \Longrightarrow ((a-b)*(c-d)*(e-f))/((c-d)*(e-f)*(a-b)) = 1$
apply(*subgoal-tac* $(c-d)*(e-f)*(a-b) \neq 0$)
apply(*simp add*:*field-eq-simps*)
apply(*simp*)
done

lemma *diff-frac-eq*: $(y::'a::field)$ ~= *0* ==> z ~= *0* ==>
$x\ /\ y - w\ /\ z = (x * z - w * y)\ /\ (y * z)$
by (*simp add*:*field-eq-simps times-divide-eq*)

lemma *frac-eq-eq*: $(y::'a::field)$ ~= *0* ==> z ~= *0* ==>
$(x\ /\ y = w\ /\ z) = (x * z = w * y)$
by (*simp add*:*field-eq-simps times-divide-eq*)

16.6 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*:
$a < 0 ==> inverse\ a < (0::'a::ordered\text{-}field)$
by (*insert positive-imp-inverse-positive* [*of* $-a$],
simp add: *nonzero-inverse-minus-eq order-less-imp-not-eq*)

lemma *inverse-le-imp-le*:
assumes *invle*: $inverse\ a \leq inverse\ b$ **and** *apos*: $0 < a$
shows $b \leq (a::'a::ordered\text{-}field)$
proof (*rule classical*)
assume $\sim b \leq a$
hence $a < b$ **by** (*simp add*: *linorder-not-le*)
hence *bpos*: $0 < b$ **by** (*blast intro*: *apos order-less-trans*)
hence $a * inverse\ a \leq a * inverse\ b$
by (*simp add*: *apos invle order-less-imp-le mult-left-mono*)
hence $(a * inverse\ a) * b \leq (a * inverse\ b) * b$
by (*simp add*: *bpos order-less-imp-le mult-right-mono*)
thus $b \leq a$ **by** (*simp add*: *mult-assoc apos bpos order-less-imp-not-eq2*)
qed

lemma *inverse-positive-imp-positive*:
assumes *inv-gt-0*: $0 < inverse\ a$ **and** *nz*: $a \neq 0$
shows $0 < (a::'a::ordered\text{-}field)$
proof –
have $0 < inverse\ (inverse\ a)$
using *inv-gt-0* **by** (*rule positive-imp-inverse-positive*)
thus $0 < a$
using *nz* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-positive-iff-positive* [*simp*]:
$(0 < inverse\ a) = (0 < (a::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
apply (*cases* $a = 0$, *simp*)
apply (*blast intro*: *inverse-positive-imp-positive positive-imp-inverse-positive*)
done

lemma *inverse-negative-imp-negative*:
assumes *inv-less-0*: $inverse\ a < 0$ **and** *nz*: $a \neq 0$
shows $a < (0::'a::ordered\text{-}field)$
proof –
have $inverse\ (inverse\ a) < 0$
using *inv-less-0* **by** (*rule negative-imp-inverse-negative*)
thus $a < 0$ **using** *nz* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-negative-iff-negative* [*simp*]:
  (*inverse* $a < 0$) = ($a <$ (0::′*a*::{*ordered-field*,*division-by-zero*}))
apply (*cases* $a = 0$, *simp*)
apply (*blast intro*: *inverse-negative-imp-negative negative-imp-inverse-negative*)
done

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
  ($0 \leq$ *inverse* a) = ($0 \leq$ (a::′*a*::{*ordered-field*,*division-by-zero*}))
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
  (*inverse* $a \leq 0$) = ($a \leq$ (0::′*a*::{*ordered-field*,*division-by-zero*}))
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *ordered-field-no-lb*: $\forall$ x. $\exists$ y. $y <$ (x::′*a*::*ordered-field*)
proof
  fix x::′a
  have *m1*: $-$ (1::′a) < 0 **by** *simp*
  from *add-strict-right-mono*[*OF m1*, **where** c=x]
  have ($-$ 1) $+ x < x$ **by** *simp*
  thus $\exists$ y. $y < x$ **by** *blast*
qed

lemma *ordered-field-no-ub*: $\forall$ x. $\exists$ y. $y >$ (x::′*a*::*ordered-field*)
proof
  fix x::′a
  have *m1*: (1::′a) > 0 **by** *simp*
  from *add-strict-right-mono*[*OF m1*, **where** c=x]
  have $1 + x > x$ **by** *simp*
  thus $\exists$ y. $y > x$ **by** *blast*
qed

16.7 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* $a \leq$ *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*,*noatp*]:
 $[|0 < a;\ 0 < b|] ==> (inverse\ a < inverse\ b) = (b < (a::'a::ordered\text{-}field))$
by (*blast intro*: *less-imp-inverse-less dest*: *inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
 $[|a \leq b;\ 0 < a|] ==> inverse\ b \leq inverse\ (a::'a::ordered\text{-}field)$
by (*force simp add*: *order-le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le* [*simp*,*noatp*]:
 $[|0 < a;\ 0 < b|] ==> (inverse\ a \leq inverse\ b) = (b \leq (a::'a::ordered\text{-}field))$
by (*blast intro*: *le-imp-inverse-le dest*: *inverse-le-imp-le*)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
 $[|inverse\ a \leq inverse\ b;\ b < 0|] ==> b \leq (a::'a::ordered\text{-}field)$
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*:
 $[|a < b;\ b < 0|] ==> inverse\ b < inverse\ (a::'a::ordered\text{-}field)$
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*:
 $[|inverse\ a < inverse\ b;\ b < 0|] ==> b < (a::'a::ordered\text{-}field)$
apply (*rule classical*)
apply (*subgoal-tac* $a < 0$)
 prefer *2*
 apply (*force simp add*: *linorder-not-less intro*: *order-le-less-trans*)
apply (*insert inverse-less-imp-less* [*of* $-b$ $-a$])
apply (*simp add*: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *inverse-less-iff-less-neg* [*simp*,*noatp*]:
 $[|a < 0;\ b < 0|] ==> (inverse\ a < inverse\ b) = (b < (a::'a::ordered\text{-}field))$
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*:
  $[|a \leq b;\ b < 0|] ==> inverse\ b \leq inverse\ (a::'a::ordered\text{-}field)$
by (*force simp add*: *order-le-less less-imp-inverse-less-neg*)

lemma *inverse-le-iff-le-neg* [*simp*,*noatp*]:
 $[|a < 0;\ b < 0|] ==> (inverse\ a \leq inverse\ b) = (b \leq (a::'a::ordered\text{-}field))$
by (*blast intro*: *le-imp-inverse-le-neg dest*: *inverse-le-imp-le-neg*)

16.8 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*: ~ $(0 < x)$
  have ~ $(1 < inverse\ x)$
  proof
    assume $1 < inverse\ x$
    also with *notless* **have** ... ≤ 0 **by** (*simp add*: *linorder-not-less*)
    also have ... < 1 **by** (*rule zero-less-one*)
    finally show *False* **by** *auto*
  qed
  with *notless* **show** *?thesis* **by** *simp*
qed

lemma *inverse-eq-1-iff* [*simp*]:
  $(inverse\ x = 1) = (x = (1::'a::\{field,division\text{-}by\text{-}zero\}))$
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*:
  $(inverse\ x < 1) = (x \leq 0\ |\ 1 < (x::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
by (*simp add*: *linorder-not-le* [*symmetric*] *one-le-inverse-iff*)

lemma *inverse-le-1-iff*:
  $(inverse\ x \leq 1) = (x \leq 0\ |\ 1 \leq (x::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
by (*simp add*: *linorder-not-less* [*symmetric*] *one-less-inverse-iff*)

16.9 Simplification of Inequalities Involving Literal Divisors

```
lemma pos-le-divide-eq: 0 < (c::'a::ordered-field) ==> (a ≤ b/c) = (a*c ≤ b)
proof -
  assume less: 0<c
  hence (a ≤ b/c) = (a*c ≤ (b/c)*c)
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
  also have ... = (a*c ≤ b)
    by (simp add: order-less-imp-not-eq2 [OF less] divide-inverse mult-assoc)
  finally show ?thesis .
qed

lemma neg-le-divide-eq: c < (0::'a::ordered-field) ==> (a ≤ b/c) = (b ≤ a*c)
proof -
  assume less: c<0
  hence (a ≤ b/c) = ((b/c)*c ≤ a*c)
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
  also have ... = (b ≤ 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 ≤ 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 (cases c=0, simp)
apply (force simp add: pos-le-divide-eq neg-le-divide-eq linorder-neq-iff)
done

lemma pos-divide-le-eq: 0 < (c::'a::ordered-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-le-cancel-right 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-le-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-le-cancel-right 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-le-eq*:
  $(b/c \le a) =$
   $(\mathit{if}\ 0 < c\ \mathit{then}\ b \le a{*}c$
      $\mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ a{*}c \le b$
      $\mathit{else}\ 0 \le (a::'a::\{\mathit{ordered\text{-}field},\mathit{division\text{-}by\text{-}zero}\}))$
apply (*cases c=0*, *simp*)
apply (*force simp add*: *pos-divide-le-eq neg-divide-le-eq linorder-neq-iff*)
done

lemma *pos-less-divide-eq*:
  $0 < (c::'a::\mathit{ordered\text{-}field}) ==> (a < b/c) = (a{*}c < b)$
proof −
  assume *less*: $0<c$
  hence $(a < b/c) = (a{*}c < (b/c){*}c)$
    by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
  also have ... $= (a{*}c < b)$
    by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
  finally show *?thesis* **.**
qed

lemma *neg-less-divide-eq*:
 $c < (0::'a::\mathit{ordered\text{-}field}) ==> (a < b/c) = (b < a{*}c)$
proof −
  assume *less*: $c<0$
  hence $(a < b/c) = ((b/c){*}c < a{*}c)$
    by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
  also have ... $= (b < a{*}c)$
    by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
  finally show *?thesis* **.**
qed

lemma *less-divide-eq*:
  $(a < b/c) =$
   $(\mathit{if}\ 0 < c\ \mathit{then}\ a{*}c < b$
      $\mathit{else}\ \mathit{if}\ c < 0\ \mathit{then}\ b < a{*}c$
      $\mathit{else}\ \ a < (0::'a::\{\mathit{ordered\text{-}field},\mathit{division\text{-}by\text{-}zero}\}))$
apply (*cases c=0*, *simp*)
apply (*force simp add*: *pos-less-divide-eq neg-less-divide-eq linorder-neq-iff*)
done

lemma *pos-divide-less-eq*:
  $0 < (c::'a::\mathit{ordered\text{-}field}) ==> (b/c < a) = (b < a{*}c)$
proof −
  assume *less*: $0<c$
  hence $(b/c < a) = ((b/c){*}c < a{*}c)$
    by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
  also have ... $= (b < a{*}c)$
    by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)

```
  finally show ?thesis .
qed

lemma neg-divide-less-eq:
 c < (0::'a::ordered-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 (cases c=0, simp)
apply (force simp add: pos-divide-less-eq neg-divide-less-eq linorder-neq-iff)
done
```

16.10 Field simplification

Lemmas *field-simps* multiply with denominators in in(equations) if they can be proved to be non-zero (for equations) or positive/negative (for inequations).

```
lemmas field-simps = field-eq-simps

  pos-divide-less-eq neg-divide-less-eq
  pos-less-divide-eq neg-less-divide-eq
  pos-divide-le-eq neg-divide-le-eq
  pos-le-divide-eq neg-le-divide-eq
```

Lemmas *sign-simps* is a first attempt to automate proofs of positivity/negativity needed for *field-simps*. Have not added *sign-simps* to *field-simps* because the former can lead to case explosions.

```
lemmas sign-simps = group-simps
  zero-less-mult-iff  mult-less-0-iff
```

16.11 Division and Signs

```
lemma zero-less-divide-iff:
     ((0::'a::{ordered-field,division-by-zero}) < a/b) = (0 < a & 0 < b | a < 0 &
b < 0)
by (simp add: divide-inverse zero-less-mult-iff)
```

lemma *divide-less-0-iff*:
$(a/b < (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})) =$
$(0 < a \ \& \ b < 0 \ | \ a < 0 \ \& \ 0 < b)$
by (*simp add*: *divide-inverse mult-less-0-iff*)

lemma *zero-le-divide-iff*:
$((0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}) \leq a/b) =$
$(0 \leq a \ \& \ 0 \leq b \ | \ a \leq 0 \ \& \ b \leq 0)$
by (*simp add*: *divide-inverse zero-le-mult-iff*)

lemma *divide-le-0-iff*:
$(a/b \leq (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})) =$
$(0 \leq a \ \& \ b \leq 0 \ | \ a \leq 0 \ \& \ 0 \leq b)$
by (*simp add*: *divide-inverse mult-le-0-iff*)

lemma *divide-eq-0-iff* [*simp*,*noatp*]:
$(a/b = 0) = (a=0 \ | \ b=(0::'a::\{field,division\text{-}by\text{-}zero\}))$
by (*simp add*: *divide-inverse*)

lemma *divide-pos-pos*:
$0 < (x::'a::ordered\text{-}field) ==> 0 < y ==> 0 < x \ / \ y$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-pos*:
$0 <= (x::'a::ordered\text{-}field) ==> 0 < y ==> 0 <= x \ / \ y$
by(*simp add*:*field-simps*)

lemma *divide-neg-pos*:
$(x::'a::ordered\text{-}field) < 0 ==> 0 < y ==> x \ / \ y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-pos*:
$(x::'a::ordered\text{-}field) <= 0 ==> 0 < y ==> x \ / \ y <= 0$
by(*simp add*:*field-simps*)

lemma *divide-pos-neg*:
$0 < (x::'a::ordered\text{-}field) ==> y < 0 ==> x \ / \ y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-neg*:
$0 <= (x::'a::ordered\text{-}field) ==> y < 0 ==> x \ / \ y <= 0$
by(*simp add*:*field-simps*)

lemma *divide-neg-neg*:
$(x::'a::ordered\text{-}field) < 0 ==> y < 0 ==> 0 < x \ / \ y$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-neg*:

$(x::'a::ordered\text{-}field) <= 0 ==> y < 0 ==> 0 <= x / y$
by(*simp add:field-simps*)

16.12 Cancellation Laws for Division

lemma *divide-cancel-right* [*simp*,*noatp*]:
$(a/c = b/c) = (c = 0 \mid a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)
done

lemma *divide-cancel-left* [*simp*,*noatp*]:
$(c/a = c/b) = (c = 0 \mid a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)
done

16.13 Division and the Number One

Simplify expressions equated with 1

lemma *divide-eq-1-iff* [*simp*,*noatp*]:
$(a/b = 1) = (b \neq 0 \ \& \ a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
apply (*cases b=0*, *simp*)
apply (*simp add*: *right-inverse-eq*)
done

lemma *one-eq-divide-iff* [*simp*,*noatp*]:
$(1 = a/b) = (b \neq 0 \ \& \ a = (b::'a::\{field,division\text{-}by\text{-}zero\}))$
by (*simp add*: *eq-commute* [*of 1*])

lemma *zero-eq-1-divide-iff* [*simp*,*noatp*]:
$((0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}) = 1/a) = (a = 0)$
apply (*cases a=0*, *simp*)
apply (*auto simp add*: *nonzero-eq-divide-eq*)
done

lemma *one-divide-eq-0-iff* [*simp*,*noatp*]:
$(1/a = (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})) = (a = 0)$
apply (*cases a=0*, *simp*)
apply (*insert zero-neq-one* [*THEN not-sym*])
apply (*auto simp add*: *nonzero-divide-eq-eq*)
done

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemmas *zero-less-divide-1-iff* = *zero-less-divide-iff* [*of 1*, *simplified*]
lemmas *divide-less-0-1-iff* = *divide-less-0-iff* [*of 1*, *simplified*]
lemmas *zero-le-divide-1-iff* = *zero-le-divide-iff* [*of 1*, *simplified*]
lemmas *divide-le-0-1-iff* = *divide-le-0-iff* [*of 1*, *simplified*]

declare *zero-less-divide-1-iff* [*simp*]
declare *divide-less-0-1-iff* [*simp*,*noatp*]
declare *zero-le-divide-1-iff* [*simp*]
declare *divide-le-0-1-iff* [*simp*,*noatp*]

16.14 Ordering Rules for Division

lemma *divide-strict-right-mono*:
$[|a < b;\ 0 < c|] ==> a\ /\ c < b\ /\ (c::'a::\mathit{ordered\text{-}field})$
by (*simp add*: *order-less-imp-not-eq2 divide-inverse mult-strict-right-mono positive-imp-inverse-positive*)

lemma *divide-right-mono*:
$[|a \le b;\ 0 \le c|] ==> a/c \le b/(c::'a::\{\mathit{ordered\text{-}field},\mathit{division\text{-}by\text{-}zero}\})$
by (*force simp add*: *divide-strict-right-mono order-le-less*)

lemma *divide-right-mono-neg*: $(a::'a::\{\mathit{division\text{-}by\text{-}zero},\mathit{ordered\text{-}field}\}) <= b$
$==> c <= 0 ==> b\ /\ c <= a\ /\ c$
apply (*drule divide-right-mono* [*of* - - $-$ *c*])
apply *auto*
done

lemma *divide-strict-right-mono-neg*:
$[|b < a;\ c < 0|] ==> a\ /\ c < b\ /\ (c::'a::\mathit{ordered\text{-}field})$
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*:
$[|b < a;\ 0 < c;\ 0 < a*b|] ==> c\ /\ a < c\ /\ (b::'a::\mathit{ordered\text{-}field})$
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono*)

lemma *divide-left-mono*:
$[|b \le a;\ 0 \le c;\ 0 < a*b|] ==> c\ /\ a \le c\ /\ (b::'a::\mathit{ordered\text{-}field})$
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-right-mono*)

lemma *divide-left-mono-neg*: $(a::'a::\{\mathit{division\text{-}by\text{-}zero},\mathit{ordered\text{-}field}\}) <= b$
$==> c <= 0 ==> 0 < a * b ==> c\ /\ a <= c\ /\ b$
apply (*drule divide-left-mono* [*of* - - $-$ *c*])
apply (*auto simp add*: *mult-commute*)
done

lemma *divide-strict-left-mono-neg*:
$[|a < b;\ c < 0;\ 0 < a*b|] ==> c\ /\ a < c\ /\ (b::'a::\mathit{ordered\text{-}field})$
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono-neg*)

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*noatp*]:

fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $(1 \le b\ /\ a) = ((0 < a\ \&\ a \le b) \mid (a < 0\ \&\ b \le a))$
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1* [*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $(b\ /\ a \le 1) = ((0 < a\ \&\ b \le a) \mid (a < 0\ \&\ a \le b) \mid a{=}0)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1* [*noatp*]:
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* [*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $(b\ /\ a < 1) = ((0 < a\ \&\ b < a) \mid (a < 0\ \&\ a < b) \mid a{=}0)$
by (*auto simp add*: *divide-less-eq*)

16.15 Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (1 \le b/a) = (a \le b)$
by (*auto simp add*: *le-divide-eq*)

lemma *le-divide-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (1 \le b/a) = (b \le a)$
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (b/a \le 1) = (b \le a)$
by (*auto simp add*: *divide-le-eq*)

lemma *divide-le-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (b/a \le 1) = (a \le b)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (1 < b/a) = (a < b)$
by (*auto simp add*: *less-divide-eq*)

lemma *less-divide-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field,division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (1 < b/a) = (b < a)$

by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (b/a < 1) = (b < a)$
by (*auto simp add*: *divide-less-eq*)

lemma *divide-less-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow b/a < 1 <-> a < b$
by (*auto simp add*: *divide-less-eq*)

lemma *eq-divide-eq-1* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(1 = b/a) = ((a \neq 0 \ \& \ a = b))$
by (*auto simp add*: *eq-divide-eq*)

lemma *divide-eq-eq-1* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(b/a = 1) = ((a \neq 0 \ \& \ a = b))$
by (*auto simp add*: *divide-eq-eq*)

16.16 Reasoning about inequalities with division

lemma *mult-right-le-one-le*: $0 <= (x::'a::ordered\text{-}idom) ==> 0 <= y ==> y <= 1$
$==> x * y <= x$
by (*auto simp add*: *mult-compare-simps*)

lemma *mult-left-le-one-le*: $0 <= (x::'a::ordered\text{-}idom) ==> 0 <= y ==> y <= 1$
$==> y * x <= x$
by (*auto simp add*: *mult-compare-simps*)

lemma *mult-imp-div-pos-le*: $0 < (y::'a::ordered\text{-}field) ==> x <= z * y ==>$
$x \ / \ y <= z$
by (*subst pos-divide-le-eq*, *assumption+*)

lemma *mult-imp-le-div-pos*: $0 < (y::'a::ordered\text{-}field) ==> z * y <= x ==>$
$z <= x \ / \ y$
by(*simp add*:*field-simps*)

lemma *mult-imp-div-pos-less*: $0 < (y::'a::ordered\text{-}field) ==> x < z * y ==>$
$x \ / \ y < z$
by(*simp add*:*field-simps*)

lemma *mult-imp-less-div-pos*: $0 < (y::'a::ordered\text{-}field) ==> z * y < x ==>$
$z < x \ / \ y$
by(*simp add*:*field-simps*)

lemma *frac-le*: (*0*::′*a*::*ordered-field*) <= *x* ==>
  x <= *y* ==> *0* < *w* ==> *w* <= *z* ==> *x* / *z* <= *y* / *w*
 apply (*rule mult-imp-div-pos-le*)
 apply *simp*
 apply (*subst times-divide-eq-left*)
 apply (*rule mult-imp-le-div-pos*, *assumption*)
 apply (*rule mult-mono*)
 apply *simp-all*
done

lemma *frac-less*: (*0*::′*a*::*ordered-field*) <= *x* ==>
  x < *y* ==> *0* < *w* ==> *w* <= *z* ==> *x* / *z* < *y* / *w*
 apply (*rule mult-imp-div-pos-less*)
 apply *simp*
 apply (*subst times-divide-eq-left*)
 apply (*rule mult-imp-less-div-pos*, *assumption*)
 apply (*erule mult-less-le-imp-less*)
 apply *simp-all*
done

lemma *frac-less2*: (*0*::′*a*::*ordered-field*) < *x* ==>
  x <= *y* ==> *0* < *w* ==> *w* < *z* ==> *x* / *z* < *y* / *w*
 apply (*rule mult-imp-div-pos-less*)
 apply *simp-all*
 apply (*subst times-divide-eq-left*)
 apply (*rule mult-imp-less-div-pos*, *assumption*)
 apply (*erule mult-le-less-imp-less*)
 apply *simp-all*
done

It's not obvious whether these should be simprules or not. Their effect is to gather terms into one big fraction, like a*b*c / x*y*z. The rationale for that is unclear, but many proofs seem to need them.

declare *times-divide-eq* [*simp*]

16.17 Ordered Fields are Dense

context *ordered-semidom*
begin

lemma *less-add-one*: *a* < *a* + *1*
proof −
 have *a* + *0* < *a* + *1*
  by (*blast intro*: *zero-less-one add-strict-left-mono*)
 thus *?thesis* **by** *simp*
qed

lemma *zero-less-two*: *0* < *1* + *1*

by (*blast intro*: *less-trans zero-less-one less-add-one*)

end

lemma *less-half-sum*: $a < b ==> a < (a+b) \ / \ (1+1::'a::ordered\text{-}field)$
by (*simp add*: *field-simps zero-less-two*)

lemma *gt-half-sum*: $a < b ==> (a+b)/(1+1::'a::ordered\text{-}field) < b$
by (*simp add*: *field-simps zero-less-two*)

instance *ordered-field* < *dense-linear-order*
proof
  fix $x\ y :: 'a$
  have $x < x + 1$ **by** *simp*
  then show $\exists y.\ x < y$ **..**
  have $x - 1 < x$ **by** *simp*
  then show $\exists y.\ y < x$ **..**
  show $x < y \Longrightarrow \exists z>x.\ z < y$ **by** (*blast intro*!: *less-half-sum gt-half-sum*)
qed

16.18 Absolute Value

context *ordered-idom*
begin

lemma *mult-sgn-abs*: $sgn\ x * abs\ x = x$
  unfolding *abs-if sgn-if* **by** *auto*

end

lemma *abs-one* [*simp*]: $abs\ 1 = (1::'a::ordered\text{-}idom)$
  by (*simp add*: *abs-if zero-less-one* [*THEN order-less-not-sym*])

class *pordered-ring-abs* = *pordered-ring* + *pordered-ab-group-add-abs* +
  assumes *abs-eq-mult*:
    $(0 \le a \vee a \le 0) \wedge (0 \le b \vee b \le 0) \Longrightarrow |a * b| = |a| * |b|$

class *lordered-ring* = *pordered-ring* + *lordered-ab-group-add-abs*
begin

subclass *lordered-ab-group-add-meet* **by** *unfold-locales*
subclass *lordered-ab-group-add-join* **by** *unfold-locales*

end

lemma *abs-le-mult*: $abs\ (a * b) \le (abs\ a) * (abs\ (b::'a::lordered\text{-}ring))$
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-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: ring-simps)
      done
  }
  note b = this[OF refl[of a] refl[of b]]
  note addm = add-mono[of 0::'a - 0::'a, simplified]
  note addm2 = add-mono[of - 0::'a - 0::'a, simplified]
  have xy: - ?x <= ?y
    apply (simp)
    apply (rule-tac y=0::'a in order-trans)
    apply (rule addm2)
    apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
    apply (rule addm)
    apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
    done
  have yx: ?y <= ?x
    apply (simp add:diff-def)
    apply (rule-tac y=0 in order-trans)
    apply (rule addm2, (simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)+)
    apply (rule addm, (simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)+)
    done
  have i1: a*b <= abs a * abs b by (simp only: a b yx)
  have i2: - (abs a * abs b) <= a*b by (simp only: a b xy)
  show ?thesis
    apply (rule abs-leI)
    apply (simp add: i1)
    apply (simp add: i2[simplified minus-le-iff])
    done
qed

instance lordered-ring ⊆ pordered-ring-abs
proof
  fix a b :: 'a:: lordered-ring
  assume (0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0)
  show abs (a*b) = abs a * abs b
proof -
  have s: (0 <= a*b) | (a*b <= 0)
    apply (auto)
    apply (rule-tac split-mult-pos-le)
    apply (rule-tac contrapos-np[of a*b <= 0])
    apply (simp)
```

```
    apply (rule-tac split-mult-neg-le)
    apply (insert prems)
    apply (blast)
    done
  have mulprts: a * b = (pprt a + nprt a) * (pprt b + nprt b)
    by (simp add: prts[symmetric])
  show ?thesis
  proof cases
    assume 0 <= a * b
    then show ?thesis
      apply (simp-all add: mulprts abs-prts)
      apply (insert prems)
      apply (auto simp add:
        ring-simps
        iffD1[OF zero-le-iff-zero-nprt] iffD1[OF le-zero-iff-zero-pprt]
        iffD1[OF le-zero-iff-pprt-id] iffD1[OF zero-le-iff-nprt-id])
        apply(drule (1) mult-nonneg-nonpos[of a b], simp)
        apply(drule (1) mult-nonneg-nonpos2[of b a], simp)
      done
  next
    assume ~(0 <= a*b)
    with s have a*b <= 0 by simp
    then show ?thesis
      apply (simp-all add: mulprts abs-prts)
      apply (insert prems)
      apply (auto simp add: ring-simps)
      apply(drule (1) mult-nonneg-nonneg[of a b],simp)
      apply(drule (1) mult-nonpos-nonpos[of a b],simp)
      done
  qed
qed
qed

instance ordered-idom ⊆ pordered-ring-abs
by default (auto simp add: abs-if not-less
  equal-neg-zero neg-equal-zero mult-less-0-iff)

lemma abs-mult: abs (a * b) = abs a * abs (b::'a::ordered-idom)
  by (simp add: abs-eq-mult linorder-linear)

lemma abs-mult-self: abs a * abs a = a * (a::'a::ordered-idom)
  by (simp add: abs-if)

lemma nonzero-abs-inverse:
    a ≠ 0 ==> abs (inverse (a::'a::ordered-field)) = inverse (abs a)
apply (auto simp add: linorder-neq-iff abs-if nonzero-inverse-minus-eq
                      negative-imp-inverse-negative)
apply (blast intro: positive-imp-inverse-positive elim: order-less-asym)
done
```

```
lemma abs-inverse [simp]:
     abs (inverse (a::'a::{ordered-field,division-by-zero})) =
      inverse (abs a)
apply (cases a=0, simp)
apply (simp add: nonzero-abs-inverse)
done

lemma nonzero-abs-divide:
     b ≠ 0 ==> abs (a / (b::'a::ordered-field)) = abs a / abs b
by (simp add: divide-inverse abs-mult nonzero-abs-inverse)

lemma abs-divide [simp]:
     abs (a / (b::'a::{ordered-field,division-by-zero})) = abs a / abs b
apply (cases b=0, simp)
apply (simp add: nonzero-abs-divide)
done

lemma abs-mult-less:
     [| abs a < c; abs b < d |] ==> abs a * abs b < c*(d::'a::ordered-idom)
proof −
  assume ac: abs a < c
  hence cpos: 0<c by (blast intro: order-le-less-trans abs-ge-zero)
  assume abs b < d
  thus ?thesis by (simp add: ac cpos mult-strict-mono)
qed

lemmas eq-minus-self-iff = equal-neg-zero

lemma less-minus-self-iff: (a < −a) = (a < (0::'a::ordered-idom))
  unfolding order-less-le less-eq-neg-nonpos equal-neg-zero ..

lemma abs-less-iff: (abs a < b) = (a < b & −a < (b::'a::ordered-idom))
apply (simp add: order-less-le abs-le-iff)
apply (auto simp add: abs-if neg-less-eq-nonneg less-eq-neg-nonpos)
done

lemma abs-mult-pos: (0::'a::ordered-idom) <= x ==>
    (abs y) * x = abs (y * x)
  apply (subst abs-mult)
  apply simp
done

lemma abs-div-pos: (0::'a::{division-by-zero,ordered-field}) < y ==>
    abs x / y = abs (x / y)
  apply (subst abs-divide)
  apply (simp add: order-less-imp-le)
done
```

16.19 Bounds of products via negative and positive Part

lemma *mult-le-prts*:
  assumes
  a1 <= (a::'a::lordered-ring)
  a <= a2
  b1 <= b
  b <= b2
  shows
  *a * b <= pprt a2 * pprt b2 + pprt a1 * nprt b2 + nprt a2 * pprt b1 + nprt a1 * nprt b1*
proof −
  have *a * b = (pprt a + nprt a) * (pprt b + nprt b)*
    apply *(subst prts[symmetric])+*
    apply *simp*
    done
  then have *a * b = pprt a * pprt b + pprt a * nprt b + nprt a * pprt b + nprt a * nprt b*
    by *(simp add: ring-simps)*
  moreover have *pprt a * pprt b <= pprt a2 * pprt b2*
    by *(simp-all add: prems mult-mono)*
  moreover have *pprt a * nprt b <= pprt a1 * nprt b2*
  proof −
    have *pprt a * nprt b <= pprt a * nprt b2*
      by *(simp add: mult-left-mono prems)*
    moreover have *pprt a * nprt b2 <= pprt a1 * nprt b2*
      by *(simp add: mult-right-mono-neg prems)*
    ultimately show *?thesis*
      by *simp*
  qed
  moreover have *nprt a * pprt b <= nprt a2 * pprt b1*
  proof −
    have *nprt a * pprt b <= nprt a2 * pprt b*
      by *(simp add: mult-right-mono prems)*
    moreover have *nprt a2 * pprt b <= nprt a2 * pprt b1*
      by *(simp add: mult-left-mono-neg prems)*
    ultimately show *?thesis*
      by *simp*
  qed
  moreover have *nprt a * nprt b <= nprt a1 * nprt b1*
  proof −
    have *nprt a * nprt b <= nprt a * nprt b1*
      by *(simp add: mult-left-mono-neg prems)*
    moreover have *nprt a * nprt b1 <= nprt a1 * nprt b1*
      by *(simp add: mult-right-mono-neg prems)*
    ultimately show *?thesis*
      by *simp*
  qed
  ultimately show *?thesis*
    by − *(rule add-mono | simp)+*

qed

lemma *mult-ge-prts*:
  assumes
  a1 <= (*a*::'*a*::*lordered-ring*)
  a <= *a2*
  b1 <= *b*
  b <= *b2*
  shows
  a ∗ *b* >= *nprt a1* ∗ *pprt b2* + *nprt a2* ∗ *nprt b2* + *pprt a1* ∗ *pprt b1* + *pprt a2* ∗ *nprt b1*
proof −
  from *prems* **have** *a1*:− *a2* <= −*a* **by** *auto*
  from *prems* **have** *a2*: −*a* <= −*a1* **by** *auto*
  from *mult-le-prts*[*of* −*a2* −*a* −*a1 b1 b b2*, *OF a1 a2 prems*(*3*) *prems*(*4*), *simplified nprt-neg pprt-neg*]
  have *le*: − (*a* ∗ *b*) <= − *nprt a1* ∗ *pprt b2* + − *nprt a2* ∗ *nprt b2* + − *pprt a1* ∗ *pprt b1* + − *pprt a2* ∗ *nprt b1* **by** *simp*
  then have −(− *nprt a1* ∗ *pprt b2* + − *nprt a2* ∗ *nprt b2* + − *pprt a1* ∗ *pprt b1* + − *pprt a2* ∗ *nprt b1*) <= *a* ∗ *b*
    by (*simp only*: *minus-le-iff*)
  then show *?thesis* **by** *simp*
qed

end

17 Nat: Natural numbers

theory *Nat*
imports *Wellfounded-Recursion Ring-and-Field*
uses
  ~~/src/Tools/rat.ML
  ~~/src/Provers/Arith/cancel-sums.ML
  (*arith-data.ML*)
  ~~/src/Provers/Arith/fast-lin-arith.ML
  (*Tools/lin-arith.ML*)
  (*Tools/function-package/size.ML*)
begin

17.1 Type *ind*

typedecl *ind*

axiomatization
  Zero-Rep :: *ind* **and**
  Suc-Rep :: *ind* => *ind*
where
  — the axiom of infinity in 2 parts

inj-Suc-Rep: *inj Suc-Rep* **and**
Suc-Rep-not-Zero-Rep: *Suc-Rep x* $\neq$ *Zero-Rep*

17.2 Type nat

Type definition

inductive-set *Nat* :: *ind set*
where
Zero-RepI: *Zero-Rep* : *Nat*
| *Suc-RepI*: *i* : *Nat* ==> *Suc-Rep i* : *Nat*

global

typedef (**open** *Nat*)
nat = *Nat*
proof
show *Zero-Rep* : *Nat* **by** (*rule Nat.Zero-RepI*)
qed

consts
Suc :: *nat* => *nat*

local

instance *nat* :: *zero*
Zero-nat-def: *0* == *Abs-Nat Zero-Rep* **..**
lemmas [*code func del*] = *Zero-nat-def*

defs
Suc-def: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat n*)))

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

lemma *Suc-not-Zero* [*iff*]: *Suc m* $\neq$ *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* $\neq$ *Suc m*
by (*rule not-sym*, *rule Suc-not-Zero not-sym*)

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-Suc-eq* [*iff*]: (*Suc m* = *Suc n*) = (*m* = *n*)
  by (*rule inj-Suc* [*THEN inj-eq*])

rep-datatype *nat*
  distinct *Suc-not-Zero Zero-not-Suc*
  inject *Suc-Suc-eq*
  induction *nat-induct*

declare *nat.induct* [*case-names 0 Suc*, *induct type*: *nat*]
declare *nat.exhaust* [*case-names 0 Suc*, *cases type*: *nat*]

lemmas *nat-rec-0* = *nat.recs*(*1*)
  and *nat-rec-Suc* = *nat.recs*(*2*)

lemmas *nat-case-0* = *nat.cases*(*1*)
  and *nat-case-Suc* = *nat.cases*(*2*)

Injectiveness and distinctness lemmas

lemma *Suc-neq-Zero*: *Suc m* = *0* ==> *R*
by (*rule notE*, *rule Suc-not-Zero*)

lemma *Zero-neq-Suc*: *0* = *Suc m* ==> *R*
by (*rule Suc-neq-Zero*, *erule sym*)

lemma *Suc-inject*: *Suc x* = *Suc y* ==> *x* = *y*
by (*rule inj-Suc* [*THEN injD*])

lemma *nat-not-singleton*: (∀ *x*. *x* = (*0*::*nat*)) = *False*
by *auto*

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

17.3 Arithmetic operators

instance *nat* :: {*one*, *plus*, *minus*, *times*}

One-nat-def [*simp*]: *1* == *Suc 0* **..**

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* | *Suc* k => k)

primrec
mult-0: *0* ∗ n = *0*
mult-Suc: *Suc* m ∗ n = n + (m ∗ n)

17.4 Orders on *nat*

definition
pred-nat :: (*nat* ∗ *nat*) *set* **where**
pred-nat = {(m, n). n = *Suc* m}

instance *nat* :: *ord*
less-def: m < n == (m, n) : *pred-nat*ˆ+
le-def: $m \leq$ (n::*nat*) == ~ (n < m) **..**

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

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-nat*ˆ+) = (m < n)
apply (*unfold less-def*)
apply (*rule refl*)
done

17.4.1 Introduction properties

lemma *less-trans*: i < j ==> j < k ==> 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* ==> *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

17.4.2 Elimination properties

lemma *less-not-sym*: *n* < *m* ==> ~ *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*: ~ *P* ==> *m* < *n* **shows** *P*
apply (*rule contrapos-np*)
apply (*rule less-not-sym*)
apply (*rule h1*)
apply (*erule h2*)
done

lemma *less-not-refl*: ~ *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* ==> *R*
by (*rule notE*, *rule less-not-refl*)

lemma *less-not-refl2*: *n* < *m* ==> *m* ≠ (*n*::*nat*) **by** *blast*

lemma *less-not-refl3*: (*s*::*nat*) < *t* ==> *s* ≠ *t*
by (*rule not-sym*, *rule less-not-refl2*)

lemma *lessE*:
assumes *major*: *i* < *k*
and *p1*: *k* = *Suc i* ==> *P* **and** *p2*: !!*j*. *i* < *j* ==> *k* = *Suc j* ==> *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*]: ~ $n < (0::nat)$
by (*blast elim*: *lessE*)

lemma *less-zeroE*: $(n::nat) < 0$ ==> R
by (*rule notE*, *rule not-less0*)

lemma *less-SucE*: **assumes** *major*: $m < Suc\ n$
and *less*: $m < n$ ==> P **and** *eq*: $m = n$ ==> 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*,*noatp*]: $(n < (1::nat)) = (n = 0)$
by (*simp add*: *less-Suc-eq*)

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
by (*simp add*: *less-Suc-eq*)

lemma *Suc-mono*: $m < n$ ==> $Suc\ m < Suc\ n$
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*: $(m::nat) < n$ ==> $P\ n\ m$
and *eqCase*: $m = n$ ==> $P\ n\ m$ **and** *lessCase*: $n<m$ ==> $P\ n\ m$

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

17.4.3 Inductive (?) properties

lemma *Suc-lessI*: $m < n ==> Suc\ m \neq n ==> Suc\ m < n$
apply (*simp add*: *nat-neq-iff*)
apply (*blast elim*!: *less-irrefl less-SucE elim*: *less-asym*)
done

lemma *Suc-lessD*: $Suc\ m < n ==> m < n$
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 $n = m \vee n < m$

lemma *not-less-eq*: $(\sim m < n) = (n < Suc\ m)$

by (*induct m n rule*: *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 (*induct n rule*: *wf-induct* [*OF 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*]

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* ≤ *n*)
  unfolding *le-def* **by** (*rule not-less-eq* [*symmetric*])

lemma *le-imp-less-Suc*: *m* ≤ *n* ==> *m* < *Suc n*
by (*rule less-Suc-eq-le* [*THEN iffD2*])

lemma *le0* [*iff*]: (*0*::*nat*) ≤ *n*
  unfolding *le-def* **by** (*rule not-less0*)

lemma *Suc-n-not-le-n*: ~ *Suc n* ≤ *n*
by (*simp add*: *le-def*)

lemma *le-0-eq* [*iff*]: ((*i*::*nat*) ≤ *0*) = (*i* = *0*)
by (*induct i*) (*simp-all add*: *le-def*)

lemma *le-Suc-eq*: (*m* ≤ *Suc n*) = (*m* ≤ *n* | *m* = *Suc n*)
by (*simp del*: *less-Suc-eq-le add*: *less-Suc-eq-le* [*symmetric*] *less-Suc-eq*)

lemma *le-SucE*: *m* ≤ *Suc n* ==> (*m* ≤ *n* ==> *R*) ==> (*m* = *Suc n* ==> *R*) ==> *R*
by (*drule le-Suc-eq* [*THEN iffD1*], *iprover*+)

lemma *Suc-leI*: *m* < *n* ==> *Suc*(*m*) ≤ *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*) ≤ *n* ==> *m* ≤ *n*
by (*simp add*: *le-def less-Suc-eq*)

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: *Suc m* ≤ *n* ==> *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*: $m \leq n ==> m \leq Suc\ n$
by (*unfold le-def*) (*blast dest*: *Suc-lessD*)

lemma *less-imp-le*: $m < n ==> m \leq (n::nat)$
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*: $m \leq n ==> m < n \mid m = (n::nat)$
unfolding *le-def*
using *less-linear*
by (*blast elim*: *less-irrefl less-asym*)

lemma *less-or-eq-imp-le*: $m < n \mid m = n ==> m \leq (n::nat)$
unfolding *le-def*
using *less-linear*
by (*blast elim*!: *less-irrefl elim*: *less-asym*)

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*: $(m::nat) = n ==> m \leq n$
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*: $[\mid i \leq j;\ j < k \mid] ==> i < (k::nat)$
by (*blast dest*!: *le-imp-less-or-eq intro*: *less-trans*)

lemma *less-le-trans*: $[\mid i < j;\ j \leq k \mid] ==> i < (k::nat)$
by (*blast dest*!: *le-imp-less-or-eq intro*: *less-trans*)

lemma *le-trans*: $[\mid i \leq j;\ j \leq k \mid] ==> i \leq (k::nat)$
by (*blast dest*!: *le-imp-less-or-eq intro*: *less-or-eq-imp-le less-trans*)

lemma *le-anti-sym*: $[\mid m \leq n;\ n \leq m \mid] ==> m = (n::nat)$
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*: $(m::nat) \leq n$ ==> $m \neq n$ ==> $m < n$
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* **by** *blast*

Type *nat* is a wellfounded linear order

instance *nat* :: *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*: $m \leq n$ ==> $(n < Suc\ m) = (n = m)$
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*

These two rules ease the use of primitive recursion. NOTE USE OF ==

lemma *def-nat-rec-0*: (!!*n*. *f n* == *nat-rec c h n*) ==> *f 0* = *c*
by *simp*

lemma *def-nat-rec-Suc*: (!!*n*. *f n* == *nat-rec c h n*) ==> *f* (*Suc n*) = *h n* (*f n*)
by *simp*

lemma *not0-implies-Suc*: $n \neq 0 ==> \exists m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr0-implies-Suc*: $n > 0 ==> \exists m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr-implies-not0*: **fixes** $n :: nat$ **shows** $m<n ==> n \neq 0$
by (*cases n*) *simp-all*

lemma *neq0-conv*[*iff*]: **fixes** $n :: nat$ **shows** $(n \neq 0) = (0 < n)$
by (*cases n*) *simp-all*

This theorem is useful with *blast*

lemma *gr0I*: $((n::nat) = 0 ==> False) ==> 0 < n$
by (*rule neq0-conv*[*THEN iffD1*], *iprover*)

lemma *gr0-conv-Suc*: $(0 < n) = (\exists m.\ n = Suc\ m)$
by (*fast intro*: *not0-implies-Suc*)

lemma *not-gr0* [*iff*,*noatp*]: $!!n::nat.\ (\sim\ (0 < n)) = (n = 0)$
using *neq0-conv* **by** *blast*

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 (*cases 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

17.5 *LEAST* theorems for type *nat*

lemma *Least-Suc*:
$[|\ P\ n;\ \sim\ P\ 0\ |] ==> (LEAST\ n.\ P\ n) = Suc\ (LEAST\ m.\ P(Suc\ m))$
apply (*case-tac n*, *auto*)
apply (*frule LeastI*)
apply (*drule-tac* $P = \%x.\ P\ (Suc\ x)$ **in** *LeastI*)
apply (*subgoal-tac* $(LEAST\ x.\ P\ x) \leq Suc\ (LEAST\ x.\ P\ (Suc\ x))$)
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*:
$[|P\ n;\ Q\ m;\ {}^{\sim}P\ 0;\ !k.\ P\ (Suc\ k) = Q\ k|] ==> Least\ P = Suc\ (Least\ Q)$
by (*erule* (*1*) *Least-Suc* [*THEN ssubst*], *simp*)

17.6 *min* and *max*

lemma *mono-Suc*: *mono Suc*
by (*rule monoI*) *simp*

lemma *min-0L* [*simp*]: $min\ 0\ n = (0::nat)$
by (*rule min-leastL*) *simp*

lemma *min-0R* [*simp*]: $min\ n\ 0 = (0::nat)$
by (*rule min-leastR*) *simp*

lemma *min-Suc-Suc* [*simp*]: $min\ (Suc\ m)\ (Suc\ n) = Suc\ (min\ m\ n)$
by (*simp add*: *mono-Suc min-of-mono*)

lemma *min-Suc1*:
$min\ (Suc\ n)\ m = (case\ m\ of\ 0 => 0\ |\ Suc\ m' => Suc(min\ n\ m'))$
by (*simp split*: *nat.split*)

lemma *min-Suc2*:
$min\ m\ (Suc\ n) = (case\ m\ of\ 0 => 0\ |\ Suc\ m' => Suc(min\ m'\ n))$
by (*simp split*: *nat.split*)

lemma *max-0L* [*simp*]: $max\ 0\ n = (n::nat)$
by (*rule max-leastL*) *simp*

lemma *max-0R* [*simp*]: $max\ n\ 0 = (n::nat)$
by (*rule max-leastR*) *simp*

lemma *max-Suc-Suc* [*simp*]: $max\ (Suc\ m)\ (Suc\ n) = Suc(max\ m\ n)$
by (*simp add*: *mono-Suc max-of-mono*)

lemma *max-Suc1*:
$max\ (Suc\ n)\ m = (case\ m\ of\ 0 => Suc\ n\ |\ Suc\ m' => Suc(max\ n\ m'))$
by (*simp split*: *nat.split*)

lemma *max-Suc2*:
$max\ m\ (Suc\ n) = (case\ m\ of\ 0 => Suc\ n\ |\ Suc\ m' => Suc(max\ m'\ n))$
by (*simp split*: *nat.split*)

17.7 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*]: $n>0 \implies Suc\ (n - Suc\ 0) = n$
by (*simp split add*: *nat.split*)

declare *diff-Suc* [*simp del*, *code del*]

17.8 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 *add-Suc-shift* [*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*]: $(k + m \leq k + n) = (m\leq(n::nat))$
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*]: **fixes** m :: *nat* **shows** $(m + n = 0) = (m = 0 \ \& \ n = 0)$
by (*cases m*) *simp-all*

lemma *add-is-1*: $(m+n= Suc\ 0) = (m= Suc\ 0\ \&\ n=0 \mid m=0\ \&\ n= Suc\ 0)$
by (*cases 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*. $(m + n > 0) = (m>0 \mid n>0)$
by(*auto dest*:*gr0-implies-Suc*)

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

17.9 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 ≠ (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 | n=0)
  apply (induct m)
   apply (induct-tac [2] n)
    apply simp-all
  done
```

17.10 Monotonicity of Addition

strict, in 1st argument

```
lemma add-less-mono1: i < j ==> i + k < j + (k::nat)
by (induct k) simp-all
```

strict, in both arguments

```
lemma add-less-mono: [|i < j; k < l|] ==> i + k < j + (l::nat)
  apply (rule add-less-mono1 [THEN less-trans], assumption+)
  apply (induct j, simp-all)
  done
```

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

```
lemma less-imp-Suc-add: m < n ==> (∃ k. n = Suc (m + k))
  apply (induct n)
  apply (simp-all add: order-le-less)
  apply (blast elim!: less-SucE
             intro!: add-0-right [symmetric] add-Suc-right [symmetric])
  done
```

strict, in 1st argument; proof is by induction on $k > 0$

```
lemma mult-less-mono2: (i::nat) < j ==> 0<k ==> k * i < k * j
apply(auto simp: gr0-conv-Suc)
apply (induct-tac m)
```

apply (*simp-all add*: *add-less-mono*)
done

The naturals form an ordered *comm-semiring-1-cancel*

instance *nat* :: *ordered-semidom*
proof
  fix *i j k* :: *nat*
  show *0* < (*1*::*nat*) **by** *simp*
  show *i* ≤ *j* ==> *k* + *i* ≤ *k* + *j* **by** *simp*
  show *i* < *j* ==> *0* < *k* ==> *k* ∗ *i* < *k* ∗ *j* **by** (*simp add*: *mult-less-mono2*)
qed

lemma *nat-mult-1*: (*1*::*nat*) ∗ *n* = *n*
by *simp*

lemma *nat-mult-1-right*: *n* ∗ (*1*::*nat*) = *n*
by *simp*

17.11 Additional theorems about "less than"

An induction rule for estabilishing binary relations

lemma *less-Suc-induct*:
  assumes *less*: *i* < *j*
    and *step*: !!*i*. *P i* (*Suc i*)
    and *trans*: !!*i j k*. *P i j* ==> *P j k* ==> *P i k*
  shows *P i j*
proof −
  from *less* **obtain** *k* **where** *j*: *j* = *Suc*(*i*+*k*) **by** (*auto dest*: *less-imp-Suc-add*)
  have *P i* (*Suc* (*i* + *k*))
  proof (*induct k*)
    case *0*
    show *?case* **by** (*simp add*: *step*)
  next
    case (*Suc k*)
    thus *?case* **by** (*auto intro*: *assms*)
  qed
  thus *P i j* **by** (*simp add*: *j*)
qed

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. $P(n)$ is true for all $n \in \mathbb{N}$ if

- case "0": given $n = 0$ prove $P(n)$,
- case "smaller": given $n > 0$ and $\neg P(n)$ prove there exists a smaller integer m such that $\neg P(m)$.

lemma *infinite-descent0*[*case-names 0 smaller*]:

$\llbracket$ $P\ 0$; $!!n.\ n{>}0 \Longrightarrow \neg\ P\ n \Longrightarrow (\exists\, m{::}nat.\ m < n \wedge \neg P\ m)$ $\rrbracket \Longrightarrow P\ n$
by (*induct n rule*: *less-induct*, *case-tac n>0*, *auto*)

A compact version without explicit base case:

lemma *infinite-descent*:
$\llbracket$ $!!n{::}nat.\ \neg\ P\ n \Longrightarrow\ \exists\, m{<}n.\ \neg\ \ P\ m$ $\rrbracket \Longrightarrow\ \ P\ n$
by (*induct n rule*: *less-induct*, *auto*)

Infinite descent using a mapping to $\mathbb{N}$: $P(x)$ is true for all $x \in D$ if there exists a $V : D \to \mathbb{N}$ and

- case "0": given $V(x) = 0$ prove $P(x)$,
- case "smaller": given $V(x) > 0$ and $\neg P(x)$ prove there exists a $y \in D$ such that $V(y) < V(x)$ and $\neg P(y)$.

NB: the proof also shows how to use the previous lemma.

corollary *infinite-descent0-measure*[*case-names 0 smaller*]:
assumes *0*: $!!x.\ V\ x = (0{::}nat) \Longrightarrow P\ x$
and *1*: $!!x.\ V\ x > 0 \Longrightarrow \neg P\ x \Longrightarrow (\exists\, y.\ V\ y < V\ x \wedge \neg P\ y)$
shows $P\ x$
proof −
obtain n **where** $n = V\ x$ **by** *auto*
moreover have $!!x.\ V\ x = n \Longrightarrow P\ x$
proof (*induct n rule*: *infinite-descent0*)
case *0* — i.e. $V(x) = 0$
with *0* **show** $P\ x$ **by** *auto*
next — now $n > 0$ and $P(x)$ does not hold for some x with $V(x) = n$
case (*smaller n*)
then obtain x **where** *vxn*: $V\ x = n$ **and** $V\ x > 0 \wedge \neg\ P\ x$ **by** *auto*
with *1* **obtain** y **where** $V\ y < V\ x \wedge \neg\ P\ y$ **by** *auto*
with *vxn* **obtain** m **where** $m = V\ y \wedge m{<}n \wedge \neg\ P\ y$ **by** *auto*
thus *?case* **by** *auto*
qed
ultimately show $P\ x$ **by** *auto*
qed

Again, without explicit base case:

lemma *infinite-descent-measure*:
assumes $!!x.\ \neg\ P\ x \Longrightarrow \exists\, y.\ (V{::}'a{\Rightarrow}nat)\ y < V\ x \wedge \neg\ P\ y$ **shows** $P\ x$
proof −
from *assms* **obtain** n **where** $n = V\ x$ **by** *auto*
moreover have $!!x.\ V\ x = n \Longrightarrow P\ x$
proof (*induct n rule*: *infinite-descent*, *auto*)
fix x **assume** $\neg\ P\ x$
with *assms* **show** $\exists\, m < V\ x.\ \exists\, y.\ V\ y = m \wedge \neg\ P\ y$ **by** *auto*
qed
ultimately show $P\ x$ **by** *auto*

qed

A [clumsy] way of lifting < monotonicity to ≤ monotonicity

lemma *less-mono-imp-le-mono*:
$[\![\ !!i\ j{::}nat.\ i < j \Longrightarrow f\ i < f\ j;\ i \le j\ ]\!] \Longrightarrow f\ i \le ((f\ j){::}nat)$
by (*simp add*: *order-le-less*) (*blast*)

non-strict, in 1st argument

lemma *add-le-mono1*: $i \le j$ ==> $i + k \le j + (k{::}nat)$
by (*rule add-right-mono*)

non-strict, in both arguments

lemma *add-le-mono*: [| $i \le j$; $k \le l$ |] ==> $i + k \le j + (l{::}nat)$
by (*rule add-mono*)

lemma *le-add2*: $n \le ((m + n){::}nat)$
by (*insert add-right-mono* [*of 0 m n*], *simp*)

lemma *le-add1*: $n \le ((n + m){::}nat)$
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*: $(i{::}nat) \le j$ ==> $i \le j + m$
by (*rule le-trans*, *assumption*, *rule le-add1*)

lemma *trans-le-add2*: $(i{::}nat) \le j$ ==> $i \le m + j$
by (*rule le-trans*, *assumption*, *rule le-add2*)

lemma *trans-less-add1*: $(i{::}nat) < j$ ==> $i < j + m$
by (*rule less-le-trans*, *assumption*, *rule le-add1*)

lemma *trans-less-add2*: $(i{::}nat) < j$ ==> $i < m + j$
by (*rule less-le-trans*, *assumption*, *rule le-add2*)

lemma *add-lessD1*: $i + j < (k{::}nat)$ ==> $i < k$
apply (*rule le-less-trans* [*of - i+j*])
apply (*simp-all add*: *le-add1*)
done

lemma *not-add-less1* [*iff*]: ~ $(i + j < (i{::}nat))$
apply (*rule notI*)

apply (*erule add-lessD1* [*THEN less-irrefl*])
done

lemma *not-add-less2* [*iff*]: ~ ($j + i < (i::nat)$)
by (*simp add*: *add-commute not-add-less1*)

lemma *add-leD1*: $m + k \leq n$ ==> $m \leq (n::nat)$
apply (*rule order-trans* [*of* - *m+k*])
apply (*simp-all add*: *le-add1*)
done

lemma *add-leD2*: $m + k \leq n$ ==> $k \leq (n::nat)$
apply (*simp add*: *add-commute*)
apply (*erule add-leD1*)
done

lemma *add-leE*: $(m::nat) + k \leq n$ ==> ($m \leq n$ ==> $k \leq n$ ==> R) ==> R
by (*blast dest*: *add-leD1 add-leD2*)

needs !!*k* for *add-ac* to work

lemma *less-add-eq-less*: !!*k*::*nat*. $k < l$ ==> $m + l = k + n$ ==> $m < n$
by (*force simp del*: *add-Suc-right*
simp add: *less-iff-Suc-add add-Suc-right* [*symmetric*] *add-ac*)

17.12 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*: ~ $m < n$ ==> $n + (m - n) = (m::nat)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *le-add-diff-inverse* [*simp*]: $n \leq m$ ==> $n + (m - n) = (m::nat)$
by (*simp add*: *add-diff-inverse linorder-not-less*)

lemma *le-add-diff-inverse2* [*simp*]: $n \leq m$ ==> $(m - n) + n = (m::nat)$
by (*simp add*: *le-add-diff-inverse add-commute*)

17.13 More results about difference

lemma *Suc-diff-le*: $n \leq m$ ==> *Suc* $m - n$ = *Suc* $(m - n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-less-Suc*: $m - n <$ *Suc* m
apply (*induct m n rule*: *diff-induct*)
apply (*erule-tac* [*3*] *less-SucE*)
apply (*simp-all add*: *less-Suc-eq*)
done

lemma *diff-le-self* [*simp*]: $m - n \le (m::nat)$
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*]: $0 < n \Longrightarrow n - Suc\ i < n$
by (*cases n*) (*auto simp add*: *le-simps*)

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*: $k \le (j::nat) \Longrightarrow (i + j) - k = i + (j - k)$
by (*induct j k rule*: *diff-induct*) *simp-all*

lemma *diff-add-assoc2*: $k \le (j::nat) \Longrightarrow (j + i) - k = (j - k) + i$
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*: $i \le (j::nat) \Longrightarrow (j - i = k) = (j = k + i)$
by (*auto simp add*: *diff-add-inverse2*)

lemma *diff-is-0-eq* [*simp*]: $((m::nat) - n = 0) = (m \le n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-is-0-eq'* [*simp*]: $m \le n \Longrightarrow (m::nat) - n = 0$
by (*rule iffD2*, *rule diff-is-0-eq*)

lemma *zero-less-diff* [*simp*]: $(0 < n - (m::nat)) = (m < n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *less-imp-add-positive*:
  assumes $i < j$
  shows $\exists k::nat.\ 0 < k\ \&\ i + k = j$
proof
  from *assms* **show** $0 < j - i\ \&\ i + (j - i) = j$

by (*simp add*: *order-less-imp-le*)
qed

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

17.14 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \leq (j::nat) ==> i * k \leq j * k$
by (*simp add*: *mult-right-mono*)

lemma *mult-le-mono2*: $i \leq (j::nat) ==> k * i \leq k * j$
by (*simp add*: *mult-left-mono*)

$\leq$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq (j::nat) ==> k \leq l ==> i * k \leq j * l$
by (*simp add*: *mult-mono*)

lemma *mult-less-mono1*: $(i::nat) < j ==> 0 < k ==> i * k < j * k$
by (*simp add*: *mult-strict-right-mono*)

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $(0 < (m::nat) * n) = (0 < m \ \& \ 0 < n)$
apply (*induct m*)
apply *simp*
apply (*case-tac n*)
apply *simp-all*
done

lemma *one-le-mult-iff* [*simp*]: $(Suc\ 0 \leq m * n) = (1 \leq m \ \& \ 1 \leq n)$
apply (*induct m*)

```
  apply simp
 apply (case-tac n)
  apply simp-all
 done

lemma mult-eq-1-iff [simp]: (m * n = Suc 0) = (m = 1 & n = 1)
 apply (induct m)
  apply simp
 apply (induct n)
  apply auto
 done

lemma one-eq-mult-iff [simp,noatp]: (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) ≤ k * n) = (0 < k --> m ≤ n)
by (simp add: linorder-not-less [symmetric], auto)

lemma mult-le-cancel2 [simp]: ((m::nat) * k ≤ n * k) = (0 < k --> m ≤ n)
by (simp add: linorder-not-less [symmetric], auto)

lemma mult-cancel2 [simp]: (m * k = n * k) = (m = n | (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 | (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 ≤ Suc k * n) = (m ≤ 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 (*drule-tac* [2] *mult-less-mono2*)
apply (*auto*)
done

17.15 size of a datatype value

class *size* = *type* +
fixes *size* :: $'a \Rightarrow nat$

use *Tools/function-package/size.ML*

setup *Size.setup*

lemma *nat-size* [*simp*, *code func*]: *size* $(n{::}nat) = n$
by (*induct n*) *simp-all*

lemma *size-bool* [*code func*]:
size $(b{::}bool) = 0$ **by** (*cases b*) *auto*

declare **.size* [*noatp*]

17.16 Code generator setup

instance *nat* :: *eq* ..

lemma [*code func*]:
$(0{::}nat) = 0 \longleftrightarrow True$
$Suc\ n = Suc\ m \longleftrightarrow n = m$
$Suc\ n = 0 \longleftrightarrow False$
$0 = Suc\ m \longleftrightarrow False$
by *auto*

lemma [*code func*]:
$(0{::}nat) \leq m \longleftrightarrow True$
$Suc\ (n{::}nat) \leq m \longleftrightarrow n < m$
$(n{::}nat) < 0 \longleftrightarrow False$
$(n{::}nat) < Suc\ m \longleftrightarrow n \leq m$
using *Suc-le-eq less-Suc-eq-le* **by** *simp-all*

17.17 Embedding of the Naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

definition
of-nat-def: *of-nat* = *nat-rec* *0* (λ-. (*op* +) *1*)

lemma *of-nat-simps* [*simp*, *code*]:
shows *of-nat-0*: *of-nat* *0* = *0*
and *of-nat-Suc*: *of-nat* (*Suc* *m*) = *1* + *of-nat* *m*
unfolding *of-nat-def* **by** *simp-all*

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*
by (*induct* *m*) (*simp-all* *add*: *add-ac*)

lemma *of-nat-mult*: *of-nat* (*m* * *n*) = *of-nat* *m* * *of-nat* *n*
by (*induct* *m*) (*simp-all* *add*: *add-ac* *left-distrib*)

end

context *ordered-semidom*
begin

lemma *zero-le-imp-of-nat*: *0* ≤ *of-nat* *m*
apply (*induct* *m*, *simp-all*)
apply (*erule* *order-trans*)
apply (*rule* *ord-le-eq-trans* [*OF* - *add-commute*])
apply (*rule* *less-add-one* [*THEN* *less-imp-le*])
done

lemma *less-imp-of-nat-less*: *m* < *n* ⟹ *of-nat* *m* < *of-nat* *n*
apply (*induct* *m* *n* *rule*: *diff-induct*, *simp-all*)
apply (*insert* *add-less-le-mono* [*OF* *zero-less-one* *zero-le-imp-of-nat*], *force*)
done

lemma *of-nat-less-imp-less*: *of-nat* *m* < *of-nat* *n* ⟹ *m* < *n*
apply (*induct* *m* *n* *rule*: *diff-induct*, *simp-all*)
apply (*insert* *zero-le-imp-of-nat*)
apply (*force* *simp* *add*: *not-less* [*symmetric*])
done

lemma *of-nat-less-iff* [*simp*]: *of-nat* *m* < *of-nat* *n* ⟷ *m* < *n*
by (*blast* *intro*: *of-nat-less-imp-less* *less-imp-of-nat-less*)

Special cases where either operand is zero

lemma *of-nat-0-less-iff* [*simp*]: *0* < *of-nat* *n* ⟷ *0* < *n*
by (*rule* *of-nat-less-iff* [*of* *0*, *simplified*])

lemma *of-nat-less-0-iff* [*simp*]: ¬ *of-nat* *m* < *0*
by (*rule* *of-nat-less-iff* [*of* - *0*, *simplified*])

lemma *of-nat-le-iff* [*simp*]:
of-nat $m \leq$ *of-nat* $n \longleftrightarrow m \leq n$
by (*simp add*: *not-less* [*symmetric*] *linorder-not-less* [*symmetric*])

Special cases where either operand is zero

lemma *of-nat-0-le-iff* [*simp*]: $0 \leq$ *of-nat* n
by (*rule of-nat-le-iff* [*of 0*, *simplified*])

lemma *of-nat-le-0-iff* [*simp*, *noatp*]: *of-nat* $m \leq 0 \longleftrightarrow m = 0$
by (*rule of-nat-le-iff* [*of - 0*, *simplified*])

end

lemma *of-nat-id* [*simp*]: *of-nat* $n = n$
by (*induct n*) *auto*

lemma *of-nat-eq-id* [*simp*]: *of-nat* = *id*
by (*auto simp add*: *expand-fun-eq*)

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *semiring-char-0* = *semiring-1* +
assumes *of-nat-eq-iff* [*simp*]: *of-nat* $m =$ *of-nat* $n \longleftrightarrow m = n$

Every *ordered-semidom* has characteristic zero.

subclass (**in** *ordered-semidom*) *semiring-char-0*
by *unfold-locales* (*simp add*: *eq-iff order-eq-iff*)

context *semiring-char-0*
begin

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*, *noatp*]: $0 =$ *of-nat* $n \longleftrightarrow 0 = n$
by (*rule of-nat-eq-iff* [*of 0*, *simplified*])

lemma *of-nat-eq-0-iff* [*simp*, *noatp*]: *of-nat* $m = 0 \longleftrightarrow m = 0$
by (*rule of-nat-eq-iff* [*of - 0*, *simplified*])

lemma *inj-of-nat*: *inj of-nat*
by (*simp add*: *inj-on-def*)

end

17.18 Further Arithmetic Facts Concerning the Natural Numbers

lemma *subst-equals*:

assumes *1*: $t = s$ **and** *2*: $u = t$
shows $u = s$
using *2 1* **by** (*rule trans*)

use *arith-data.ML*
declaration ⟪ *K arith-data-setup* ⟫

use *Tools/lin-arith.ML*
declaration ⟪ *K LinArith.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*:
$P(a - b::nat) = ((a<b \dashrightarrow P\ 0)\ \&\ (ALL\ d.\ a = b + d \dashrightarrow P\ d))$
— elimination of $-$ on *nat*
by (*cases a<b rule*: *case-split*) (*auto simp add*: *diff-is-0-eq* [*THEN iffD2*])

context *ring-1*
begin

lemma *of-nat-diff*: $n \leq m \Longrightarrow of\text{-}nat\ (m - n) = of\text{-}nat\ m - of\text{-}nat\ n$
by (*simp del*: *of-nat-add*
add: *compare-rls of-nat-add* [*symmetric*] *split add*: *nat-diff-split*)

end

lemma *abs-of-nat* [*simp*]: $|of\text{-}nat\ n::'a::ordered\text{-}idom| = of\text{-}nat\ n$
unfolding *abs-if* **by** *auto*

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 m*) *auto*

lemma *le-cube*: $(m::nat) \leq m * (m * m)$
by (*induct 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 (*cases m*) *auto*

lemma *n-less-m-mult-n*: [| *Suc 0* < *n*; *Suc 0* < *m* |] ==> *n*<*m*∗*n*
by (*cases m*) *auto*

lemma *n-less-n-mult-m*: [| *Suc 0* < *n*; *Suc 0* < *m* |] ==> *n*<*n*∗*m*
by (*cases m*) *auto*

Specialized induction principles that work "backwards":

lemma *inc-induct*[*consumes 1*, *case-names base step*]:
  assumes *less*: *i* <= *j*
  assumes *base*: *P j*
  assumes *step*: !!*i*. [| *i* < *j*; *P* (*Suc i*) |] ==> *P i*
  shows *P i*
  using *less*
proof (*induct d*==*j* − *i arbitrary*: *i*)
  case (*0 i*)
  hence *i* = *j* **by** *simp*
  with *base* **show** *?case* **by** *simp*
next
  case (*Suc d i*)
  hence *i* < *j P* (*Suc i*)
    by *simp-all*
  thus *P i* **by** (*rule step*)
qed

lemma *strict-inc-induct*[*consumes 1*, *case-names base step*]:
  assumes *less*: *i* < *j*
  assumes *base*: !!*i*. *j* = *Suc i* ==> *P i*
  assumes *step*: !!*i*. [| *i* < *j*; *P* (*Suc i*) |] ==> *P i*
  shows *P i*
  using *less*
proof (*induct d*==*j* − *i* − *1 arbitrary*: *i*)
  case (*0 i*)
  with ‹*i* < *j*› **have** *j* = *Suc i* **by** *simp*
  with *base* **show** *?case* **by** *simp*
next
  case (*Suc d i*)
  hence *i* < *j P* (*Suc i*)
    by *simp-all*
  thus *P i* **by** (*rule step*)
qed

lemma *zero-induct-lemma*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P* (*k* − *i*)
  using *inc-induct*[*of k* − *i k P*, *simplified*] **by** *blast*

lemma *zero-induct*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P 0*

using *inc-induct*[*of 0 k P*] **by** *blast*

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \le j$ --> $i - (j - k) = i + (k::nat) - j$
by *arith*

lemma *diff-Suc-diff-eq1* [*simp*]: $k \le j$ ==> $m - Suc\ (j - k) = m + k - Suc\ j$
by *arith*

lemma *diff-Suc-diff-eq2* [*simp*]: $k \le j$ ==> $Suc\ (j - k) - m = Suc\ j - (k + m)$
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.

17.19 The Set of Natural Numbers

context *semiring-1*
begin

definition
Nats :: *'a set* **where**
Nats = *range of-nat*

notation (*xsymbols*)
Nats ($\mathbb{N}$)

end

context *semiring-1*
begin

lemma *of-nat-in-Nats* [*simp*]: *of-nat* $n \in \mathbb{N}$
by (*simp add*: *Nats-def*)

lemma *Nats-0* [*simp*]: $0 \in \mathbb{N}$
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-0* [*symmetric*])
done

lemma *Nats-1* [*simp*]: $1 \in \mathbb{N}$
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)

apply (*rule of-nat-1* [*symmetric*])
done

lemma *Nats-add* [*simp*]: $a \in \mathbb{N} \Longrightarrow b \in \mathbb{N} \Longrightarrow a + b \in \mathbb{N}$
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-add* [*symmetric*])
done

lemma *Nats-mult* [*simp*]: $a \in \mathbb{N} \Longrightarrow b \in \mathbb{N} \Longrightarrow a * b \in \mathbb{N}$
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-mult* [*symmetric*])
done

end

the lattice order on *nat*

instance *nat* :: *distrib-lattice*
  inf ≡ *min*
  sup ≡ *max*
  by *intro-classes*
    (*simp-all add*: *inf-nat-def sup-nat-def*)

17.20 legacy bindings

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*;
⟫

end

18 Power: Exponentiation

theory *Power*
imports *Nat*
begin

class *power* = *type* +
  fixes *power* :: $'a \Rightarrow nat \Rightarrow 'a$ (**infixr** ˆ *80*)

18.1 Powers for Arbitrary Monoids

class *recpower* = *monoid-mult* + *power* +
  assumes *power-0* [*simp*]: *a* ˆ *0* = *1*
  assumes *power-Suc*: *a* ˆ *Suc n* = *a* ∗ (*a* ˆ *n*)

lemma *power-0-Suc* [*simp*]: (*0*::'*a*::{*recpower*,*semiring-0*}) ˆ (*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*ˆ*n* = (*if n*=*0 then 1 else* (*0*::'*a*::{*recpower*,*semiring-0*}))
  by (*induct n*) *simp-all*

lemma *power-one* [*simp*]: *1*ˆ*n* = (*1*::'*a*::*recpower*)
  by (*induct n*) (*simp-all add*: *power-Suc*)

lemma *power-one-right* [*simp*]: (*a*::'*a*::*recpower*) ˆ *1* = *a*
  by (*simp add*: *power-Suc*)

lemma *power-commutes*: (*a*::'*a*::*recpower*) ˆ *n* ∗ *a* = *a* ∗ *a* ˆ *n*
  by (*induct n*) (*simp-all add*: *power-Suc mult-assoc*)

lemma *power-add*: (*a*::'*a*::*recpower*) ˆ (*m*+*n*) = (*a*ˆ*m*) ∗ (*a*ˆ*n*)
  by (*induct m*) (*simp-all add*: *power-Suc mult-ac*)

lemma *power-mult*: (*a*::'*a*::*recpower*) ˆ (*m*∗*n*) = (*a*ˆ*m*) ˆ *n*
  by (*induct n*) (*simp-all add*: *power-Suc power-add*)

lemma *power-mult-distrib*: ((*a*::'*a*::{*recpower*,*comm-monoid-mult*}) ∗ *b*) ˆ *n* = (*a*ˆ*n*) ∗ (*b*ˆ*n*)

by (*induct n*) (*simp-all add*: *power-Suc mult-ac*)

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 *one-less-power*:
$[\![1 < (a::'a::\{ordered\text{-}semidom,recpower\}); 0 < n]\!] \Longrightarrow 1 < a \hat{}\ n$
by (*cases n*, *simp-all add*: *power-gt1-lemma power-Suc*)

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*: $(1::'a::\{recpower,ordered\text{-}semidom\}) < a$
shows $!!n.\ a\hat{}m \leq a\hat{}n ==> m \leq n$
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^n = 0) = (a = (0::'a::{ring-1-no-zero-divisors,recpower}) & n>0)
apply (induct n)
```

apply (*auto simp add*: *power-Suc zero-neq-one* [*THEN not-sym*])
done

lemma *field-power-not-zero*:
$a \neq (0::'a::\{ring\text{-}1\text{-}no\text{-}zero\text{-}divisors,recpower\}) ==> a\hat{}n \neq 0$
by *force*

lemma *nonzero-power-inverse*:
fixes $a :: 'a::\{division\text{-}ring,recpower\}$
shows $a \neq 0 ==> inverse\ (a\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
apply (*induct n*)
apply (*auto simp add*: *power-Suc nonzero-inverse-mult-distrib power-commutes*)
done

Perhaps these should be simprules.

lemma *power-inverse*:
fixes $a :: 'a::\{division\text{-}ring,division\text{-}by\text{-}zero,recpower\}$
shows $inverse\ (a\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
apply (*cases* $a = 0$)
apply (*simp add*: *power-0-left*)
apply (*simp add*: *nonzero-power-inverse*)
done

lemma *power-one-over*: $1\ /\ (a::'a::\{field,division\text{-}by\text{-}zero,recpower\})\hat{}n =$
$(1\ /\ a)\hat{}n$
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,noatp*]:
$(0 < (abs\ a)\hat{}n) = (a \neq (0::'a::\{ordered\text{-}idom,recpower\})\ |\ n=0)$
proof (*induct n*)
case *0*

show *?case* **by** (*simp add*: *zero-less-one*)
next
case (*Suc n*)
show *?case* **by** (*auto simp add*: *prems power-Suc zero-less-mult-iff abs-zero*)
qed

lemma *zero-le-power-abs* [*simp*]:
$(0::'a::\{ordered\text{-}idom,recpower\}) \leq (abs\ a)\hat{}n$
by (*rule zero-le-power* [*OF abs-ge-zero*])

lemma *power-minus*: $(-a)\ \hat{}\ n = (-\ 1)\hat{}n * (a::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ n$
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*:
$[|(0::'a::\{ordered\text{-}semidom,recpower\}) < a;\ a < 1|]$
$==> a * a\hat{}n < a\hat{}n$
apply (*induct n*)
apply (*auto simp add*: *power-Suc mult-strict-left-mono*)
done

lemma *power-strict-decreasing*:
$[|n < N;\ 0 < a;\ a < (1::'a::\{ordered\text{-}semidom,recpower\})|]$
$==> a\hat{}N < a\hat{}n$
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* $a * a\hat{}m < 1 * a\hat{}n$, *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*:
$[|n \leq N;\ 0 \leq a;\ a \leq (1::'a::\{ordered\text{-}semidom,recpower\})|]$
$==> a\hat{}N \leq a\hat{}n$
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-Suc le-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac* $a * a\hat{}m \leq 1 * a\hat{}n$, *simp*)
apply (*rule mult-mono*)
apply (*auto simp add*: *zero-le-power zero-le-one*)
done

lemma *power-Suc-less-one*:
  $[\mid 0 < a;\ a < (1::'a::\{ordered\text{-}semidom,recpower\})\ \mid] ==> a\ \hat{}\ Suc\ n < 1$
apply (*insert power-strict-decreasing* [*of 0 Suc n a*], *simp*)
done

Proof again resembles that of *power-strict-decreasing*

lemma *power-increasing*:
  $[\mid n \leq N;\ (1::'a::\{ordered\text{-}semidom,recpower\}) \leq a\mid] ==> a\hat{}n \leq a\hat{}N$
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-Suc le-Suc-eq*)
apply (*rename-tac m*)
apply ($subgoal\text{-}tac\ 1 * a\hat{}n \leq a * a\hat{}m$, *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*:
  $(1::'a::\{ordered\text{-}semidom,recpower\}) < a ==> a\hat{}n < a * a\hat{}n$
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*:
  $[\mid n < N;\ (1::'a::\{ordered\text{-}semidom,recpower\}) < a\mid] ==> a\hat{}n < a\hat{}N$
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 ($subgoal\text{-}tac\ 1 * a\hat{}n < a * a\hat{}m$, *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*]:
  $1 < (b::'a::\{ordered\text{-}semidom,recpower\}) ==> (b\ \hat{}\ x \leq b\ \hat{}\ y) = (x \leq y)$
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 *ynonneg*: $(0::'a::\{ordered\text{-}semidom,recpower\}) \leq b$
shows $a \leq b$

proof (*rule ccontr*)
  assume $\sim$ $a \leq b$
  then have $b < a$ **by** (*simp only*: *linorder-not-le*)
  then have b ˆ *Suc n* $<$ a ˆ *Suc n*
    by (*simp only*: *prems power-strict-mono*)
  from *le* **and** *this* **show** *False*
    by (*simp add*: *linorder-not-less* [*symmetric*])
qed

lemma *power-less-imp-less-base*:
  fixes a b :: $'a$::{*ordered-semidom*,*recpower*}
  assumes *less*: a ˆ $n < b$ ˆ n
  assumes *nonneg*: $0 \leq b$
  shows $a < b$
proof (*rule contrapos-pp* [*OF less*])
  assume $\sim$ $a < b$
  hence $b \leq a$ **by** (*simp only*: *linorder-not-less*)
  hence b ˆ $n \leq a$ ˆ n **using** *nonneg* **by** (*rule power-mono*)
  thus $\sim$ a ˆ $n < b$ ˆ n **by** (*simp only*: *linorder-not-less*)
qed

lemma *power-inject-base*:
    [| a ˆ *Suc n* $=$ b ˆ *Suc n*; $0 \leq a$; $0 \leq b$ |]
    ==> $a = (b$::$'a$::{*ordered-semidom*,*recpower*})
by (*blast intro*: *power-le-imp-le-base order-antisym order-eq-refl sym*)

lemma *power-eq-imp-eq-base*:
  fixes a b :: $'a$::{*ordered-semidom*,*recpower*}
  shows $[\![a$ ˆ $n = b$ ˆ n; $0 \leq a$; $0 \leq b$; $0 < n]\!] \Longrightarrow a = b$
by (*cases n*, *simp-all*, *rule power-inject-base*)

18.2 Exponentiation for the Natural Numbers

instance *nat* :: *power* **..**

primrec (*power*)
  p ˆ $0 = 1$
  p ˆ (*Suc n*) $=$ (p::*nat*) $*$ (p ˆ n)

instance *nat* :: *recpower*
proof
  fix z n :: *nat*
  show zˆ$0 = 1$ **by** *simp*
  show zˆ(*Suc n*) $= z * (z$ˆ$n)$ **by** *simp*
qed

lemma *of-nat-power*:
  of-nat (m ˆ n) $=$ (*of-nat* m::$'a$::{*semiring-1*,*recpower*}) ˆ n
by (*induct n*, *simp-all add*: *power-Suc of-nat-mult*)

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 *nat-zero-less-power-iff* [*simp*]: $(x\hat{}n > 0) = (x > (0::nat) \mid n=0)$
by (*induct n*, *auto*)

Valid for the naturals, but what if $0<i<1$? Premises cannot be weakened: consider the case where $i = (0::'a)$, $m = (1::'a)$ and $n = (0::'a)$.

lemma *nat-power-less-imp-less*:
  assumes *nonneg*: $0 < (i::nat)$
  assumes *less*: $i\hat{}m < i\hat{}n$
  shows $m < n$
proof (*cases* $i = 1$)
  case *True* **with** *less power-one* [**where** $'a = nat$] **show** *?thesis* **by** *simp*
next
  case *False* **with** *nonneg* **have** $1 < i$ **by** *auto*
  from *power-strict-increasing-iff* [*OF this*] *less* **show** *?thesis* **..**
qed

lemma *power-diff*:
  assumes *nz*: $a \ \tilde{}= 0$
  shows $n <= m ==> (a::'a::\{recpower, field\})\ \hat{}\ (m-n) = (a\hat{}m) / (a\hat{}n)$
  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 = *thmpower-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 nat-power-less-imp-less = *thmnat-power-less-imp-less*;
val nat-zero-less-power-iff = *thmnat-zero-less-power-iff*;
⟫

end

19 Divides: The division operators div, mod and the divides relation "dvd"

theory *Divides*
imports *Power*
uses ~~/*src/Provers/Arith/cancel-div-mod.ML*
begin

class *div* = *times* +
  fixes *div* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *div* *70*)
  fixes *mod* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *mod* *70*)

instance *nat* :: *Divides.div*
  div-def: *m div n* == *wfrec* (*pred-nat*^+)
                              (%*f j*. *if j*<*n* | *n*=*0 then 0 else Suc* (*f* (*j*−*n*))) *m*
  mod-def: *m mod n* == *wfrec* (*pred-nat*^+)

(%f j. if j<n | n=0 then j else f (j−n)) m ..

definition (**in** *div*)
dvd :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** *dvd 50*)
where
[*code func del*]: *m dvd n* $\longleftrightarrow$ ($\exists$ *k*. *n* = *m* ∗ *k*)

class *dvd-mod* = *div* + *zero* + — for code generation
assumes *dvd-def-mod* [*code func*]: *x dvd y* $\longleftrightarrow$ *y mod x* = *0*

definition
quorem :: (*nat*∗*nat*) ∗ (*nat*∗*nat*) => *bool* **where**

quorem = (%((*a*,*b*), (*q*,*r*)).
a = *b*∗*q* + *r* &
(*if 0*<*b then 0*≤*r* & *r*<*b else b*<*r* & *r* ≤*0*))

19.1 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 (*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* (*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*)

19.2 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 (*cases n*=*0*)
apply *simp*
apply (*rule mod-eq* [*THEN wf-less-trans*])
apply (*simp add*: *cut-apply less-eq*)

done

lemma *le-mod-geq*: (*n*::*nat*) $\leq$ *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*
by (*induct m*) (*simp-all add*: *mod-geq*)

lemma *mod-self* [*simp*]: *n mod n* = (*0*::*nat*)
by (*cases n* = *0*) (*simp-all add*: *mod-geq*)

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*)
by (*induct k*) (*simp-all add*: *add-left-commute* [*of* - *n*])

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 (*cases n* = *0*, *simp*)
apply (*cases k* = *0*, *simp*)
apply (*induct m rule*: *nat-less-induct*)
apply (*subst mod-if*, *simp*)
apply (*simp add*: *mod-geq diff-mult-distrib*)
done

lemma *mod-mult-distrib2*: (*k*::*nat*) ∗ (*m mod n*) = (*k*∗*m*) *mod* (*k*∗*n*)
by (*simp add*: *mult-commute* [*of k*] *mod-mult-distrib*)

lemma *mod-mult-self-is-0* [*simp*]: (*m*∗*n*) *mod n* = (*0*::*nat*)
apply (*cases 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*)

19.3 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 (*cases n* = *0*, *simp*)
apply (*induct m rule*: *nat-less-induct*)
apply (*subst mod-if*)
apply (*simp-all 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

19.4 Simproc for Cancelling Div and Mod

lemma *div-mod-equality*: ((*m div n*)∗*n* + *m mod n*) + *k* = (*m*::*nat*) + *k*
by (*simp add*: *mod-div-equality*)

lemma *div-mod-equality2*: (*n*∗(*m div n*) + *m mod n*) + *k* = (*m*::*nat*) + *k*
by (*simp add*: *mod-div-equality2*)

ML
⟪
structure CancelDivModData =
struct

val div-name = @{*const-name Divides.div*};
val mod-name = @{*const-name Divides.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 [@{thm div-mod-equality}, @{thm div-mod-equality2}]

val trans = trans

val prove-eq-sums =
  let val simps = @{thm add-0} :: @{thm add-0-right} :: @{thms add-ac}
  in 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], K 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 (*rename-tac m*)
  apply (*case-tac* $m<n$, *simp*)

case $n \le m$

  apply (*simp add*: *mod-geq*)
  done

lemma *mod-le-divisor*[*simp*]: $0 < n \Longrightarrow m\ mod\ n \le (n::nat)$
  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*)

19.5 Proving facts about Quotient and Remainder

lemma *unique-quotient-lemma*:
  $[\!|$ $b*q' + r' \le b*q + r$; $x < b$; $r < b$ $|\!]$
  $==> q' \le (q::nat)$
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*: $b > 0 ==>$ *quorem* $((a, b), (a \textit{ div } b, a \textit{ mod } b))$
unfolding *quorem-def* **by** *simp*

lemma *quorem-div*: $[\!|$ *quorem*$((a,b),(q,r))$; $b > 0$ $|\!] ==> a \textit{ div } b = q$
by (*simp add*: *quorem-div-mod* [*THEN unique-quotient*])

lemma *quorem-mod*: $[\!|$ *quorem*$((a,b),(q,r))$; $b > 0$ $|\!] ==> a \textit{ mod } b = r$
by (*simp add*: *quorem-div-mod* [*THEN unique-remainder*])

lemma *div-0* [*simp*]: $0 \textit{ div } m = (0::nat)$
by (*cases* $m = 0$) *simp-all*

lemma *mod-0* [*simp*]: $0 \textit{ mod } m = (0::nat)$
by (*cases* $m = 0$) *simp-all*

lemma *quorem-mult1-eq*:
 $[\!|$ *quorem*$((b,c),(q,r))$; $c > 0$ $|\!]$
 $==>$ *quorem* $((a*b, c), (a*q + a*r \textit{ div } c, a*r \textit{ mod } c))$
by (*auto simp add*: *split-ifs mult-ac quorem-def add-mult-distrib2*)

lemma *div-mult1-eq*: $(a*b)$ *div* $c = a*(b$ *div* $c) + a*(b$ *mod* $c)$ *div* $(c::nat)$
apply (*cases* $c = 0$, *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 (*cases* $c = 0$, *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 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*$((b,c),(bq,br))$; $c > 0$ |]
   ==> *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 (*cases* $c = 0$, *simp*)
apply (*blast intro*: *quorem-add1-eq* [*THEN quorem-div*] *quorem-div-mod*)
done

lemma *mod-add1-eq*: $(a+b)$ *mod* $(c::nat) = (a$ *mod* $c + b$ *mod* $c)$ *mod* c
apply (*cases* $c = 0$, *simp*)
apply (*blast intro*: *quorem-div-mod quorem-add1-eq* [*THEN quorem-mod*])
done

19.6 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*))
by (*auto simp add*: *mult-ac quorem-def add-mult-distrib2* [*symmetric*] *mod-lemma*)

lemma *div-mult2-eq*: *a div* (*b*∗*c*) = (*a div b*) *div* (*c*::*nat*)
apply (*cases b* = *0*, *simp*)
apply (*cases 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 (*cases b* = *0*, *simp*)
apply (*cases c* = *0*, *simp*)
apply (*auto simp add*: *mult-commute quorem-div-mod* [*THEN quorem-mult2-eq*,
THEN quorem-mod])
done

19.7 Cancellation of Common Factors in Division

lemma *div-mult-mult-lemma*:
[| (*0*::*nat*) < *b*; *0* < *c* |] ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
by (*auto simp add*: *div-mult2-eq*)

lemma *div-mult-mult1* [*simp*]: (*0*::*nat*) < *c* ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
apply (*cases b* = *0*)
apply (*auto simp add*: *linorder-neq-iff* [*of b*] *div-mult-mult-lemma*)
done

lemma *div-mult-mult2* [*simp*]: (*0*::*nat*) < *c* ==> (*a*∗*c*) *div* (*b*∗*c*) = *a div b*
apply (*drule div-mult-mult1*)
apply (*auto simp add*: *mult-commute*)
done

19.8 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: *m div Suc 0* = *m*
by (*induct m*) (*simp-all add*: *div-geq*)

lemma *div-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 (*cases n* = *0*) *auto*

lemma *nat-mod-mod-trivial* [*simp*]: *m mod n mod n* = (*m mod n* :: *nat*)
  by (*cases n* = *0*) *auto*

19.9 The Divides Relation

lemma *dvdI* [*intro?*]: *n* = *m* ∗ *k* ==> *m dvd n*
  unfolding *dvd-def* **by** *blast*

lemma *dvdE* [*elim?*]: !!*P*. [|*m dvd n*; !!*k*. *n* = *m*∗*k* ==> *P*|] ==> *P*
  unfolding *dvd-def* **by** *blast*

lemma *dvd-0-right* [*iff*]: *m dvd* (*0*::*nat*)

unfolding *dvd-def* **by** (*blast intro*: *mult-0-right* [*symmetric*])

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*)

declare *dvd-0-left-iff* [*noatp*]

lemma *dvd-1-left* [*iff*]: *Suc 0 dvd k*
unfolding *dvd-def* **by** *simp*

lemma *dvd-1-iff-1* [*simp*]: (*m dvd Suc 0*) = (*m* = *Suc 0*)
by (*simp add*: *dvd-def*)

lemma *dvd-refl* [*simp*]: *m dvd* (*m::nat*)
unfolding *dvd-def* **by** (*blast intro*: *mult-1-right* [*symmetric*])

lemma *dvd-trans* [*trans*]: [| *m dvd n*; *n dvd p* |] ==> *m dvd* (*p::nat*)
unfolding *dvd-def* **by** (*blast intro*: *mult-assoc*)

lemma *dvd-anti-sym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n::nat*)
unfolding *dvd-def*
by (*force dest*: *mult-eq-self-implies-10 simp add*: *mult-assoc mult-eq-1-iff*)

op dvd is a partial order

interpretation *dvd*: *order* [*op dvd* $\lambda n\ m$ :: *nat*. *n dvd m* $\wedge$ *m* $\neq$ *n*]
by *unfold-locales* (*auto intro*: *dvd-trans dvd-anti-sym*)

lemma *dvd-add*: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m+n* :: *nat*)
unfolding *dvd-def*
by (*blast intro*: *add-mult-distrib2* [*symmetric*])

lemma *dvd-diff*: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m−n* :: *nat*)
unfolding *dvd-def*
by (*blast intro*: *diff-mult-distrib2* [*symmetric*])

lemma *dvd-diffD*: [| *k dvd m−n*; *k dvd n*; *n≤m* |] ==> *k dvd* (*m::nat*)
apply (*erule linorder-not-less* [*THEN iffD2*, *THEN add-diff-inverse*, *THEN subst*])
apply (*blast intro*: *dvd-add*)
done

lemma *dvd-diffD1*: [| *k dvd m−n*; *k dvd m*; *n≤m* |] ==> *k dvd* (*n::nat*)
by (*drule-tac m* = *m* **in** *dvd-diff*, *auto*)

lemma *dvd-mult*: *k dvd n* ==> *k dvd* (*m∗n* :: *nat*)
unfolding *dvd-def* **by** (*blast intro*: *mult-left-commute*)

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*
  unfolding *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*
  unfolding *dvd-def*
  apply (*erule exE*)
  apply (*simp add*: *mult-ac*)
  done

lemma *dvd-mult-cancel1*: *0*<*m* ==> (*m*∗*n dvd m*) = (*n* = (*1*::*nat*))
  apply *auto*
   apply (*subgoal-tac m*∗*n dvd m*∗*1*)
   apply (*drule dvd-mult-cancel*, *auto*)
  done

```
lemma dvd-mult-cancel2: 0<m ==> (n*m dvd m) = (n = (1::nat))
  apply (subst mult-commute)
  apply (erule dvd-mult-cancel1)
  done

lemma 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 le-imp-power-dvd: !!i::nat. m ≤ n ==> i^m dvd i^n
  apply (unfold dvd-def)
   apply (erule linorder-not-less [THEN iffD2, THEN add-diff-inverse, THEN
subst])
  apply (simp add: power-add)
```

done

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

lemma *power-le-dvd* [*rule-format*]: $k\hat{}j$ *dvd* n $-->$ $i \le j$ $-->$ $k\hat{}i$ *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* $i\hat{}n$; $(1::nat) < i|] ==> m \le n$
apply (*rule power-le-imp-le-exp*, *assumption*)
apply (*erule dvd-imp-le*, *simp*)
done

lemma *mod-eq-0-iff*: $(m$ *mod* $d = 0) = (\exists\, q::nat.\ m = d*q)$
by (*auto simp add*: *dvd-eq-mod-eq-0* [*symmetric*] *dvd-def*)

lemmas *mod-eq-0D* [*dest*!] = *mod-eq-0-iff* [*THEN iffD1*]

lemma *mod-eqD*: $(m$ *mod* $d = r) ==> \exists\, q::nat.\ m = r + q*d$
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** - = (- $\wedge$ (- $\longrightarrow$ $?R$)))
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 ≠ 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 ≠ 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

lemma div-mod-equality':
  fixes m n :: nat
  shows m div n * n = m − m mod n
proof −
  have m mod n ≤ m mod n ..
  from div-mod-equality have
    m div n * n + m mod n − m mod n = m − m mod n by simp
  with diff-add-assoc [OF ‹m mod n ≤ m mod n›, of m div n * n] have
    m div n * n + (m mod n − m mod n) = m − m mod n
    by simp
  then show ?thesis by simp
qed
```

19.10 An "induction" law for modulus arithmetic.

```
lemma mod-induct-0:
  assumes step: ∀ i<p. P i ⟶ P ((Suc i) mod p)
  and base: P i and i: i<p
  shows P 0
proof (rule ccontr)
  assume contra: ¬(P 0)
  from i have p: 0<p by simp
  have ∀ k. 0<k ⟶ ¬ P (p−k) (is ∀ k. ?A k)
  proof
    fix k
    show ?A k
    proof (induct k)
      show ?A 0 by simp — by contradiction
    next
      fix n
      assume ih: ?A n
      show ?A (Suc n)
      proof (clarsimp)
        assume y: P (p − Suc n)
        have n: Suc n < p
        proof (rule ccontr)
          assume ¬(Suc n < p)
          hence p − Suc n = 0
            by simp
          with y contra show False
            by simp
        qed
        hence n2: Suc (p − Suc n) = p−n by arith
        from p have p − Suc n < p by arith
        with y step have z: P ((Suc (p − Suc n)) mod p)
          by blast
        show False
        proof (cases n=0)
          case True
          with z n2 contra show ?thesis by simp
        next
          case False
          with p have p−n < p by arith
          with z n2 False ih show ?thesis by simp
        qed
      qed
    qed
  qed
  moreover
  from i obtain k where 0<k ∧ i+k=p
    by (blast dest: less-imp-add-positive)
  hence 0<k ∧ i=p−k by auto
  moreover
```

```
  note base
  ultimately
  show False by blast
qed

lemma mod-induct:
  assumes step: ∀ i<p. P i ⟶ P ((Suc i) mod p)
  and base: P i and i: i<p and j: j<p
  shows P j
proof -
  have ∀ j<p. P j
  proof
    fix j
    show j<p ⟶ P j (is ?A j)
    proof (induct j)
      from step base i show ?A 0
        by (auto elim: mod-induct-0)
    next
      fix k
      assume ih: ?A k
      show ?A (Suc k)
      proof
        assume suc: Suc k < p
        hence k: k<p by simp
        with ih have P k ..
        with step k have P (Suc k mod p)
          by blast
        moreover
        from suc have Suc k mod p = Suc k
          by simp
        ultimately
        show P (Suc k) by simp
      qed
    qed
  qed
  with j show ?thesis by blast
qed


lemma mod-add-left-eq: ((a::nat) + b) mod c = (a mod c + b) mod c
  apply (rule trans [symmetric])
   apply (rule mod-add1-eq, simp)
  apply (rule mod-add1-eq [symmetric])
  done

lemma mod-add-right-eq: (a+b) mod (c::nat) = (a + (b mod c)) mod c
  apply (rule trans [symmetric])
   apply (rule mod-add1-eq, simp)
  apply (rule mod-add1-eq [symmetric])
```

done

lemma *mod-div-decomp*:
fixes $n\ k :: nat$
obtains $m\ q$ **where** $m = n\ div\ k$ **and** $q = n\ mod\ k$
and $n = m * k + q$
proof −
from *mod-div-equality* **have** $n = n\ div\ k * k + n\ mod\ k$ **by** *auto*
moreover have $n\ div\ k = n\ div\ k$ **..**
moreover have $n\ mod\ k = n\ mod\ k$ **..**
note *that* **ultimately show** *thesis* **by** *blast*
qed

19.11 Code generation for div, mod and dvd on nat

definition [*code func del*]:
$divmod\ (m::nat)\ n = (m\ div\ n,\ m\ mod\ n)$

lemma *divmod-zero* [*code*]: $divmod\ m\ 0 = (0,\ m)$
unfolding *divmod-def* **by** *simp*

lemma *divmod-succ* [*code*]:
$divmod\ m\ (Suc\ k) = (if\ m < Suc\ k\ then\ (0,\ m)\ else$
let
$(p,\ q) = divmod\ (m - Suc\ k)\ (Suc\ k)$
$in\ (Suc\ p,\ q))$
unfolding *divmod-def Let-def split-def*
by (*auto intro*: *div-geq mod-geq*)

lemma *div-divmod* [*code*]: $m\ div\ n = fst\ (divmod\ m\ n)$
unfolding *divmod-def* **by** *simp*

lemma *mod-divmod* [*code*]: $m\ mod\ n = snd\ (divmod\ m\ n)$
unfolding *divmod-def* **by** *simp*

instance *nat* :: *dvd-mod*
by *default* (*simp add*: *dvd-eq-mod-eq-0*)

code-modulename *SML*
Divides Nat

code-modulename *OCaml*
Divides Nat

code-modulename *Haskell*
Divides Nat

hide (**open**) *const divmod*

end

20 Record: Extensible records with structural subtyping

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

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
by *simp*

lemma *rec-UNIV-I*: $\bigwedge x.\ x \in UNIV \equiv True$
by *simp*

lemma *rec-True-simp*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
by *simp*

constdefs
K-record:: $'a \Rightarrow 'b \Rightarrow 'a$
K-record-apply [*simp*, *code func*]: $K\text{-}record\ c\ x \equiv c$

lemma *K-record-comp* [*simp*]: $(K\text{-}record\ c \circ f) = K\text{-}record\ c$
by (*rule ext*) (*simp add*: *K-record-apply comp-def*)

lemma *K-record-cong* [*cong*]: $K\text{-}record\ c\ x = K\text{-}record\ c\ x$
by (*rule refl*)

20.1 Concrete record syntax

nonterminals
ident field-type field-types field fields update updates
syntax
```
-constify            :: id => ident                       (-)
-constify            :: longid => ident                   (-)

-field-type          :: [ident, type] => field-type       ((2- ::/ -))
                     :: field-type => field-types         (-)
-field-types         :: [field-type, field-types] => field-types    (-,/ -)
-record-type         :: field-types => type               ((3'(| - |')))
-record-type-scheme  :: [field-types, type] => type       ((3'(| -,/ (2... ::/ -) |')))

-field               :: [ident, 'a] => field              ((2- =/ -))
                     :: field => fields                   (-)
-fields              :: [field, fields] => fields         (-,/ -)
-record              :: fields => 'a                      ((3'(| - |')))
```

```
  -record-scheme      :: [fields, 'a] => 'a              ((3'(| -,/ (2... =/ -) |')))

  -update-name        :: idt
  -update             :: [ident, 'a] => update           ((2- :=/ -))
                      :: update => updates               (-)
  -updates            :: [update, updates] => updates    (-,/ -)
  -record-update      :: ['a, updates] => 'b             (-/(3'(| - |')) [900,0] 900)
```

syntax (*xsymbols*)

```
  -record-type        :: field-types => type             ((3(|-|)))
  -record-type-scheme :: [field-types, type] => type     ((3(|-,/ (2... ::/ -)|)))
  -record             :: fields => 'a                    ((3(|-|)))
  -record-scheme      :: [fields, 'a] => 'a              ((3(|-,/ (2... =/ -)|)))
  -record-update      :: ['a, updates] => 'b             (-/(3(|-|)) [900,0] 900)
```

use *Tools/record-package.ML*
setup *RecordPackage.setup*

end

21 Hilbert-Choice: Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*
imports *Nat*
uses (*Tools/meson.ML*) (*Tools/specification-package.ML*)
begin

21.1 Hilbert's epsilon

axiomatization
  Eps :: ('a => bool) => 'a
where
  someI: P x ==> P (Eps P)

syntax (*epsilon*)
  -Eps :: [pttrn, bool] => 'a ((3ϵ -./ -) [0, 10] 10)
syntax (*HOL*)
  -Eps :: [pttrn, bool] => 'a ((3@ -./ -) [0, 10] 10)
syntax
  -Eps :: [pttrn, bool] => 'a ((3SOME -./ -) [0, 10] 10)
translations
  SOME x. P == CONST Eps (%x. P)

print-translation ⟪
(∗ to avoid eta−contraction of body ∗)
[(@{const-syntax Eps}, fn [Abs abs] =>

let val (x,t) = *atomic-abs-tr′ abs*
in Syntax.const -Eps \$ *x* \$ *t end*)]
⟫

constdefs
inv :: $('a => 'b) => ('b => 'a)$
inv$(f$ :: $'a => 'b)$ == %*y*. *SOME x*. *f x* = *y*

Inv :: $'a$ *set* => $('a => 'b) => ('b => 'a)$
Inv A f == %*x*. *SOME y*. $y \in A$ & *f y* = *x*

21.2 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

21.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

lemma *choice*: $\forall x. \exists y. Q\ x\ y$ ==> $\exists f. \forall x. Q\ x\ (f\ x)$
by (*fast elim*: *someI*)

lemma *bchoice*: $\forall x \in S. \exists y. Q\ x\ y$ ==> $\exists f. \forall x \in S. Q\ x\ (f\ x)$
by (*fast elim*: *someI*)

21.4 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*; $\forall x.$ *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* $\in$ *range*(*f*) ==> *P*(*inv f y*)
  shows *P x*
proof −
  have *f x* $\in$ *range f* **by** *auto*
  hence *P*(*inv f* (*f x*)) **by** (*rule minor*)
  thus *P x* **by** (*simp add*: *inv-f-f* [*OF injf*])
qed

lemma *inj-iff*: (*inj f*) = (*inv f o f* = *id*)
apply (*simp add*: *o-def expand-fun-eq*)
apply (*blast intro*: *inj-on-inverseI inv-f-f*)
done

lemma *inv-o-cancel*[*simp*]: *inj f* ==> *inv f o f* = *id*
by (*simp add*: *inj-iff*)

lemma *o-inv-o-cancel*[*simp*]: *inj f* ==> *g o inv f o f* = *g*
by (*simp add*: *o-assoc*[*symmetric*])

lemma *inv-image-cancel*[*simp*]:
  inj f ==> *inv f* ' *f* ' *S* = *S*
by (*simp add*: *image-compose*[*symmetric*])

lemma *inj-imp-surj-inv*: *inj f* ==> *surj* (*inv f*)
by (*blast intro*: *surjI inv-f-f*)

lemma *f-inv-f*: *y* ∈ *range*(*f*) ==> *f*(*inv f y*) = *y*
apply (*simp add*: *inv-def*)
apply (*fast intro*: *someI*)
done

lemma *surj-f-inv-f*: *surj f* ==> *f*(*inv f y*) = *y*
by (*simp add*: *f-inv-f surj-range*)

lemma *inv-injective*:
  assumes *eq*: *inv f x* = *inv f y*
    and *x*: *x*: *range f*
    and *y*: *y*: *range f*
  shows *x*=*y*
proof −
  have *f* (*inv f x*) = *f* (*inv f y*) **using** *eq* **by** *simp*
  thus *?thesis* **by** (*simp add*: *f-inv-f x y*)
qed

lemma *inj-on-inv*: *A* <= *range*(*f*) ==> *inj-on* (*inv f*) *A*
by (*fast intro*: *inj-onI elim*: *inv-injective injD*)

lemma *surj-imp-inj-inv*: *surj f* ==> *inj* (*inv f*)
by (*simp add*: *inj-on-inv surj-range*)

lemma *surj-iff*: (*surj f*) = (*f o inv f* = *id*)
apply (*simp add*: *o-def expand-fun-eq*)
apply (*blast intro*: *surjI surj-f-inv-f*)
done

lemma *surj-imp-inv-eq*: [| *surj f*; ∀ *x*. *g*(*f x*) = *x* |] ==> *inv f* = *g*
apply (*rule ext*)
apply (*drule-tac x* = *inv f x* **in** *spec*)
apply (*simp add*: *surj-f-inv-f*)
done

lemma *bij-imp-bij-inv*: *bij f* ==> *bij* (*inv f*)
by (*simp add*: *bij-def inj-imp-surj-inv surj-imp-inj-inv*)

lemma *inv-equality*: [| !!*x*. *g* (*f x*) = *x*; !!*y*. *f* (*g y*) = *y* |] ==> *inv f* = *g*
apply (*rule ext*)
apply (*auto simp add*: *inv-def*)
done

lemma *inv-inv-eq*: *bij f* ==> *inv* (*inv f*) = *f*
apply (*rule inv-equality*)
apply (*auto simp add*: *bij-def surj-f-inv-f*)
done

lemma *o-inv-distrib*: [| *bij f*; *bij g* |] ==> *inv* (*f o g*) = *inv g o inv f*
apply (*rule inv-equality*)
apply (*auto simp add*: *bij-def surj-f-inv-f*)
done

lemma *image-surj-f-inv-f*: *surj f* ==> *f* ' (*inv f* ' *A*) = *A*
by (*simp add*: *image-eq-UN surj-f-inv-f*)

lemma *image-inv-f-f*: *inj f* ==> (*inv f*) ' (*f* ' *A*) = *A*
by (*simp add*: *image-eq-UN*)

lemma *inv-image-comp*: *inj f* ==> *inv f* ' (*f'X*) = *X*
by (*auto simp add*: *image-def*)

lemma *bij-image-Collect-eq*: *bij f* ==> *f* ' *Collect P* = {*y*. *P* (*inv f y*)}
apply *auto*
apply (*force simp add*: *bij-is-inj*)
apply (*blast intro*: *bij-is-surj* [*THEN surj-f-inv-f*, *symmetric*])
done

lemma *bij-vimage-eq-inv-image*: *bij f* ==> *f* −' *A* = *inv f* ' *A*
apply (*auto simp add*: *bij-is-surj* [*THEN surj-f-inv-f*])
apply (*blast intro*: *bij-is-inj* [*THEN inv-f-f*, *symmetric*])
done

21.5 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

21.6 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 = (~(∃f. ∀i. (f(Suc i),f i) ∈ r))

```
apply (simp only: wf-eq-minimal)
apply (rule iffI)
 apply (rule notI)
 apply (erule exE)
 apply (erule-tac x = {w. ∃ i. w=f i} in allE, blast)
apply (erule contrapos-np, simp, clarify)
apply (subgoal-tac ∀ n. nat-rec x (%i y. @z. z:Q & (z,y) :r) n ∈ Q)
 apply (rule-tac x = nat-rec x (%i y. @z. z:Q & (z,y) :r) in exI)
 apply (rule allI, simp)
 apply (rule someI2-ex, blast, blast)
apply (rule allI)
apply (induct-tac n, simp-all)
apply (rule someI2-ex, blast+)
done
```

A dynamically-scoped fact for TFL

```
lemma tfl-some: ∀ P x. P x --> P (Eps P)
  by (blast intro: someI)
```

21.7 Least value operator

```
constdefs
  LeastM :: ['a => 'b::ord, 'a => bool] => 'a
  LeastM m P == SOME x. P x & (∀ y. P y --> m x <= m y)

syntax
  -LeastM :: [pttrn, 'a => 'b::ord, bool] => 'a    (LEAST - WRT -. - [0, 4, 10]
10)
translations
  LEAST x WRT m. P == LeastM m (%x. P)

lemma LeastMI2:
  P x ==> (!!y. P y ==> m x <= m y)
    ==> (!!x. P x ==> ∀ y. P y --> m x ≤ m y ==> Q x)
    ==> Q (LeastM m P)
  apply (simp add: LeastM-def)
  apply (rule someI2-ex, blast, blast)
  done

lemma LeastM-equality:
  P k ==> (!!x. P x ==> m k <= m x)
    ==> m (LEAST x WRT m. P x) = (m k::'a::order)
  apply (rule LeastMI2, assumption, blast)
  apply (blast intro!: order-antisym)
  done

lemma wf-linord-ex-has-least:
  wf r ==> ∀ x y. ((x,y):r^+) = ((y,x)~:r^*) ==> P k
    ==> ∃ x. P x & (!y. P y --> (m x,m y):r^*)
```

apply (*drule wf-trancl* [*THEN wf-eq-minimal* [*THEN iffD1*]])
apply (*drule-tac x* = *m*'*Collect P* **in** *spec*, *force*)
done

lemma *ex-has-least-nat*:
$P\ k ==> \exists x.\ P\ x\ \&\ (\forall y.\ P\ y\ --> m\ x <= (m\ y::nat))$
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*:
$P\ k ==> P\ (LeastM\ m\ P)\ \&\ (\forall y.\ P\ y\ --> m\ (LeastM\ m\ P) <= (m\ y::nat))$
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*)

21.8 Greatest value operator

constdefs
GreatestM :: ['*a* => '*b*::*ord*, '*a* => *bool*] => '*a*
$GreatestM\ m\ P == SOME\ x.\ P\ x\ \&\ (\forall y.\ P\ y\ --> m\ y <= m\ x)$

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*:
$P\ x ==> (!!y.\ P\ y ==> m\ y <= m\ x)$
$==> (!!x.\ P\ x ==> \forall y.\ P\ y\ --> m\ y \leq m\ x ==> Q\ x)$
$==> Q\ (GreatestM\ m\ P)$
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*:
$P\ k ==> \forall x.\ P\ x --> (\exists y.\ P\ y\ \&\ \tilde{}\ ((m\ y::nat) <= m\ x))$
  $==> \exists y.\ P\ y\ \&\ \tilde{}\ (m\ y < m\ k + n)$
apply (*induct n*, *force*)
apply (*force simp add*: *le-Suc-eq*)
done

lemma *ex-has-greatest-nat*:
$P\ k ==> \forall y.\ P\ y --> m\ y < b$
  $==> \exists x.\ P\ x\ \&\ (\forall y.\ P\ y --> (m\ y::nat) <= m\ x)$
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*:
$P\ k ==> \forall y.\ P\ y --> m\ y < b$
  $==> P\ (GreatestM\ m\ P)\ \&\ (\forall y.\ P\ y --> (m\ y::nat) <= m\ (GreatestM\ m\ P))$
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*:
$P\ x ==> \forall y.\ P\ y --> m\ y < b$
  $==> (m\ x::nat) <= m\ (GreatestM\ m\ P)$
apply (*blast dest*: *GreatestM-nat-lemma* [*THEN conjunct2*, *THEN spec*, *of P*])
done

Specialization to *GREATEST*.

lemma *GreatestI*: $P\ (k::nat) ==> \forall y.\ P\ y --> y < b ==> P\ (GREATEST\ x.\ P\ x)$
apply (*simp add*: *Greatest-def*)
apply (*rule GreatestM-natI*, *auto*)

done

lemma *Greatest-le*:
$P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow y < b \Longrightarrow (x{::}nat) <= (GREATEST\ x.\ P\ x)$
apply (*simp add*: *Greatest-def*)
apply (*rule GreatestM-nat-le*, *auto*)
done

21.9 The Meson proof procedure

21.9.1 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 $-->$ and $<->$ (positive and negative occurrences)

lemma *meson-imp-to-disjD*: $P{-}{-}{>}Q \Longrightarrow \sim P \mid Q$
and *meson-not-impD*: $\sim(P{-}{-}{>}Q) \Longrightarrow P\ \&\ \sim Q$
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
and *meson-not-refl-disj-D*: $x \sim= x \mid P \Longrightarrow P$
by *fast+*

21.9.2 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+*

21.9.3 Generating clauses for the Meson Proof Procedure

Disjunctions

lemma *meson-disj-assoc*: $(P|Q)|R ==> P|(Q|R)$
  and *meson-disj-comm*: $P|Q ==> Q|P$
  and *meson-disj-FalseD1*: $False|P ==> P$
  and *meson-disj-FalseD2*: $P|False ==> P$
  by *fast+*

21.10 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*: $\sim P|Q ==> ((\sim P==>P) ==> Q)$
by *blast*

Version for Plaisted's "Postive refinement" of the Meson procedure

lemma *make-refined-neg-rule*: $\sim P|Q ==> (P ==> Q)$
by *blast*

P should be a literal

lemma *make-pos-rule*: $P|Q ==> ((P==>\sim P) ==> Q)$
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′*: $[|P|Q; \sim P|] ==> Q$
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*

21.10.1 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;
⟫

21.11 Meson package

use *Tools/meson.ML*

21.12 Specification package – Hilbertized version

lemma *exE-some*: [| *Ex P* ; *c* == *Eps P* |] ==> *P c*
 by (*simp only*: *someI-ex*)

use *Tools/specification-package.ML*

end

22 Finite-Set: Finite sets

theory *Finite-Set*
imports *Divides*
begin

22.1 Definition and basic properties

inductive *finite* :: *'a set* => *bool*
  where
    emptyI [*simp*, *intro*!]: *finite* {}
  | *insertI* [*simp*, *intro*!]: *finite A* ==> *finite* (*insert a A*)

lemma *ex-new-if-finite*: — does not depend on def of finite at all
  assumes ¬ *finite* (*UNIV* :: *'a set*) **and** *finite A*
  shows $\exists\, a{::}'a.\ a \notin A$
proof –
  from *prems* **have** $A \neq UNIV$ **by** *blast*
  thus *?thesis* **by** *blast*
qed

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
  finite F ==>
    P {} ==> (!!*x F*. *finite F* ==> $x \notin F$ ==> *P F* ==> *P* (*insert x F*)) ==>
P F
  — Discharging $x \notin F$ entails extra work.
proof –
  assume *P* {} **and**
    insert: !!*x F*. *finite F* ==> $x \notin F$ ==> *P F* ==> *P* (*insert x F*)
  assume *finite F*
  thus *P F*
  proof *induct*
    show *P* {} **by** *fact*
    fix *x F* **assume** *F*: *finite F* **and** *P*: *P F*
    show *P* (*insert x F*)
    proof *cases*
      assume $x \in F$
      hence *insert x F* = *F* **by** (*rule insert-absorb*)
      with *P* **show** *?thesis* **by** (*simp only*:)
    next
      assume $x \notin F$
      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** $F \neq \{\} \Longrightarrow$
$[\![\ \bigwedge x.\ P\{x\};$
  $\bigwedge x\ F.\ [\![\ finite\ F;\ F \neq \{\};\ x \notin F;\ P\ F\ ]\!] \Longrightarrow P\ (insert\ x\ F)\ ]\!]$
$\Longrightarrow P\ F$
using *fin*
proof *induct*
  case *empty* **thus** *?case* **by** *simp*
next
  case (*insert x F*)

```
  show ?case
  proof cases
    assume F = {}
    thus ?thesis using ‹P {x}› by simp
  next
    assume F ≠ {}
    thus ?thesis using insert by blast
  qed
qed

lemma finite-subset-induct [consumes 2, case-names empty insert]:
  assumes finite F and F ⊆ A
    and empty: P {}
    and insert: !!a F. finite F ==> a ∈ A ==> a ∉ F ==> P F ==> P (insert
a F)
  shows P F
proof −
  from ‹finite F› and ‹F ⊆ A›
  show ?thesis
  proof induct
    show P {} by fact
  next
    fix x F
    assume finite F and x ∉ F and
      P: F ⊆ A ==> P F and i: insert x F ⊆ A
    show P (insert x F)
    proof (rule insert)
      from i show x ∈ A by blast
      from i have F ⊆ A by blast
      with P show P F .
      show finite F by fact
      show x ∉ F by fact
    qed
  qed
qed
```

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 by fact
```

from *insert.hyps* **obtain** *n f*
  where $A = f$ ‘ $\{i::nat.\ i < n\}$ *inj-on f* $\{i.\ i < n\}$ **by** *blast*
hence *insert a A* $= f(n:=a)$ ‘ $\{i.\ i < Suc\ n\}$
    inj-on $(f(n:=a))$ $\{i.\ i < Suc\ n\}$ **using** *notinA*
  by (*auto simp add*: *image-def Ball-def inj-on-def less-Suc-eq*)
thus *?case* **by** *blast*
qed

lemma *nat-seg-image-imp-finite*:
!!*f A*. $A = f$ ‘ $\{i::nat.\ i<n\} \Longrightarrow$ *finite A*
proof (*induct n*)
  case *0* **thus** *?case* **by** *simp*
next
  case (*Suc n*)
  let $?B = f$ ‘ $\{i.\ i < n\}$
  have *finB*: *finite ?B* **by**(*rule Suc.hyps*[*OF refl*])
  show *?case*
  proof *cases*
    assume $\exists\, k<n.\ f\ n = f\ k$
    hence $A = ?B$ **using** *Suc.prems* **by**(*auto simp*:*less-Suc-eq*)
    thus *?thesis* **using** *finB* **by** *simp*
  next
    assume $\neg(\exists\ k<n.\ f\ n = f\ k)$
    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 $A = (\exists\ (n::nat)\ f.\ A = f$ ‘ $\{i::nat.\ i<n\})$
by(*blast intro*: *nat-seg-image-imp-finite dest*: *finite-imp-nat-seg-image-inj-on*)

22.1.1 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*: *finite*) *simp-all*

lemma *finite-subset*: $A \subseteq B$ ==> *finite B* ==> *finite A*
  — Every subset of a finite set is finite.
proof −
  assume *finite B*
  thus !!A. $A \subseteq B$ ==> *finite A*
  proof *induct*
    case *empty*
    thus *?case* **by** *simp*
  next
    case (*insert x F A*)
    have *A*: $A \subseteq$ *insert x F* **and** *r*: $A - \{x\} \subseteq F$ ==> *finite* $(A - \{x\})$ **by** *fact+*

show *finite A*
proof *cases*
assume x: $x \in A$
with A **have** $A - \{x\} \subseteq F$ **by** (*simp add*: *subset-insert-iff*)
with r **have** *finite* $(A - \{x\})$ **.**
hence *finite* (*insert* x $(A - \{x\})$) **..**
also have *insert* x $(A - \{x\}) = A$ **using** x **by** (*rule insert-Diff*)
finally show *?thesis* **.**
next
show $A \subseteq F$ ==> *?thesis* **by** *fact*
assume $x \notin A$
with A **show** $A \subseteq F$ **by** (*simp add*: *subset-insert-iff*)
qed
qed
qed

lemma *finite-Collect-subset*[*simp*]: *finite* $A \Longrightarrow$ *finite*$\{x \in A.\ P\ x\}$
using *finite-subset*[*of* $\{x \in A.\ P\ x\}$ A] **by** *blast*

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. $M \in A \Longrightarrow$ *finite* M $]\!] \Longrightarrow$ *finite*$(\bigcup A)$
by (*induct rule*:*finite-induct*) *simp-all*

lemma *finite-empty-induct*:
assumes *finite* A
and P A
and !!a A. *finite* A ==> a:A ==> P A ==> P $(A - \{a\})$
shows P {}
proof −
have P $(A - A)$
proof −
{
fix c b :: *'a set*
assume c: *finite* c **and** b: *finite* b
and *P1*: P b **and** *P2*: !!x y. *finite* y ==> $x \in y$ ==> P y ==> P $(y - \{x\})$
have $c \subseteq b$ ==> P $(b - c)$

```
      using c
    proof induct
      case empty
      from P1 show ?case by simp
    next
      case (insert x F)
      have P (b − F − {x})
      proof (rule P2)
        from - b show finite (b − F) by (rule finite-subset) blast
        from insert show x ∈ b − F by simp
        from insert show P (b − F) by simp
      qed
      also have b − F − {x} = b − insert x F by (rule Diff-insert [symmetric])
      finally show ?case .
    qed
  }
  then show ?thesis by this (simp-all add: assms)
qed
then show ?thesis by simp
qed
```

```
lemma finite-Diff [simp]: finite 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
```

```
lemma finite-Diff-singleton [simp]: finite (A − {a}) = finite A
  by simp
```

Image and Inverse Image over Finite Sets

```
lemma finite-imageI[simp]: finite F ==> finite (h ` F)
  — The image of a finite set is finite.
  by (induct set: finite) simp-all
```

```
lemma finite-surj: finite A ==> B <= f ` A ==> finite B
  apply (frule finite-imageI)
  apply (erule finite-subset, assumption)
  done
```

```
lemma finite-range-imageI:
    finite (range g) ==> finite (range (%x. f (g x)))
  apply (drule finite-imageI, simp)
  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* ==> $f-`\{a\} \subseteq \{THE\ x.\ f\ x = a\}$
  — The inverse image of a singleton under an injective function is included in a singleton.
  apply (*auto simp add*: *inj-on-def*)
  apply (*blast intro*: *the-equality* [*symmetric*])
  done

lemma *finite-vimageI*: [|*finite F*; *inj h*|] ==> *finite* (*h* −` *F*)
  — The inverse image of a finite set under an injective function is finite.
  apply (*induct set*: *finite*)
   apply *simp-all*
  apply (*subst vimage-insert*)
  apply (*simp add*: *finite-Un finite-subset* [*OF inj-vimage-singleton*])
  done

The finite UNION of finite sets

lemma *finite-UN-I*: *finite A* ==> (!!*a*. *a*:*A* ==> *finite* (*B a*)) ==> *finite* (*UN a*:*A*. *B a*)
  by (*induct set*: *finite*) *simp-all*

Strengthen RHS to $(\forall x \in A.\ finite\ (B\ x)) \wedge finite\ \{x \in A.\ B\ x \neq \{\}\}$?

We'd need to prove $finite\ C \Longrightarrow \forall A\ B.\ UNION\ A\ B \subseteq C \longrightarrow finite\ \{x \in A.\ B\ x \neq \{\}\}$ 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*($\bigcup$ *A*) $\Longrightarrow$ *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*: *finite*)
   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

22.2 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.

inductive
foldSet :: ($'a => 'a => 'a$) => ($'b => 'a$) => $'a => 'b$ *set* => $'a$ => *bool*
for f :: $'a => 'a => 'a$
and g :: $'b => 'a$
and z :: $'a$
where
emptyI [*intro*]: *foldSet* f g z {} z
| *insertI* [*intro*]:
$[\![$ $x \notin A$; *foldSet* f g z A y $]\!]$
$\Longrightarrow$ *foldSet* f g z (*insert* x A) (f (g x) y)

inductive-cases *empty-foldSetE* [*elim*!]: *foldSet* f g z {} x

constdefs
fold :: ($'a => 'a => 'a$) => ($'b => 'a$) => $'a => 'b$ *set* => $'a$
fold f g z A == *THE* x. *foldSet* f g z A x

A tempting alternative for the definiens is *if finite A then THE x. foldSet f g e A x 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*:
foldSet f g z ($A - \{x\}$) y ==> x: A ==> *foldSet* f g z A (f (g x) y)
by (*erule insert-Diff* [*THEN subst*], *rule foldSet.intros*, *auto*)

lemma *foldSet-imp-finite*: *foldSet* f g z A x==> *finite* A
by (*induct set*: *foldSet*) *auto*

lemma *finite-imp-foldSet*: *finite* A ==> *EX* x. *foldSet* f g z A x
by (*induct set*: *finite*) *auto*

22.2.1 Commutative monoids

locale *ACf* =
fixes f :: $'a => 'a => 'a$ (**infixl** $\cdot$ *70*)
assumes *commute*: $x \cdot y = y \cdot x$
and *assoc*: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$
begin

lemma *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 ... $= y \cdot (z \cdot x)$ **by** (*simp only*: *assoc*)
  also have $z \cdot x = x \cdot z$ **by** (*simp only*: *commute*)
  finally show *?thesis* **.**
qed

lemmas *AC* = *assoc commute left-commute*

end

locale *ACe* = *ACf* +
  fixes $e :: 'a$
  assumes *ident* [*simp*]: $x \cdot e = x$
begin

lemma *left-ident* [*simp*]: $e \cdot x = x$
proof −
  have $x \cdot e = x$ **by** (*rule ident*)
  thus *?thesis* **by** (*subst commute*)
qed

end

locale *ACIf* = *ACf* +
  assumes *idem*: $x \cdot x = x$
begin

lemma *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 *ACI* = *AC idem idem2*

end

22.2.2 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-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*: $n = Suc\ m$ **by** *fact*
  have *mlessn*: $m<n$ **by** (*simp add*: *nSuc*)
  from *aA* **obtain** *k* **where** *hkeq*: $h\ k = a$ **and** *klessn*: $k<n$ **by** (*blast elim!*: *equalityE*)
  let $?hm = swap\ k\ m\ h$
  have *inj-hm*: $inj\text{-}on\ ?hm\ \{i.\ i < n\}$ **using** *klessn mlessn*
    by (*simp add*: *inj-on-swap-iff inj-on*)
  show *?thesis*
  proof (*intro exI conjI*)
    show $inj\text{-}on\ ?hm\ \{i.\ i < m\}$ **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 $inj\text{-}on\ (swap\ k\ m\ h)\ \{i.\ i < Suc\ m\}$ **using** *inj-hm nSuc* **by** *simp*
      show $?hm\ m \notin A$ **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*:
  $!!A\ x\ x'\ h.\ [\![\ A = h`\{i::nat.\ i<n\};\ inj\text{-}on\ h\ \{i.\ i<n\};$
        $foldSet\ f\ g\ z\ A\ x;\ foldSet\ f\ g\ z\ A\ x'\ ]\!]$
   $\Longrightarrow x' = x$
proof (*induct n rule*: *less-induct*)
  case (*less n*)
    have *IH*: $!!m\ h\ A\ x\ x'.$
          $[\![m<n;\ A = h\ `\ \{i.\ i<m\};\ inj\text{-}on\ h\ \{i.\ i<m\};$
          $foldSet\ f\ g\ z\ A\ x;\ foldSet\ f\ g\ z\ A\ x']\!] \Longrightarrow x' = x$ **by** *fact*
    have *Afoldx*: $foldSet\ f\ g\ z\ A\ x$ **and** *Afoldx'*: $foldSet\ f\ g\ z\ A\ x'$
     and *A*: $A = h`\{i.\ i<n\}$ **and** *injh*: $inj\text{-}on\ h\ \{i.\ i<n\}$ **by** *fact+*
    show *?case*
    proof (*rule foldSet.cases* [*OF Afoldx*])

assume $A = \{\}$ **and** $x = z$
with *Afoldx'* **show** $x' = x$ **by** *blast*
next
fix $B\ b\ u$
assume *AbB*: $A = insert\ b\ B$ **and** x: $x = g\ b \cdot u$
and *notinB*: $b \notin B$ **and** *Bu*: *foldSet f g z B u*
show $x'=x$
proof (*rule foldSet.cases* [*OF Afoldx'*])
assume $A = \{\}$ **and** $x' = z$
with *AbB* **show** $x' = x$ **by** *blast*
next
fix $C\ c\ v$
assume *AcC*: $A = insert\ c\ C$ **and** x': $x' = g\ c \cdot v$
and *notinC*: $c \notin C$ **and** *Cv*: *foldSet f g z C v*
from A *AbB* **have** *Beq*: $insert\ b\ B = h`\{i.\ i<n\}$ **by** *simp*
from *insert-inj-onE* [*OF Beq notinB injh*]
obtain *hB mB* **where** *inj-onB*: *inj-on hB* $\{i.\ i < mB\}$
and *Beq*: $B = hB\ `\ \{i.\ i < mB\}$
and *lessB*: $mB < n$ **by** *auto*
from A *AcC* **have** *Ceq*: $insert\ c\ C = h`\{i.\ i<n\}$ **by** *simp*
from *insert-inj-onE* [*OF Ceq notinC injh*]
obtain *hC mC* **where** *inj-onC*: *inj-on hC* $\{i.\ i < mC\}$
and *Ceq*: $C = hC\ `\ \{i.\ i < mC\}$
and *lessC*: $mC < n$ **by** *auto*
show $x'=x$
proof *cases*
assume $b=c$
then moreover have $B = C$ **using** *AbB AcC notinB notinC* **by** *auto*
ultimately show *?thesis* **using** *Bu Cv x x' IH*[*OF lessC Ceq inj-onC*]
by *auto*
next
assume *diff*: $b \neq c$
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*: *foldSet f g z ?D d*
using *finite-imp-foldSet* **by** *iprover*
moreover have *cinB*: $c \in B$ **using** B **by** *auto*
ultimately have *foldSet f g z B* $(g\ c \cdot d)$
by(*rule Diff1-foldSet*)
hence $g\ c \cdot d = u$ **by** (*rule IH* [*OF lessB Beq inj-onB Bu*])
moreover have $g\ b \cdot d = v$
proof (*rule IH*[*OF lessC Ceq inj-onC Cv*])
show *foldSet f g z C* $(g\ b \cdot d)$ **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:
  foldSet f g z A x ==> foldSet f g z A y ==> 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: foldSet f g z A y ==> 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 ==>
    (foldSet f g z (insert x A) v) =
    (EX y. foldSet f g z A y & 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)
```

22.2.3 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: finite)
   apply 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: finite)
   apply 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* $\Longrightarrow$ *fold f g z* (*h ‘ A*) = *fold f* (*g* $\circ$ *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*: *finite*, *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* $\neq$ *j* −−> *A i Int A j* = {} ⟧
  $\Longrightarrow$ *fold f g e* (*UNION I A*) =
      fold f (%*i*. *fold f g e* (*A i*)) *e I*
  apply (*induct set*: *finite*, *simp*, *atomize*)
  apply (*subgoal-tac ALL i*:*F*. *x* $\neq$ *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*: *finite*) *simp-all*

lemma (**in** *ACf*) *fold-cong*:
  finite A $\Longrightarrow$ (!!*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 ⟹
   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
```

Interpretation of locales – see OrderedGroup.thy

```
interpretation AC-add: ACe [op + 0::'a::comm-monoid-add]
  by unfold-locales (auto intro: add-assoc add-commute)

interpretation AC-mult: ACe [op ∗ 1::'a::comm-monoid-mult]
  by unfold-locales (auto intro: mult-assoc mult-commute)
```

22.3 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

abbreviation
  Setsum (∑- [1000] 999) where
  ∑A == setsum (%x. x) A
```

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*$\sum$ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setsum :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-add* ((*3*$\sum$ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM i:*A*. *b* == *setsum* (*%i*. *b*) *A*
$\sum$ *i*∈*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* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3SUM* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*$\sum$ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'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*}

print-translation ⟪
```
let
  fun setsum-tr' [Abs(x,Tx,t), Const (Collect,-) $ Abs(y,Ty,P)] =
    if x<>y then raise Match
    else let val x' = Syntax.mark-bound x
             val t' = subst-bound(x',t)
             val P' = subst-bound(x',P)
         in Syntax.const -qsetsum $ Syntax.mark-bound x $ P' $ t' end
in [(setsum, setsum-tr')] end
```
⟫

lemma *setsum-empty* [*simp*]: *setsum f* {} = *0*
by (*simp add*: *setsum-def*)

lemma *setsum-insert* [*simp*]:
finite F ==> *a* ∉ *F* ==> *setsum f* (*insert a F*) = *f a* + *setsum f F*
by (*simp add*: *setsum-def*)

lemma *setsum-infinite* [*simp*]: ~ *finite A* ==> *setsum f A* = *0*
by (*simp add*: *setsum-def*)

lemma *setsum-reindex*:
inj-on f B ==> *setsum h* (*f* ' *B*) = *setsum* (*h* ∘ *f*) *B*

by(*auto simp add*: *setsum-def 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 $\Longrightarrow$ *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 $\neq$ *j --> A i Int A j = {}) ==>*
   setsum f (UNION I A) = ($\sum$ *i*$\in$*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*. $A \neq B$ *−−>* *A Int B* = {}) |]
   ==> *setsum f* (*Union C*) = *setsum* (*setsum f*) *C*
apply (*cases finite C*)
 prefer *2* **apply** (*force dest*: *finite-UnionD simp add*: *setsum-def*)
  apply (*frule setsum-UN-disjoint* [*of C id f*])
 apply (*unfold Union-def id-def*, *assumption*+)
done

lemma *setsum-Sigma*: *finite A* ==> *ALL x*:*A*. *finite* (*B x*) ==>
    ($\sum$ *x*∈*A*. ($\sum$ *y*∈*B x*. *f x y*)) = ($\sum$ (*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*∈*A*. ($\sum$ *y*∈*B*. *f x y*)) = ($\sum$ (*x*,*y*) ∈ *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*)

22.3.1 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*: *finite*) *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-simps*)

lemma *setsum-Un*: *finite A* ==> *finite B* ==>

  $(setsum\ f\ (A\ Un\ B) :: 'a :: ab\text{-}group\text{-}add) =$
    $setsum\ f\ A + setsum\ f\ B - setsum\ f\ (A\ Int\ B)$
  by (*subst setsum-Un-Int* [*symmetric*], *auto simp add*: *ring-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* $\Longrightarrow$
  $(setsum\ f\ (A - \{a\}) :: ('a{::}ab\text{-}group\text{-}add)) =$
  $(if\ a{:}A\ then\ setsum\ f\ A - f\ a\ else\ setsum\ f\ A)$
  by (*erule finite-induct*) (*auto simp add*: *insert-Diff-if*)

lemma *setsum-diff1'*[*rule-format*]: *finite A* $\Longrightarrow a \in A \longrightarrow (\sum\ x \in A.\ f\ x) = f\ a + (\sum\ x \in (A - \{a\}).\ f\ x)$
  apply (*erule finite-induct*[**where** $F{=}A$ **and** $P{=}\%\ A.\ (a \in A \longrightarrow (\sum\ x \in A.\ f\ x) = f\ a + (\sum\ x \in (A - \{a\}).\ f\ x))$])
  apply (*auto simp add*: *insert-Diff-if add-ac*)
  done

lemma *setsum-diff-nat*:
  assumes *finite B*
    and $B \subseteq A$
  shows $(setsum\ f\ (A - B) :: nat) = (setsum\ f\ A) - (setsum\ f\ B)$
  using *prems*
proof *induct*
  show $setsum\ f\ (A - \{\}) = (setsum\ f\ A) - (setsum\ f\ \{\})$ **by** *simp*
next
  fix *F x* **assume** *finF*: *finite F* **and** *xnotinF*: $x \notin F$
    and *xFinA*: $insert\ x\ F \subseteq A$
    and *IH*: $F \subseteq A \Longrightarrow setsum\ f\ (A - F) = setsum\ f\ A - setsum\ f\ F$
  from *xnotinF xFinA* **have** *xinAF*: $x \in (A - F)$ **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** $F \subseteq A$ **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 $(A - \textit{insert}\ x\ F) = \textit{setsum}\ f\ A - \textit{setsum}\ f\ (\textit{insert}\ x\ F)$
  by *simp*
thus *setsum* f $(A - \textit{insert}\ x\ F) = \textit{setsum}\ f\ A - \textit{setsum}\ f\ (\textit{insert}\ x\ F)$ **by** *simp*
qed

lemma *setsum-diff*:
  assumes *le*: *finite* A $B \subseteq A$
  shows *setsum* f $(A - B) = \textit{setsum}\ f\ A - ((\textit{setsum}\ f\ B)::('a::\textit{ab-group-add}))$
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*: $\bigwedge i.\ i{\in}K \Longrightarrow f\ (i::'a) \leq ((g\ i)::('b::\{\textit{comm-monoid-add}, \textit{pordered-ab-semigroup-add}\}))$
  shows $(\sum i{\in}K.\ f\ i) \leq (\sum i{\in}K.\ g\ i)$
proof (*cases finite* K)
  case *True*
  thus *?thesis* **using** *le*
  proof *induct*
    case *empty*
    thus *?case* **by** *simp*
  next
    case *insert*
    thus *?case* **using** *add-mono* **by** *fastsimp*
  qed
next
  case *False*
  thus *?thesis*
    by (*simp add*: *setsum-def*)
qed

lemma *setsum-strict-mono*:
  fixes $f :: 'a \Rightarrow 'b::\{\textit{pordered-cancel-ab-semigroup-add}, \textit{comm-monoid-add}\}$
  assumes *finite* A $A \neq \{\}$
    and $!!x.\ x{:}A \Longrightarrow f\ x < g\ x$
  shows *setsum* f $A <$ *setsum* g A
  using *prems*
proof (*induct rule*: *finite-ne-induct*)
  case *singleton* **thus** *?case* **by** *simp*
next

```
  case insert thus ?case by (auto simp: add-strict-mono)
qed

lemma setsum-negf:
  setsum (%x. - (f x)::'a::ab-group-add) A = - setsum f A
proof (cases finite A)
  case True thus ?thesis by (induct set: finite) auto
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-subtractf:
  setsum (%x. ((f x)::'a::ab-group-add) - g x) A =
    setsum f A - setsum g A
proof (cases finite A)
  case True thus ?thesis by (simp add: diff-minus setsum-addf setsum-negf)
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-nonneg:
  assumes nn: ∀x∈A. (0::'a::{pordered-ab-semigroup-add,comm-monoid-add}) ≤
f x
  shows 0 ≤ setsum f A
proof (cases finite A)
  case True thus ?thesis using nn
  proof induct
    case empty then show ?case by simp
  next
    case (insert x F)
    then have 0 + 0 ≤ f x + setsum f F by (blast intro: add-mono)
    with insert show ?case by simp
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-nonpos:
 assumes np: ∀x∈A. f x ≤ (0::'a::{pordered-ab-semigroup-add,comm-monoid-add})
  shows setsum f A ≤ 0
proof (cases finite A)
  case True thus ?thesis using np
  proof induct
    case empty then show ?case by simp
  next
    case (insert x F)
    then have f x + setsum f F ≤ 0 + 0 by (blast intro: add-mono)
    with insert show ?case by simp
  qed
```

next
  case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-mono2*:
fixes f :: $'a \Rightarrow 'b$ :: {*pordered-ab-semigroup-add-imp-le*,*comm-monoid-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 ... = *setsum* f $(A \cup (B{-}A))$ **using** *fin finite-subset*[*OF sub fin*]
    by (*simp add*:*setsum-Un-disjoint del*:*Un-Diff-cancel*)
  also have $A \cup (B{-}A) = B$ **using** *sub* **by** *blast*
  finally show *?thesis* **.**
qed

lemma *setsum-mono3*: *finite* B ==> A <= B ==>
    ALL x: B – A.
      0 <= ((*f x*)::'*a*::{*comm-monoid-add*,*pordered-ab-semigroup-add*}) ==>
        setsum f A <= *setsum f B*
  apply (*subgoal-tac setsum f B* = *setsum f A* + *setsum f* (*B* – *A*))
  apply (*erule ssubst*)
  apply (*subgoal-tac setsum f A* + *0* <= *setsum f A* + *setsum f* (*B* – *A*))
  apply *simp*
  apply (*rule add-left-mono*)
  apply (*erule setsum-nonneg*)
  apply (*subst setsum-Un-disjoint* [*THEN sym*])
  apply (*erule finite-subset*, *assumption*)
  apply (*rule finite-subset*)
  prefer *2*
  apply *assumption*
  apply *auto*
  apply (*rule setsum-cong*)
  apply *auto*
done

lemma *setsum-right-distrib*:
  fixes *f* :: '*a* => ('*b*::*semiring-0*)
  shows $r * $ *setsum f A* = *setsum* (%*n*. $r * f\ n$) *A*
proof (*cases finite A*)
  case *True*
  thus *?thesis*
  proof *induct*
    case *empty* **thus** *?case* **by** *simp*
  next
    case (*insert x A*) **thus** *?case* **by** (*simp add*: *right-distrib*)
  qed
next

```
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma setsum-left-distrib:
  setsum f A * (r::'a::semiring-0) = (∑ 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: 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::pordered-ab-group-add-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::pordered-ab-group-add-abs)
```

```
  shows 0 ≤ setsum (%i. abs(f i)) A
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert x A) thus ?case by (auto simp: add-nonneg-nonneg)
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed

lemma abs-setsum-abs[simp]:
  fixes f :: 'a => ('b::pordered-ab-group-add-abs)
  shows abs (∑ a∈A. abs(f a)) = (∑ a∈A. abs(f a))
proof (cases finite A)
  case True
  thus ?thesis
  proof induct
    case empty thus ?case by simp
  next
    case (insert a A)
    hence |∑ a∈insert a A. |f a|| = ||f a| + (∑ a∈A. |f a|)| by simp
    also have ... = ||f a| + |∑ a∈A. |f a|||  using insert by simp
    also have ... = |f a| + |∑ a∈A. |f a||
      by (simp del: abs-of-nonneg)
    also have ... = (∑ a∈insert a A. |f a|) using insert by simp
    finally show ?case .
  qed
next
  case False thus ?thesis by (simp add: setsum-def)
qed
```

Commuting outer and inner summation

```
lemma swap-inj-on:
  inj-on (%(i, j). (j, i)) (A × B)
  by (unfold inj-on-def) fast

lemma swap-product:
  (%(i, j). (j, i)) ` (A × B) = B × A
  by (simp add: split-def image-def) blast

lemma setsum-commute:
  (∑ i∈A. ∑ j∈B. f i j) = (∑ j∈B. ∑ i∈A. f i j)
proof (simp add: setsum-cartesian-product)
  have (∑ (x,y) ∈ A <*> B. f x y) =
    (∑ (y,x) ∈ (%(i, j). (j, i)) ` (A × B). f x y)
    (is ?s = -)
```

apply (*simp add*: *setsum-reindex* [**where** $f = \%(i, j). (j, i)$] *swap-inj-on*)
apply (*simp add*: *split-def*)
done
also have ... = ($\sum (y,x){\in}B \times A.\ f\ x\ y$)
(**is** - = *?t*)
apply (*simp add*: *swap-product*)
done
finally show *?s* = *?t* **.**
qed

lemma *setsum-product*:
fixes f :: $'a$ => ($'b$::*semiring-0*)
shows *setsum f A* $*$ *setsum g B* = ($\sum i{\in}A.\ \sum j{\in}B.\ f\ i * g\ j$)
by (*simp add*: *setsum-right-distrib setsum-left-distrib*) (*rule setsum-commute*)

22.4 Generalized product over a set

constdefs
setprod :: ($'a$ => $'b$) => $'a$ *set* => $'b$::*comm-monoid-mult*
setprod f A == *if finite A then fold* (*op* $*$) *f 1 A else 1*

abbreviation
Setprod ($\prod$ - [*1000*] *999*) **where**
$\prod$ *A* == *setprod* (%*x*. *x*) *A*

syntax
-setprod :: *pttrn* => $'a$ *set* => $'b$ => $'b$::*comm-monoid-mult* ((*3PROD* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-setprod :: *pttrn* => $'a$ *set* => $'b$ => $'b$::*comm-monoid-mult* ((*3*$\prod$ -$\in$-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setprod :: *pttrn* => $'a$ *set* => $'b$ => $'b$::*comm-monoid-mult* ((*3*$\prod$ -$\in$-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
PROD i:*A*. *b* == *setprod* (%*i*. *b*) *A*
$\prod i{\in}A.\ b$ == *setprod* (%*i*. *b*) *A*

Instead of $\prod x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\prod x|P.\ e$.

syntax
-qsetprod :: *pttrn* $\Rightarrow$ *bool* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ ((*3PROD* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetprod :: *pttrn* $\Rightarrow$ *bool* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ ((*3*$\prod$ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetprod :: *pttrn* $\Rightarrow$ *bool* $\Rightarrow$ $'a$ $\Rightarrow$ $'a$ ((*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}

lemma *setprod-empty* [*simp*]: *setprod f* {} = *1*
by (*auto simp add*: *setprod-def*)

lemma *setprod-insert* [*simp*]: [| *finite A*; $a \notin A$ |] ==>
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)
```

22.4.1 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: finite) auto

lemma setprod-zero:
     finite A ==> EX x: A. f x = (0::'a::comm-semiring-1) ==> setprod f A = 0
  apply (induct set: finite, force, clarsimp)
```

apply (*erule disjE*, *auto*)
done

lemma *setprod-nonneg* [*rule-format*]:
$(ALL\ x: A.\ (0::'a::ordered\text{-}idom) \leq f\ x) \dashrightarrow 0 \leq setprod\ f\ A$
apply (*case-tac finite A*)
apply (*induct set*: *finite*, *force*, *clarsimp*)
apply (*subgoal-tac* $0 * 0 \leq f\ x * setprod\ f\ F$, *force*)
apply (*rule mult-mono*, *assumption+*)
apply (*auto simp add*: *setprod-def*)
done

lemma *setprod-pos* [*rule-format*]: $(ALL\ x: A.\ (0::'a::ordered\text{-}idom) < f\ x)$
$\dashrightarrow 0 < setprod\ f\ A$
apply (*case-tac finite A*)
apply (*induct set*: *finite*, *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*]:
$(ALL\ x\ y.\ (x::'a::comm\text{-}semiring\text{-}1) * y = 0 \dashrightarrow x = 0 \mid y = 0) ==>$
$finite\ A ==> (ALL\ x: A.\ f\ x \neq (0::'a)) \dashrightarrow setprod\ f\ A \neq 0$
apply (*erule finite-induct*, *auto*)
done

lemma *setprod-zero-eq*:
$(ALL\ x\ y.\ (x::'a::comm\text{-}semiring\text{-}1) * y = 0 \dashrightarrow x = 0 \mid y = 0) ==>$
$finite\ A ==> (setprod\ f\ A = (0::'a)) = (EX\ x: A.\ f\ x = 0)$
apply (*insert setprod-zero* [*of A f*] *setprod-nonzero* [*of A f*], *blast*)
done

lemma *setprod-nonzero-field*:
$finite\ A ==> (ALL\ x: A.\ f\ x \neq (0::'a::idom)) ==> setprod\ f\ A \neq 0$
apply (*rule setprod-nonzero*, *auto*)
done

lemma *setprod-zero-eq-field*:
$finite\ A ==> (setprod\ f\ A = (0::'a::idom)) = (EX\ x: A.\ f\ x = 0)$
apply (*rule setprod-zero-eq*, *auto*)
done

lemma *setprod-Un*: $finite\ A ==> finite\ B ==> (ALL\ x: A\ Int\ B.\ f\ x \neq 0) ==>$
$(setprod\ f\ (A\ Un\ B) :: 'a ::\{field\})$
$= setprod\ f\ A * setprod\ f\ B\ /\ setprod\ f\ (A\ Int\ B)$
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*)
done

lemma *setprod-diff1*: *finite A ==> f a* $\neq$ *0 ==>*
(*setprod f* (*A* $-$ {*a*}) :: *'a* :: {*field*}) =
(*if a*:*A then setprod f A* / *f a else setprod f A*)
by (*erule finite-induct*) (*auto simp add*: *insert-Diff-if*)

lemma *setprod-inversef*: *finite A ==>*
ALL x: *A*. *f x* $\neq$ (*0*::*'a*::{*field*,*division-by-zero*}) *==>*
setprod (*inverse* $\circ$ *f*) *A* = *inverse* (*setprod f A*)
apply (*erule finite-induct*)
apply (*simp*, *simp*)
done

lemma *setprod-dividef*:
[|*finite A*;
$\forall x \in A$. *g x* $\neq$ (*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* $\circ$ *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

22.5 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 $\notin$ *A ==> card* (*insert x A*) = *Suc*(*card A*)

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

lemma *card-insert-if*:
finite A ==> card (insert x A) = (if x:A then card A else Suc(card(A)))
by (*simp add*: *insert-absorb*)

lemma *card-0-eq* [*simp*,*noatp*]: *finite A ==> (card A = 0) = (A = {})*
apply *auto*
apply (*drule-tac a = x* **in** *mk-disjoint-insert*, *clarify*, *auto*)
done

lemma *card-eq-0-iff*: (*card A = 0*) = (*A = {} | ~ finite A*)
by *auto*

lemma *card-Suc-Diff1*: *finite A ==> x: A ==> Suc (card (A − {x})) = card A*
apply(*rule-tac t = A* **in** *insert-Diff* [*THEN subst*], *assumption*)
apply(*simp del*:*insert-Diff-single*)
done

lemma *card-Diff-singleton*:
finite A ==> x: A ==> card (A − {x}) = card A − 1
by (*simp add*: *card-Suc-Diff1* [*symmetric*])

lemma *card-Diff-singleton-if*:
finite A ==> card (A−{x}) = (if x : A then card A − 1 else card A)
by (*simp add*: *card-Diff-singleton*)

lemma *card-Diff-insert*[*simp*]:
assumes *finite A* **and** *a*:*A* **and** *a ~: B*
shows *card(A − insert a B) = card(A − B) − 1*
proof −
have *A − insert a B = (A − B) − {a}* **using** *assms* **by** *blast*
then show *?thesis* **using** *assms* **by**(*simp add*:*card-Diff-singleton*)
qed

lemma *card-insert*: *finite A ==> card (insert x A) = Suc (card (A − {x}))*
by (*simp add*: *card-insert-if card-Suc-Diff1 del*:*card-Diff-insert*)

lemma *card-insert-le*: *finite A ==> card A <= card (insert x A)*
by (*simp add*: *card-insert-if*)

lemma *card-mono*: $[\![$ *finite B*; $A \subseteq B$ $]\!] \Longrightarrow$ *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*: *finite*, *simp*, *clarify*)
apply (*subgoal-tac finite A & A − {x} <= F*)

prefer *2* **apply** (*blast intro*: *finite-subset*, *atomize*)
apply (*drule-tac* $x = A - \{x\}$ **in** *spec*)
apply (*simp add*: *card-Diff-singleton-if split add*: *split-if-asm*)
apply (*case-tac card A*, *auto*)
done

lemma *psubset-card-mono*: *finite B* ==> $A < B$ ==> *card A* < *card B*
apply (*simp add*: *psubset-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 del*:*card-Diff-insert*)
done

lemma *card-Diff2-less*:
finite A ==> *x*: *A* ==> *y*: *A* ==> *card* ($A - \{x\} - \{y\}$) < *card A*
apply (*case-tac* $x = y$)
apply (*simp add*: *card-Diff1-less del*:*card-Diff-insert*)
apply (*rule less-trans*)
prefer *2* **apply** (*auto intro*!: *card-Diff1-less simp del*:*card-Diff-insert*)
done

lemma *card-Diff1-le*: *finite A* ==> *card* ($A - \{x\}$) <= *card A*
apply (*case-tac x* : *A*)
apply (*simp-all add*: *card-Diff1-less less-imp-le*)
done

lemma *card-psubset*: *finite B* ==> $A \subseteq B$ ==> *card A* < *card B* ==> *A* < *B*
by (*erule psubsetI*, *blast*)

lemma *insert-partition*:
$[\![\, x \notin F;\ \forall c1 \in insert\ x\ F.\ \forall c2 \in insert\ x\ F.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}\,]\!]$
$\Longrightarrow x \cap \bigcup F = \{\}$
by *auto*

main cardinality theorem

lemma *card-partition* [*rule-format*]:
  finite C ==>
    finite ($\bigcup$ C) -->
    ($\forall c \in C$. *card* $c = k$) -->
    ($\forall c1 \in C$. $\forall c2 \in C$. $c1 \neq c2$ --> $c1 \cap c2 = \{\}$) -->
    $k *$ *card*(C) = *card* ($\bigcup$ C)
apply (*erule finite-induct*, *simp*)
apply (*simp add*: *card-insert-disjoint card-Un-disjoint insert-partition*
  finite-subset [*of* - $\bigcup$ (*insert x F*)])
done

The form of a finite set of given cardinality

lemma *card-eq-SucD*:
assumes *card* A = *Suc* k
shows $\exists b\ B$. A = *insert* b B & $b \notin B$ & *card* $B = k$ & (k=*0* $\longrightarrow$ B={})
proof –
  have *fin*: *finite* A **using** *assms* **by** (*auto intro*: *ccontr*)
  moreover have *card* $A \neq 0$ **using** *assms* **by** *auto*
  ultimately obtain b **where** b: $b \in A$ **by** *auto*
  show *?thesis*
  proof (*intro exI conjI*)
    show A = *insert* b ($A - \{b\}$) **using** b **by** *blast*
    show $b \notin A - \{b\}$ **by** *blast*
    show *card* ($A - \{b\}$) = k **and** $k = 0 \longrightarrow A - \{b\} = \{\}$
      using *assms b fin* **by**(*fastsimp dest*:*mk-disjoint-insert*)+
  qed
qed

lemma *card-Suc-eq*:
  (*card* A = *Suc* k) =
   ($\exists b\ B$. A = *insert* b B & $b \notin B$ & *card* $B = k$ & (k=*0* $\longrightarrow$ B={}))
apply(*rule iffI*)
 apply(*erule card-eq-SucD*)
apply(*auto*)
apply(*subst card-insert*)
 apply(*auto intro*:*ccontr*)
done

lemma *setsum-constant* [*simp*]: ($\sum x \in A$. y) = *of-nat*(*card* A) $*$ y
apply (*cases finite A*)
apply (*erule finite-induct*)
apply (*auto simp add*: *ring-simps*)
done

lemma *setprod-constant*: *finite* A ==> ($\prod x \in A$. (y::'a::{*recpower*, *comm-monoid-mult*}))
= y^(*card* A)
  apply (*erule finite-induct*)
  apply (*auto simp add*: *power-Suc*)
  done

```
lemma setsum-bounded:
  assumes le: ⋀i. i∈A ⟹ f i ≤ (K::'a::{semiring-1, pordered-ab-semigroup-add})
  shows setsum f A ≤ 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
```

22.5.1 Cardinality of unions

```
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) = (∑ i∈I. card(A i))
  apply (simp add: card-def del: setsum-constant)
  apply (subgoal-tac
           setsum (%i. card (A i)) I = setsum (%i. (setsum (%x. 1) (A i))) I)
  apply (simp add: setsum-UN-disjoint del: setsum-constant)
  apply (simp cong: setsum-cong)
  done
```

```
lemma card-Union-disjoint:
  finite C ==> (ALL A:C. finite A) ==>
        (ALL A:C. ALL B:C. A ≠ B --> A Int B = {}) ==>
      card (Union C) = setsum card C
  apply (frule card-UN-disjoint [of C id])
  apply (unfold Union-def id-def, assumption+)
  done
```

22.5.2 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: finite)
   apply simp
  apply (simp add: le-SucI finite-imageI card-insert-if)
  done
```

```
lemma card-image: inj-on f A ==> card (f ` A) = card A
by(simp add:card-def setsum-reindex o-def del:setsum-constant)
```

lemma *endo-inj-surj*: *finite* A ==> f ' $A \subseteq A$ ==> *inj-on* f A ==> f ' $A = A$
by (*simp add*: *card-seteq card-image*)

lemma *eq-card-imp-inj-on*:
[| *finite* A; *card*(f ' A) = *card* A |] ==> *inj-on* f A
apply (*induct rule*:*finite-induct*)
apply *simp*
apply(*frule card-image-le*[**where** $f = f$])
apply(*simp add*:*card-insert-if split*:*if-splits*)
done

lemma *inj-on-iff-eq-card*:
finite A ==> *inj-on* f A = (*card*(f ' A) = *card* A)
by(*blast intro*: *card-image eq-card-imp-inj-on*)

lemma *card-inj-on-le*:
[|*inj-on* f A; f ' $A \subseteq B$; *finite* B |] ==> *card* $A \leq$ *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 ' $A \subseteq B$; *inj-on* g B; g ' $B \subseteq A$;
finite A; *finite* B |] ==> *card* A = *card* B
by (*auto intro*: *le-anti-sym card-inj-on-le*)

22.5.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
$[\![$ *finite* A; *ALL* a:A. *finite* (B a) $]\!]$
$\Longrightarrow$ *card* (*SIGMA* x: A. B x) = ($\sum a \in A$. *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*)

22.5.4 Cardinality of the Powerset

lemma *card-Pow*: *finite* A ==> *card* (*Pow* A) = *Suc* (*Suc* *0*) ^ *card* A
apply (*induct set*: *finite*)

```
  apply (simp-all add: Pow-insert)
 apply (subst card-Un-disjoint, blast)
   apply (blast intro: finite-imageI, blast)
 apply (subgoal-tac inj-on (insert x) (Pow F))
  apply (simp add: card-image Pow-insert)
 apply (unfold inj-on-def)
 apply (blast elim!: equalityE)
 done
```

Relates to equivalence classes. Based on a theorem of F. Kammüller.

```
lemma dvd-partition:
 finite (Union C) ==>
   ALL c : C. k dvd card c ==>
   (ALL c1: C. ALL c2: C. c1 ≠ c2 --> c1 Int c2 = {}) ==>
 k dvd card (Union C)
apply(frule finite-UnionD)
apply(rotate-tac -1)
 apply (induct set: finite, simp-all, clarify)
 apply (subst card-Un-disjoint)
 apply (auto simp add: dvd-add disjoint-eq-subset-Compl)
 done
```

22.5.5 Relating injectivity and surjectivity

```
lemma finite-surj-inj: finite(A) ⟹ A <= f‘A ⟹ inj-on f A
apply(rule eq-card-imp-inj-on, assumption)
apply(frule finite-imageI)
apply(drule (1) card-seteq)
apply(erule card-image-le)
apply simp
done

lemma finite-UNIV-surj-inj: fixes f :: 'a ⇒ 'a
shows finite(UNIV:: 'a set) ⟹ surj f ⟹ inj f
by (blast intro: finite-surj-inj subset-UNIV dest:surj-range)

lemma finite-UNIV-inj-surj: fixes f :: 'a ⇒ 'a
shows finite(UNIV:: 'a set) ⟹ inj f ⟹ surj f
by(fastsimp simp:surj-def dest!: endo-inj-surj)

corollary infinite-UNIV-nat: ~finite(UNIV::nat set)
proof
 assume finite(UNIV::nat set)
 with finite-UNIV-inj-surj[of Suc]
 show False by simp (blast dest: Suc-neq-Zero surjD)
qed
```

22.6 A fold functional for non-empty sets

Does not require start value.

inductive
fold1Set :: $('a => 'a => 'a) => 'a\ set => 'a => bool$
for $f :: 'a => 'a => 'a$
where
fold1Set-insertI [*intro*]:
$[\![$ *foldSet f id a A x*; $a \notin A$ $]\!] \Longrightarrow$ *fold1Set f* (*insert a A*) *x*

constdefs
fold1 :: $('a => 'a => 'a) => 'a\ set => 'a$
fold1 f A == *THE x*. *fold1Set f A x*

lemma *fold1Set-nonempty*:
fold1Set f A x $\Longrightarrow A \neq \{\}$
by(*erule fold1Set.cases*, *simp-all*)

inductive-cases *empty-fold1SetE* [*elim*!]: *fold1Set f* {} *x*

inductive-cases *insert-fold1SetE* [*elim*!]: *fold1Set f* (*insert a X*) *x*

lemma *fold1Set-sing* [*iff*]: (*fold1Set f* {*a*} *b*) = (*a* = *b*)
by (*blast intro*: *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*. *fold1Set f A x*
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*: *foldSet f id b A y*
shows $b \notin A \Longrightarrow$ *foldSet f id z* (*insert b A*) $(z \cdot y)$
using *fold*
proof (*induct rule*: *foldSet.induct*)
case *emptyI* **thus** *?case* **by** (*force simp add*: *fold-insert-aux commute*)
next
case (*insertI x A y*)
have *foldSet f* ($\lambda u.\ u$) *z* (*insert x* (*insert b A*)) $(x \cdot (z \cdot y))$
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*: *foldSet f id b A x*
shows !!a. $[\![a \in A;\ b \notin A]\!] \Longrightarrow$ *foldSet f id a* (*insert b* $(A - \{a\})$) x
using *fold*
proof (*induct rule*: *foldSet.induct*)
  case *emptyI* **thus** *?case* **by** *simp*
next
  case (*insertI x A y*)
  have $a = x \lor a \in A$ **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*: $a \in A$
    hence *foldSet f id a* (*insert x* (*insert b* $(A - \{a\})$)) $(x \cdot y)$
      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*; $a \notin A$|] ==> *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 foldSet f id a A x*)
 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*: $(A \neq \{\}) = (\exists x\ B.\ A = insert\ x\ B\ \&\ x \notin B)$
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*: $A \neq \{\}$ **and** *A*: *finite A* $x \notin A$

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

lemma (**in** *ACIf*) *hom-fold1-commute*:
assumes *hom*: !!*x y*. *h*(*f x y*) = *f* (*h x*) (*h y*)
and *N*: *finite N N* ≠ {} **shows** *h*(*fold1 f N*) = *fold1 f* (*h* ‘ *N*)
using *N* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case (*insert n N*)
then have *h*(*fold1 f* (*insert n N*)) = *h*(*f n* (*fold1 f N*)) **by** *simp*
also have ... = *f* (*h n*) (*h*(*fold1 f N*)) **by**(*rule hom*)
also have *h*(*fold1 f N*) = *fold1 f* (*h* ‘ *N*) **by**(*rule insert*)
also have *f* (*h n*) ... = *fold1 f* (*insert* (*h n*) (*h* ‘ *N*))
using *insert* **by**(*simp*)
also have *insert* (*h n*) (*h* ‘ *N*) = *h* ‘ *insert n N* **by** *simp*
finally show *?case* **.**
qed

Now the recursion rules for definitions:

lemma *fold1-singleton-def*: $g = fold1\ f \Longrightarrow g\ \{a\} = a$
by(*simp add*:*fold1-singleton*)

lemma (**in** *ACf*) *fold1-insert-def*:
$[\![\ g = fold1\ f;\ finite\ A;\ x \notin A;\ A \neq \{\}\ ]\!] \Longrightarrow g\ (insert\ x\ A) = x \cdot (g\ A)$
by(*simp add*:*fold1-insert*)

lemma (**in** *ACIf*) *fold1-insert-idem-def*:
$[\![\ g = fold1\ f;\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow g\ (insert\ x\ A) = x \cdot (g\ A)$
by(*simp add*:*fold1-insert-idem*)

22.6.1 Determinacy for *fold1Set*

Not actually used!!

lemma (**in** *ACf*) *foldSet-permute*:
[|*foldSet f id b* (*insert a A*) *x*; $a \notin A$; $b \notin A$|]
==> *foldSet f id a* (*insert b A*) *x*
apply (*case-tac a*=*b*)
apply (*auto dest*: *foldSet-permute-diff*)
done

lemma (**in** *ACf*) *fold1Set-determ*:
fold1Set f A x ==> *fold1Set f A y* ==> *y* = *x*
proof (*clarify elim*!: *fold1Set.cases*)
fix *A x B y a b*
assume *Ax*: *foldSet f id a A x*
assume *By*: *foldSet f id b B y*
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*: $a \neq b$
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 *foldSet f id a* (*insert b ?D*) *y*
by (*auto intro*: *foldSet-permute simp add*: *insert-absorb*)
moreover
have *foldSet f id a* (*insert b ?D*) *x*
by (*simp add*: *A* [*symmetric*] *Ax*)
ultimately show *?thesis* **by** (*blast intro*: *foldSet-determ*)
qed

qed

lemma (**in** *ACf*) *fold1Set-equality*: *fold1Set f A y ==> 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 proofs involve these relations.

22.6.2 Semi-Lattices

locale *ACIfSL* = *ord* + *ACIf* +
  assumes *below-def*: *less-eq x y* $\longleftrightarrow$ $x \cdot y = x$
  and *strict-below-def*: *less x y* $\longleftrightarrow$ *less-eq x y* $\wedge$ $x \neq y$
begin

notation
  less ((-/ $\prec$ -) [*51*, *51*] *50*)

notation (*xsymbols*)
  less-eq ((-/ $\preceq$ -) [*51*, *51*] *50*)

notation (*HTML* **output**)
  less-eq ((-/ $\preceq$ -) [*51*, *51*] *50*)

lemma *below-refl* [*simp*]: $x \preceq x$
  by (*simp add*: *below-def idem*)

lemma *below-antisym*:
  assumes *xy*: $x \preceq y$ **and** *yx*: $y \preceq x$
  shows $x = y$
  using *xy* [*unfolded below-def*, *symmetric*]
    yx [*unfolded below-def commute*]
  by (*rule trans*)

lemma *below-trans*:
  assumes *xy*: $x \preceq y$ **and** *yz*: $y \preceq z$
  shows $x \preceq z$
proof −
  from *xy* **have** *x-xy*: $x \cdot y = x$ **by** (*simp add*: *below-def*)
  from *yz* **have** *y-yz*: $y \cdot z = y$ **by** (*simp add*: *below-def*)
  from *y-yz* **have** $x \cdot y \cdot z = x \cdot y$ **by** (*simp add*: *assoc*)
  with *x-xy* **have** $x \cdot y \cdot z = x$ **by** *simp*
  moreover from *x-xy* **have** $x \cdot z = x \cdot y \cdot z$ **by** *simp*
  ultimately have $x \cdot z = x$ **by** *simp*
  then show *?thesis* **by** (*simp add*: *below-def*)
qed

lemma *below-f-conv* [*simp*,*noatp*]: $x \preceq y \cdot z = (x \preceq y \wedge x \preceq z)$
proof
  assume $x \preceq y \cdot z$
  hence *xyzx*: $x \cdot (y \cdot z) = x$ **by**(*simp add*: *below-def*)
  have $x \cdot y = x$
  proof −
    have $x \cdot y = (x \cdot (y \cdot z)) \cdot y$ **by**(*rule subst*[*OF xyzx*], *rule refl*)
    also have $\ldots = x \cdot (y \cdot z)$ **by**(*simp add*:*ACI*)
    also have $\ldots = x$ **by**(*rule xyzx*)
    finally show *?thesis* **.**
  qed
  moreover have $x \cdot z = x$
  proof −
    have $x \cdot z = (x \cdot (y \cdot z)) \cdot z$ **by**(*rule subst*[*OF xyzx*], *rule refl*)
    also have $\ldots = x \cdot (y \cdot z)$ **by**(*simp add*:*ACI*)
    also have $\ldots = x$ **by**(*rule xyzx*)
    finally show *?thesis* **.**
  qed
  ultimately show $x \preceq y \wedge x \preceq z$ **by**(*simp add*: *below-def*)
next
  assume *a*: $x \preceq y \wedge x \preceq z$
  hence *y*: $x \cdot y = x$ **and** *z*: $x \cdot z = x$ **by**(*simp-all add*: *below-def*)
  have $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ **by**(*simp add*:*assoc*)
  also have $x \cdot y = x$ **using** *a* **by**(*simp-all add*: *below-def*)
  also have $x \cdot z = x$ **using** *a* **by**(*simp-all add*: *below-def*)
  finally show $x \preceq y \cdot z$ **by**(*simp-all add*: *below-def*)
qed

end

interpretation *ACIfSL* $<$ *order*
by *unfold-locales*
  (*simp add*: *strict-below-def*, *auto intro*: *below-refl below-trans below-antisym*)

locale *ACIfSLlin* = *ACIfSL* +
  assumes *lin*: $x{\cdot}y \in \{x{,}y\}$
begin

lemma *above-f-conv*:
 $x \cdot y \preceq z = (x \preceq z \vee y \preceq z)$
proof
  assume *a*: $x \cdot y \preceq z$
  have $x \cdot y = x \vee x \cdot y = y$ **using** *lin*[*of x y*] **by** *simp*
  thus $x \preceq z \vee y \preceq z$
  proof
    assume $x \cdot y = x$ **hence** $x \preceq z$ **by**(*rule subst*)(*rule a*) **thus** *?thesis* **..**
  next
    assume $x \cdot y = y$ **hence** $y \preceq z$ **by**(*rule subst*)(*rule a*) **thus** *?thesis* **..**
  qed

next
  assume $x \preceq z \vee y \preceq z$
  thus $x \cdot y \preceq z$
  proof
    assume *a*: $x \preceq z$
    have $(x \cdot y) \cdot z = (x \cdot z) \cdot y$ **by**(*simp add*:*ACI*)
    also have $x \cdot z = x$ **using** *a* **by**(*simp add*:*below-def*)
    finally show $x \cdot y \preceq z$ **by**(*simp add*:*below-def*)
  next
    assume *a*: $y \preceq z$
    have $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ **by**(*simp add*:*ACI*)
    also have $y \cdot z = y$ **using** *a* **by**(*simp add*:*below-def*)
    finally show $x \cdot y \preceq z$ **by**(*simp add*:*below-def*)
  qed
qed

lemma *strict-below-f-conv*[*simp*,*noatp*]: $x \prec y \cdot z = (x \prec y \wedge x \prec z)$
apply(*simp add*: *strict-below-def*)
using *lin*[*of y z*] **by** (*auto simp*:*below-def ACI*)

lemma *strict-above-f-conv*:
  $x \cdot y \prec z = (x \prec z \vee y \prec z)$
apply(*simp add*: *strict-below-def above-f-conv*)
using *lin*[*of y z*] *lin*[*of x z*] **by** (*auto simp*:*below-def ACI*)

end

interpretation *ACIfSLlin* < *linorder*
  by *unfold-locales*
    (*insert lin* [*simplified insert-iff*], *simp add*: *below-def commute*)

22.6.3 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 $x \preceq$ *fold1 f A* = ($\forall a{\in}A.\ x \preceq a$)
using *A*
by(*induct rule*:*finite-ne-induct*) *simp-all*

lemma (**in** *ACIfSLlin*) *strict-below-fold1-iff*:
  finite A $\Longrightarrow A \neq \{\} \Longrightarrow x \prec$ *fold1 f A* = ($\forall a{\in}A.\ x \prec a$)
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* $\preceq a$
using *A*
proof (*induct rule*:*finite-ne-induct*)
  case *singleton* **thus** *?case* **by** *simp*
next
  case (*insert x F*)
  from *insert*(*5*) **have** $a = x \vee a \in F$ **by** *simp*
  thus *?case*
  proof
    assume $a = x$ **thus** *?thesis* **using** *insert* **by**(*simp add*:*below-def ACI*)
  next
    assume $a \in F$
    hence *bel*: *fold1 f F* $\preceq a$ **by**(*rule insert*)
    have *fold1 f* (*insert x F*) $\cdot\ a = x \cdot$ (*fold1 f F* $\cdot\ a$)
      using *insert* **by**(*simp add*:*below-def ACI*)
    also have *fold1 f F* $\cdot\ a$ = *fold1 f F*
      using *bel* **by**(*simp add*:*below-def ACI*)
    also have $x \cdot \ldots$ = *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* $A \neq \{\}$
shows *fold1 f A* $\preceq x = (\exists a \in A.\ a \preceq x)$
using *A*
by(*induct rule*:*finite-ne-induct*)(*simp-all add*:*above-f-conv*)

lemma (**in** *ACIfSLlin*) *fold1-strict-below-iff*:
assumes *A*: *finite A* $A \neq \{\}$
shows *fold1 f A* $\prec x = (\exists a \in A.\ a \prec x)$
using *A*
by(*induct rule*:*finite-ne-induct*)(*simp-all add*:*strict-above-f-conv*)

lemma (**in** *ACIfSLlin*) *fold1-antimono*:
assumes $A \neq \{\}$ **and** $A \subseteq B$ **and** *finite B*
shows *fold1 f B* $\preceq$ *fold1 f A*
proof(*cases*)
  assume $A = B$ **thus** *?thesis* **by** *simp*
next
  assume $A \neq B$
  have *B*: $B = A \cup (B-A)$ **using** $\langle A \subseteq B\rangle$ **by** *blast*
  have *fold1 f B* = *fold1 f* $(A \cup (B-A))$ **by**(*subst B*)(*rule refl*)
  also have ... = *f* (*fold1 f A*) (*fold1 f* $(B-A)$)
  proof −
    have *finite A* **by**(*rule finite-subset*[*OF* $\langle A \subseteq B\rangle$ ⟨*finite B*⟩])
    moreover have *finite*$(B-A)$ **by**(*rule finite-Diff*[*OF* ⟨*finite B*⟩])
    moreover have $(B-A) \neq \{\}$ **using** *prems* **by** *blast*
    moreover have *A Int* $(B-A) = \{\}$ **using** *prems* **by** *blast*
    ultimately show *?thesis* **using** $\langle A \neq \{\}\rangle$ **by**(*rule-tac fold1-Un*)
  qed
  also have ... $\preceq$ *fold1 f A* **by**(*simp add*: *above-f-conv*)
  finally show *?thesis* **.**
qed

22.6.4 Fold1 in lattices with *inf* and *sup*

As an application of *fold1* we define infimum and supremum in (not necessarily complete!) lattices over (non-empty) sets by means of *fold1*.

lemma (**in** *lower-semilattice*) *ACf-inf*: *ACf inf*
  by (*blast intro*: *ACf.intro inf-commute inf-assoc*)

lemma (**in** *upper-semilattice*) *ACf-sup*: *ACf sup*
  by (*blast intro*: *ACf.intro sup-commute sup-assoc*)

lemma (**in** *lower-semilattice*) *ACIf-inf*: *ACIf inf*
apply(*rule ACIf.intro*)
apply(*rule ACf-inf*)

apply(*rule ACIf-axioms.intro*)
apply(*rule inf-idem*)
done

lemma (**in** *upper-semilattice*) *ACIf-sup*: *ACIf sup*
apply(*rule ACIf.intro*)
apply(*rule ACf-sup*)
apply(*rule ACIf-axioms.intro*)
apply(*rule sup-idem*)
done

lemma (**in** *lower-semilattice*) *ACIfSL-inf*: *ACIfSL* (*op* $\leq$) (*op* $<$) *inf*
apply(*rule ACIfSL.intro*)
apply(*rule ACIf.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-greatest refl*)
apply(*erule subst*)
apply(*rule inf-le2*)
apply(*rule less-le*)
done

lemma (**in** *upper-semilattice*) *ACIfSL-sup*: *ACIfSL* (%x y. $y \leq x$) (%x y. $y < x$) *sup*
apply(*rule ACIfSL.intro*)
apply(*rule ACIf.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-least refl*)
apply(*erule subst*)
apply(*rule sup-ge2*)
apply(*simp add*: *neq-commute less-le*)
done

context *lattice*
begin

definition
  Inf-fin :: $'a$ *set* $\Rightarrow$ $'a$ ($\bigsqcap_{fin}$- [*900*] *900*)
where
  Inf-fin = *fold1 inf*

definition
  Sup-fin :: $'a$ *set* $\Rightarrow$ $'a$ ($\bigsqcup_{fin}$- [*900*] *900*)
where

Sup-fin = *fold1 sup*

lemma *Inf-le-Sup* [*simp*]: $\llbracket$ *finite A*; $A \neq \{\}$ $\rrbracket \Longrightarrow \bigsqcap_{fin} A \leq \bigsqcup_{fin} A$
apply(*unfold Sup-fin-def Inf-fin-def*)
apply(*subgoal-tac EX a. a*:*A*)
prefer *2* **apply** *blast*
apply(*erule exE*)
apply(*rule order-trans*)
apply(*erule* (*2*) *ACIfSL.fold1-belowI*[*OF ACIfSL-inf*])
apply(*erule* (*2*) *ACIfSL.fold1-belowI*[*OF ACIfSL-sup*])
done

lemma *sup-Inf-absorb* [*simp*]:
$\llbracket$ *finite A*; $A \neq \{\}$; $a \in A$ $\rrbracket \Longrightarrow$ (*sup a* ($\bigsqcap_{fin} A$)) = *a*
apply(*subst sup-commute*)
apply(*simp add*: *Inf-fin-def sup-absorb2 ACIfSL.fold1-belowI* [*OF ACIfSL-inf*])
done

lemma *inf-Sup-absorb* [*simp*]:
$\llbracket$ *finite A*; $A \neq \{\}$; $a \in A$ $\rrbracket \Longrightarrow$ (*inf a* ($\bigsqcup_{fin} A$)) = *a*
by(*simp add*: *Sup-fin-def inf-absorb1 ACIfSL.fold1-belowI* [*OF ACIfSL-sup*])

end

context *distrib-lattice*
begin

lemma *sup-Inf1-distrib*:
finite A $\Longrightarrow A \neq \{\} \Longrightarrow$ *sup x* ($\bigsqcap_{fin} A$) = $\bigsqcap_{fin}$\{*sup x a*|*a*. $a \in A$\}
apply(*simp add*: *Inf-fin-def image-def*
ACIf.hom-fold1-commute[*OF ACIf-inf*, **where** *h*=*sup x*, *OF sup-inf-distrib1*])
apply(*rule arg-cong*, *blast*)
done

lemma *sup-Inf2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows *sup* ($\bigsqcap_{fin} A$) ($\bigsqcap_{fin} B$) = $\bigsqcap_{fin}$\{*sup a b*|*a b*. $a \in A \wedge b \in B$\}
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by (*simp add*: *sup-Inf1-distrib* [*OF B*] *fold1-singleton-def* [*OF Inf-fin-def*])
next
case (*insert x A*)
have *finB*: *finite* \{*sup x b* |*b*. $b \in B$\}
by(*rule finite-surj*[**where** *f* = *sup x*, *OF B*(*1*)], *auto*)
have *finAB*: *finite* \{*sup a b* |*a b*. $a \in A \wedge b \in B$\}
proof −
have \{*sup a b* |*a b*. $a \in A \wedge b \in B$\} = (*UN a*:*A*. *UN b*:*B*. \{*sup a b*\})
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))

qed
have *ne*: $\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B* **by** *blast*
have *sup* $(\bigsqcap_{fin}(insert\ x\ A))\ (\bigsqcap_{fin}B) = sup\ (inf\ x\ (\bigsqcap_{fin}A))\ (\bigsqcap_{fin}B)$
using *insert*
by(*simp add*:*ACIf.fold1-insert-idem-def*[*OF ACIf-inf Inf-fin-def*])
also have $\ldots = inf\ (sup\ x\ (\bigsqcap_{fin}B))\ (sup\ (\bigsqcap_{fin}A)\ (\bigsqcap_{fin}B))$ **by**(*rule sup-inf-distrib2*)
also have $\ldots = inf\ (\bigsqcap_{fin}\{sup\ x\ b|b.\ b \in B\})\ (\bigsqcap_{fin}\{sup\ a\ b|a\ b.\ a \in A \wedge b \in B\})$
using *insert* **by**(*simp add*:*sup-Inf1-distrib*[*OF B*])
also have $\ldots = \bigsqcap_{fin}(\{sup\ x\ b\ |b.\ b \in B\} \cup \{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\})$
(**is** - $= \bigsqcap_{fin}?M$)
using *B insert*
by (*simp add*: *Inf-fin-def ACIf.fold1-Un2*[*OF ACIf-inf finB - finAB ne*])
also have $?M = \{sup\ a\ b\ |a\ b.\ a \in insert\ x\ A \wedge b \in B\}$
by *blast*
finally show *?case* **.**
qed

lemma *inf-Sup1-distrib*:
finite $A \Longrightarrow A \neq \{\} \Longrightarrow inf\ x\ (\bigsqcup_{fin}A) = \bigsqcup_{fin}\{inf\ x\ a|a.\ a \in A\}$
apply (*simp add*: *Sup-fin-def image-def*
ACIf.hom-fold1-commute[*OF ACIf-sup*, **where** *h=inf x*, *OF inf-sup-distrib1*])
apply (*rule arg-cong*, *blast*)
done

lemma *inf-Sup2-distrib*:
assumes *A*: *finite* $A\ A \neq \{\}$ **and** *B*: *finite* $B\ B \neq \{\}$
shows $inf\ (\bigsqcup_{fin}A)\ (\bigsqcup_{fin}B) = \bigsqcup_{fin}\{inf\ a\ b|a\ b.\ a \in A \wedge b \in B\}$
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by(*simp add*: *inf-Sup1-distrib* [*OF B*] *fold1-singleton-def* [*OF Sup-fin-def*])
next
case (*insert x A*)
have *finB*: *finite* $\{inf\ x\ b\ |b.\ b \in B\}$
by(*rule finite-surj*[**where** $f = \%b.\ inf\ x\ b$, *OF B*(*1*)], *auto*)
have *finAB*: *finite* $\{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\}$
proof −
have $\{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} = (UN\ a{:}A.\ UN\ b{:}B.\ \{inf\ a\ b\})$
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: $\{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B* **by** *blast*
have $inf\ (\bigsqcup_{fin}(insert\ x\ A))\ (\bigsqcup_{fin}B) = inf\ (sup\ x\ (\bigsqcup_{fin}A))\ (\bigsqcup_{fin}B)$
using *insert* **by** (*simp add*: *ACIf.fold1-insert-idem-def* [*OF ACIf-sup Sup-fin-def*])
also have $\ldots = sup\ (inf\ x\ (\bigsqcup_{fin}B))\ (inf\ (\bigsqcup_{fin}A)\ (\bigsqcup_{fin}B))$ **by**(*rule inf-sup-distrib2*)
also have $\ldots = sup\ (\bigsqcup_{fin}\{inf\ x\ b|b.\ b \in B\})\ (\bigsqcup_{fin}\{inf\ a\ b|a\ b.\ a \in A \wedge b \in B\})$
using *insert* **by**(*simp add*:*inf-Sup1-distrib*[*OF B*])
also have $\ldots = \bigsqcup_{fin}(\{inf\ x\ b\ |b.\ b \in B\} \cup \{inf\ a\ b\ |a\ b.\ a \in A \wedge b \in B\})$

(**is** - = $\bigsqcup_{fin}$ *?M*)
using *B insert*
by (*simp add*: *Sup-fin-def ACIf.fold1-Un2*[*OF ACIf-sup finB - finAB ne*])
also have *?M* = {*inf a b* |*a b*. *a* ∈ *insert x A* ∧ *b* ∈ *B*}
by *blast*
finally show *?case* .
qed

end

context *complete-lattice*
begin

Coincidence on finite sets in complete lattices:

lemma *Inf-fin-Inf*:
finite A ⟹ *A* ≠ {} ⟹ $\bigsqcap_{fin}$*A* = *Inf A*
unfolding *Inf-fin-def* **by** (*induct A set*: *finite*)
(*simp-all add*: *Inf-insert-simp ACIf.fold1-insert-idem* [*OF ACIf-inf*])

lemma *Sup-fin-Sup*:
finite A ⟹ *A* ≠ {} ⟹ $\bigsqcup_{fin}$*A* = *Sup A*
unfolding *Sup-fin-def* **by** (*induct A set*: *finite*)
(*simp-all add*: *Sup-insert-simp ACIf.fold1-insert-idem* [*OF ACIf-sup*])

end

22.6.5 Fold1 in linear orders with *min* and *max*

As an application of *fold1* we define minimum and maximum in (not necessarily complete!) linear orders over (non-empty) sets by means of *fold1*.

context *linorder*
begin

definition
Min :: *'a set* ⇒ *'a*
where
Min = *fold1 min*

definition
Max :: *'a set* ⇒ *'a*
where
Max = *fold1 max*

end context *linorder* **begin**

recall: *min* and *max* behave like *inf* and *sup*

lemma *ACIf-min*: *ACIf min*
by (*rule lower-semilattice.ACIf-inf*,

rule lattice.axioms,
rule distrib-lattice.axioms,
rule distrib-lattice-min-max)

lemma *ACf-min*: *ACf min*
by (*rule lower-semilattice.ACf-inf*,
rule lattice.axioms,
rule distrib-lattice.axioms,
rule distrib-lattice-min-max)

lemma *ACIfSL-min*: *ACIfSL* (*op* $\leq$) (*op* $<$) *min*
by (*rule lower-semilattice.ACIfSL-inf*,
rule lattice.axioms,
rule distrib-lattice.axioms,
rule distrib-lattice-min-max)

lemma *ACIfSLlin-min*: *ACIfSLlin* (*op* $\leq$) (*op* $<$) *min*
by (*rule ACIfSLlin.intro*,
rule lower-semilattice.ACIfSL-inf,
rule lattice.axioms,
rule distrib-lattice.axioms,
rule distrib-lattice-min-max)
(*unfold-locales*, *simp add*: *min-def*)

lemma *ACIf-max*: *ACIf max*
by (*rule upper-semilattice.ACIf-sup*,
rule lattice.axioms,
rule distrib-lattice.axioms,
rule distrib-lattice-min-max)

lemma *ACf-max*: *ACf max*
by (*rule upper-semilattice.ACf-sup*,
rule lattice.axioms,
rule distrib-lattice.axioms,
rule distrib-lattice-min-max)

lemma *ACIfSL-max*: *ACIfSL* ($\lambda x\ y.\ y \leq x$) ($\lambda x\ y.\ y < x$) *max*
by (*rule upper-semilattice.ACIfSL-sup*,
rule lattice.axioms,
rule distrib-lattice.axioms,
rule distrib-lattice-min-max)

lemma *ACIfSLlin-max*: *ACIfSLlin* ($\lambda x\ y.\ y \leq x$) ($\lambda x\ y.\ y < x$) *max*
by (*rule ACIfSLlin.intro*,
rule upper-semilattice.ACIfSL-sup,
rule lattice.axioms,
rule distrib-lattice.axioms,
rule distrib-lattice-min-max)
(*unfold-locales*, *simp add*: *max-def*)

lemmas *Min-singleton* [*simp*] = *fold1-singleton-def* [*OF Min-def*]
lemmas *Max-singleton* [*simp*] = *fold1-singleton-def* [*OF Max-def*]
lemmas *Min-insert* [*simp*] = *ACIf.fold1-insert-idem-def* [*OF ACIf-min Min-def*]
lemmas *Max-insert* [*simp*] = *ACIf.fold1-insert-idem-def* [*OF ACIf-max Max-def*]

lemma *Min-in* [*simp*]:
shows $finite\ A \Longrightarrow A \neq \{\} \Longrightarrow Min\ A \in A$
using *ACf.fold1-in* [*OF ACf-min*]
by (*fastsimp simp*: *Min-def min-def*)

lemma *Max-in* [*simp*]:
shows $finite\ A \Longrightarrow A \neq \{\} \Longrightarrow Max\ A \in A$
using *ACf.fold1-in* [*OF ACf-max*]
by (*fastsimp simp*: *Max-def max-def*)

lemma *Min-antimono*: $[\![\ M \subseteq N;\ M \neq \{\};\ finite\ N\ ]\!] \Longrightarrow Min\ N \leq Min\ M$
by (*simp add*: *Min-def ACIfSLlin.fold1-antimono* [*OF ACIfSLlin-min*])

lemma *Max-mono*: $[\![\ M \subseteq N;\ M \neq \{\};\ finite\ N\ ]\!] \Longrightarrow Max\ M \leq Max\ N$
by (*simp add*: *Max-def ACIfSLlin.fold1-antimono* [*OF ACIfSLlin-max*])

lemma *Min-le* [*simp*]: $[\![\ finite\ A;\ A \neq \{\};\ x \in A\ ]\!] \Longrightarrow Min\ A \leq x$
by (*simp add*: *Min-def ACIfSL.fold1-belowI* [*OF ACIfSL-min*])

lemma *Max-ge* [*simp*]: $[\![\ finite\ A;\ A \neq \{\};\ x \in A\ ]\!] \Longrightarrow x \leq Max\ A$
by (*simp add*: *Max-def ACIfSL.fold1-belowI* [*OF ACIfSL-max*])

lemma *Min-ge-iff* [*simp*,*noatp*]:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow x \leq Min\ A \longleftrightarrow (\forall a \in A.\ x \leq a)$
by (*simp add*: *Min-def ACIfSL.below-fold1-iff* [*OF ACIfSL-min*])

lemma *Max-le-iff* [*simp*,*noatp*]:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Max\ A \leq x \longleftrightarrow (\forall a \in A.\ a \leq x)$
by (*simp add*: *Max-def ACIfSL.below-fold1-iff* [*OF ACIfSL-max*])

lemma *Min-gr-iff* [*simp*,*noatp*]:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow x < Min\ A \longleftrightarrow (\forall a \in A.\ x < a)$
by (*simp add*: *Min-def ACIfSLlin.strict-below-fold1-iff* [*OF ACIfSLlin-min*])

lemma *Max-less-iff* [*simp*,*noatp*]:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Max\ A < x \longleftrightarrow (\forall a \in A.\ a < x)$
by (*simp add*: *Max-def ACIfSLlin.strict-below-fold1-iff* [*OF ACIfSLlin-max*])

lemma *Min-le-iff* [*noatp*]:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Min\ A \leq x \longleftrightarrow (\exists a \in A.\ a \leq x)$
by (*simp add*: *Min-def ACIfSLlin.fold1-below-iff* [*OF ACIfSLlin-min*])

lemma *Max-ge-iff* [*noatp*]:

$\llbracket$ *finite* A; $A \neq \{\}$ $\rrbracket \Longrightarrow x \leq$ *Max* $A \longleftrightarrow (\exists\, a{\in}A.\ x \leq a)$
by (*simp add*: *Max-def ACIfSLlin.fold1-below-iff* [*OF ACIfSLlin-max*])

lemma *Min-less-iff* [*noatp*]:
$\llbracket$ *finite* A; $A \neq \{\}$ $\rrbracket \Longrightarrow$ *Min* $A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
by (*simp add*: *Min-def ACIfSLlin.fold1-strict-below-iff* [*OF ACIfSLlin-min*])

lemma *Max-gr-iff* [*noatp*]:
$\llbracket$ *finite* A; $A \neq \{\}$ $\rrbracket \Longrightarrow x <$ *Max* $A \longleftrightarrow (\exists\, a{\in}A.\ x < a)$
by (*simp add*: *Max-def ACIfSLlin.fold1-strict-below-iff* [*OF ACIfSLlin-max*])

lemma *Min-Un*: $\llbracket$*finite* A; $A \neq \{\}$; *finite* B; $B \neq \{\}\rrbracket$
$\Longrightarrow$ *Min* $(A \cup B) =$ *min* (*Min* A) (*Min* B)
by (*simp add*: *Min-def ACIf.fold1-Un2* [*OF ACIf-min*])

lemma *Max-Un*: $\llbracket$*finite* A; $A \neq \{\}$; *finite* B; $B \neq \{\}\rrbracket$
$\Longrightarrow$ *Max* $(A \cup B) =$ *max* (*Max* A) (*Max* B)
by (*simp add*: *Max-def ACIf.fold1-Un2* [*OF ACIf-max*])

lemma *hom-Min-commute*:
$(\bigwedge x\ y.\ h\ (min\ x\ y) = min\ (h\ x)\ (h\ y))$
$\Longrightarrow$ *finite* $N \Longrightarrow N \neq \{\} \Longrightarrow h$ (*Min* N) $=$ *Min* (h ‘ N)
by (*simp add*: *Min-def ACIf.hom-fold1-commute* [*OF ACIf-min*])

lemma *hom-Max-commute*:
$(\bigwedge x\ y.\ h\ (max\ x\ y) = max\ (h\ x)\ (h\ y))$
$\Longrightarrow$ *finite* $N \Longrightarrow N \neq \{\} \Longrightarrow h$ (*Max* N) $=$ *Max* (h ‘ N)
by (*simp add*: *Max-def ACIf.hom-fold1-commute* [*OF ACIf-max*])

end

context *ordered-ab-semigroup-add*
begin

lemma *add-Min-commute*:
fixes k
assumes *finite* N **and** $N \neq \{\}$
shows $k +$ *Min* $N =$ *Min* $\{k + m \mid m.\ m \in N\}$
proof −
have $\bigwedge x\ y.\ k + min\ x\ y = min\ (k + x)\ (k + y)$
by (*simp add*: *min-def not-le*)
(*blast intro*: *antisym less-imp-le add-left-mono*)
with *assms* **show** *?thesis*
using *hom-Min-commute* [*of plus k N*]
by *simp* (*blast intro*: *arg-cong* [**where** $f =$ *Min*])
qed

lemma *add-Max-commute*:
fixes k

assumes *finite N* **and** $N \neq \{\}$
shows $k + Max\ N = Max\ \{k + m \mid m.\ m \in N\}$
proof –
have $\bigwedge x\ y.\ k + max\ x\ y = max\ (k + x)\ (k + y)$
by (*simp add*: *max-def not-le*)
(*blast intro*: *antisym less-imp-le add-left-mono*)
with *assms* **show** *?thesis*
using *hom-Max-commute* [*of plus k N*]
by *simp* (*blast intro*: *arg-cong* [**where** $f = Max$])
qed

end

22.7 Class *finite* and code generation

lemma *finite-code* [*code func*]:
finite {} $\longleftrightarrow$ *True*
finite (*insert a A*) $\longleftrightarrow$ *finite A*
by *auto*

lemma *card-code* [*code func*]:
card {} = *0*
card (*insert a A*) =
(*if finite A then Suc* (*card* $(A - \{a\})$) *else card* (*insert a A*))
by (*auto simp add*: *card-insert*)

setup ⟪ *Sign.add-path finite* ⟫ — FIXME: name tweaking
class *finite* (**attach** *UNIV*) = *type* +
fixes *itself* :: *'a itself*
assumes *finite-UNIV*: *finite* (*UNIV* :: *'a set*)
setup ⟪ *Sign.parent-path* ⟫
hide *const finite*

lemma *finite* [*simp*]: *finite* (*A* :: *'a::finite set*)
by (*rule finite-subset* [*OF subset-UNIV finite-UNIV*])

lemma *univ-unit* [*noatp*]:
UNIV = {()} **by** *auto*

instance *unit* :: *finite*
Finite-Set.itself ≡ *TYPE*(*unit*)
proof
have *finite* {()} **by** *simp*
also note *univ-unit* [*symmetric*]
finally show *finite* (*UNIV* :: *unit set*) **.**
qed

lemmas [*code func*] = *univ-unit*

lemma *univ-bool* [*noatp*]:
  UNIV = {*False*, *True*} **by** *auto*

instance *bool* :: *finite*
  itself ≡ *TYPE*(*bool*)
proof
  have *finite* {*False*, *True*} **by** *simp*
  also note *univ-bool* [*symmetric*]
  finally show *finite* (*UNIV* :: *bool set*) **.**
qed

lemmas [*code func*] = *univ-bool*

instance ∗ :: (*finite*, *finite*) *finite*
  itself ≡ *TYPE*(′*a*::*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

lemma *univ-prod* [*noatp*, *code func*]:
  UNIV = (*UNIV* :: ′*a*::*finite set*) × (*UNIV* :: ′*b*::*finite set*)
  unfolding *UNIV-Times-UNIV* **..**

instance + :: (*finite*, *finite*) *finite*
  itself ≡ *TYPE*(′*a*::*finite* + ′*b*::*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

lemma *univ-sum* [*noatp*, *code func*]:
  UNIV = (*UNIV* :: ′*a*::*finite set*) <+> (*UNIV* :: ′*b*::*finite set*)
  unfolding *UNIV-Plus-UNIV* **..**

instance *set* :: (*finite*) *finite*
  itself ≡ *TYPE*(′*a*::*finite set*)
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 *univ-set* [*noatp*, *code func*]:
UNIV = *Pow* (*UNIV* :: *'a::finite set*) **unfolding** *Pow-UNIV* **..**

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*
itself ≡ *TYPE*(*'a::finite* ⇒ *'b::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*)
show *inj ?graph* **by** (*rule inj-graph*)
qed
qed

hide (**open**) *const itself*

22.8 Equality and order on functions

instance *fun* :: (*finite*, *eq*) *eq* **..**

lemma *eq-fun* [*code func*]:
fixes *f g* :: *'a::finite* ⇒ *'b::eq*
shows *f* = *g* ⟷ (∀ *x*∈*UNIV*. *f x* = *g x*)
unfolding *expand-fun-eq* **by** *auto*

lemma *order-fun* [*code func*]:
fixes *f g* :: *'a::finite* ⇒ *'b::order*
shows *f* ≤ *g* ⟷ (∀ *x*∈*UNIV*. *f x* ≤ *g x*)
and *f* < *g* ⟷ *f* ≤ *g* ∧ (∃ *x*∈*UNIV*. *f x* ≠ *g x*)
by (*auto simp add*: *expand-fun-eq le-fun-def less-fun-def order-less-le*)

end

23 Datatype: Analogues of the Cartesian Product and Disjoint Sum for Datatypes

theory *Datatype*
imports *Finite-Set*
uses *Tools/datatype-codegen.ML*
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*] => ('*a*, '*b*) *dtree*
In0 :: ('*a*, '*b*) *dtree* => ('*a*, '*b*) *dtree*
In1 :: ('*a*, '*b*) *dtree* => ('*a*, '*b*) *dtree*
Lim :: ('*b* => ('*a*, '*b*) *dtree*) => ('*a*, '*b*) *dtree*

ntrunc :: [*nat*, ('*a*, '*b*) *dtree*] => ('*a*, '*b*) *dtree*

uprod :: [('*a*, '*b*) *dtree set*, ('*a*, '*b*) *dtree set*]=> ('*a*, '*b*) *dtree set*
usum :: [('*a*, '*b*) *dtree set*, ('*a*, '*b*) *dtree set*]=> ('*a*, '*b*) *dtree set*

Split :: [[('*a*, '*b*) *dtree*, ('*a*, '*b*) *dtree*]=>'*c*, ('*a*, '*b*) *dtree*] => '*c*
Case :: [[('*a*, '*b*) *dtree*]=>'*c*, [('*a*, '*b*) *dtree*]=>'*c*, ('*a*, '*b*) *dtree*] => '*c*

dprod :: [(('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*, (('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*]
=> (('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*
dsum :: [(('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*, (('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*]
=> (('*a*, '*b*) *dtree* ∗ ('*a*, '*b*) *dtree*)*set*

defs

Push-Node-def: *Push-Node* == (%*n x*. *Abs-Node* (*apfst* (*Push n*) (*Rep-Node* *x*)))

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*, *code*]: *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

23.1 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N* ≠ *Atom*(*a*)
apply (*simp add*: *Atom-def Scons-def Push-Node-def One-nat-def*)
apply (*blast intro*: *Node-K0-I Rep-Node* [*THEN Node-Push-I*]
        dest!: *Abs-Node-inj*
        elim!: *apfst-convE sym* [*THEN Push-neq-K0*])
done

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*, *standard*]

lemma *inj-Atom*: *inj*(*Atom*)

apply (*simp add*: *Atom-def*)
apply (*blast intro*!: *inj-onI Node-K0-I dest*!: *Abs-Node-inj*)
done
lemmas *Atom-inject* = *inj-Atom* [*THEN injD*, *standard*]

lemma *Atom-Atom-eq* [*iff*]: (*Atom*(*a*)=*Atom*(*b*)) = (*a*=*b*)
by (*blast dest*!: *Atom-inject*)

lemma *inj-Leaf*: *inj*(*Leaf*)
apply (*simp add*: *Leaf-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inl-inject*])
done

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*, *standard*]

lemma *inj-Numb*: *inj*(*Numb*)
apply (*simp add*: *Numb-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inr-inject*])
done

lemmas *Numb-inject* [*dest*!] = *inj-Numb* [*THEN injD*, *standard*]

lemma *Push-Node-inject*:
  [| *Push-Node i m* =*Push-Node j n*; [| *i*=*j*; *m*=*n* |] ==> *P*
   |] ==> *P*
apply (*simp add*: *Push-Node-def*)
apply (*erule Abs-Node-inj* [*THEN apfst-convE*])
apply (*rule Rep-Node* [*THEN Node-Push-I*])+
apply (*erule sym* [*THEN apfst-convE*])
apply (*blast intro*: *Rep-Node-inject* [*THEN iffD1*] *trans sym elim*!: *Push-inject*)
done

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M' N'* ==> *M*<=*M'*
apply (*simp add*: *Scons-def One-nat-def*)
apply (*blast dest*!: *Push-Node-inject*)
done

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M' N'* ==> *N*<=*N'*
apply (*simp add*: *Scons-def One-nat-def*)
apply (*blast dest*!: *Push-Node-inject*)
done

lemma *Scons-inject1*: *Scons M N = Scons M′ N′ ==> M=M′*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma1*)
done

lemma *Scons-inject2*: *Scons M N = Scons M′ N′ ==> N=N′*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma2*)
done

lemma *Scons-inject*:
  [| *Scons M N = Scons M′ N′*; [| *M=M′*; *N=N′* |] ==> *P* |] ==> *P*
by (*iprover dest*: *Scons-inject1 Scons-inject2*)

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N = Scons M′ N′*) = (*M=M′* & *N=N′*)
by (*blast elim*!: *Scons-inject*)

lemma *Scons-not-Leaf* [*iff*]: *Scons M N* ≠ *Leaf*(*a*)
by (*simp add*: *Leaf-def o-def Scons-not-Atom*)

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*, *standard*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)
by (*simp add*: *Numb-def o-def Scons-not-Atom*)

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*, *standard*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
by (*simp add*: *Leaf-def Numb-def*)

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*, *standard*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr 0*, *x*)) = *0*
by (*simp add*: *ndepth-def Node-K0-I* [*THEN Abs-Node-inverse*] *Least-equality*)

lemma *ndepth-Push-Node-aux*:

nat-case (Inr (Suc i)) f k = Inr 0 --> Suc(LEAST x. f x = Inr 0) <= k
apply (*induct-tac k, auto*)
apply (*erule Least-le*)
done

lemma *ndepth-Push-Node*:
ndepth (Push-Node (Inr (Suc i)) n) = Suc(ndepth(n))
apply (*insert Rep-Node [of n, unfolded Node-def]*)
apply (*auto simp add: ndepth-def Push-Node-def*
Rep-Node [THEN Node-Push-I, THEN Abs-Node-inverse])
apply (*rule Least-equality*)
apply (*auto simp add: Push-def ndepth-Push-Node-aux*)
apply (*erule LeastI*)
done

lemma *ntrunc-0 [simp]: ntrunc 0 M = {}*
by (*simp add: ntrunc-def*)

lemma *ntrunc-Atom [simp]: ntrunc (Suc k) (Atom a) = Atom(a)*
by (*auto simp add: Atom-def ntrunc-def ndepth-K0*)

lemma *ntrunc-Leaf [simp]: ntrunc (Suc k) (Leaf a) = Leaf(a)*
by (*simp add: Leaf-def o-def ntrunc-Atom*)

lemma *ntrunc-Numb [simp]: ntrunc (Suc k) (Numb i) = Numb(i)*
by (*simp add: Numb-def o-def ntrunc-Atom*)

lemma *ntrunc-Scons [simp]*:
ntrunc (Suc k) (Scons M N) = Scons (ntrunc k M) (ntrunc k N)
by (*auto simp add: Scons-def ntrunc-def One-nat-def ndepth-Push-Node*)

lemma *ntrunc-one-In0 [simp]: ntrunc (Suc 0) (In0 M) = {}*
apply (*simp add: In0-def*)
apply (*simp add: Scons-def*)
done

lemma *ntrunc-In0 [simp]: ntrunc (Suc(Suc k)) (In0 M) = In0 (ntrunc (Suc k) M)*
by (*simp add: In0-def*)

lemma *ntrunc-one-In1 [simp]: ntrunc (Suc 0) (In1 M) = {}*
apply (*simp add: In1-def*)

apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
by (*simp add*: *In1-def*)

23.2 Set Constructions

lemma *uprodI* [*intro*!]: [| *M*:*A*; *N*:*B* |] ==> *Scons M N* : *uprod A B*
by (*simp add*: *uprod-def*)

lemma *uprodE* [*elim*!]:
[| *c* : *uprod A B*;
!!*x y*. [| *x*:*A*; *y*:*B*; *c* = *Scons x y* |] ==> *P*
|] ==> *P*
by (*auto simp add*: *uprod-def*)

lemma *uprodE2*: [| *Scons M N* : *uprod A B*; [| *M*:*A*; *N*:*B* |] ==> *P* |] ==> *P*
by (*auto simp add*: *uprod-def*)

lemma *usum-In0I* [*intro*]: *M*:*A* ==> *In0*(*M*) : *usum A B*
by (*simp add*: *usum-def*)

lemma *usum-In1I* [*intro*]: *N*:*B* ==> *In1*(*N*) : *usum A B*
by (*simp add*: *usum-def*)

lemma *usumE* [*elim*!]:
[| *u* : *usum A B*;
!!*x*. [| *x*:*A*; *u*=*In0*(*x*) |] ==> *P*;
!!*y*. [| *y*:*B*; *u*=*In1*(*y*) |] ==> *P*
|] ==> *P*
by (*auto simp add*: *usum-def*)

lemma *In0-not-In1* [*iff*]: *In0*(*M*) ≠ *In1*(*N*)
by (*auto simp add*: *In0-def In1-def One-nat-def*)

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*, *standard*]

lemma *In0-inject*: *In0*(*M*) = *In0*(*N*) ==> *M*=*N*

by (*simp add*: *In0-def*)

lemma *In1-inject*: *In1*(*M*) = *In1*(*N*) ==> *M*=*N*
by (*simp add*: *In1-def*)

lemma *In0-eq* [*iff*]: (*In0 M* = *In0 N*) = (*M*=*N*)
by (*blast dest*!: *In0-inject*)

lemma *In1-eq* [*iff*]: (*In1 M* = *In1 N*) = (*M*=*N*)
by (*blast dest*!: *In1-inject*)

lemma *inj-In0*: *inj In0*
by (*blast intro*!: *inj-onI*)

lemma *inj-In1*: *inj In1*
by (*blast intro*!: *inj-onI*)

lemma *Lim-inject*: *Lim f* = *Lim g* ==> *f* = *g*
apply (*simp add*: *Lim-def*)
apply (*rule ext*)
apply (*blast elim*!: *Push-Node-inject*)
done

lemma *ntrunc-subsetI*: *ntrunc k M* <= *M*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-subsetD*: (!!*k*. *ntrunc k M* <= *N*) ==> *M*<=*N*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-equality*: (!!*k*. *ntrunc k M* = *ntrunc k N*) ==> *M*=*N*
apply (*rule equalityI*)
apply (*rule-tac* [!] *ntrunc-subsetD*)
apply (*rule-tac* [!] *ntrunc-subsetI* [*THEN* [*2*] *subset-trans*], *auto*)
done

lemma *ntrunc-o-equality*:
  [| !!*k*. (*ntrunc*(*k*) *o h1*) = (*ntrunc*(*k*) *o h2*) |] ==> *h1*=*h2*
apply (*rule ntrunc-equality* [*THEN ext*])
apply (*simp add*: *expand-fun-eq*)
done

lemma *uprod-mono*: $[| A<=A';\ \ B<=B' |] ==> uprod\ A\ B <= uprod\ A'\ B'$
by (*simp add*: *uprod-def*, *blast*)

lemma *usum-mono*: $[| A<=A';\ \ B<=B' |] ==> usum\ A\ B <= usum\ A'\ B'$
by (*simp add*: *usum-def*, *blast*)

lemma *Scons-mono*: $[| M<=M';\ \ N<=N' |] ==> Scons\ M\ N <= Scons\ M'\ N'$
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-mono*: $M<=N ==> In0(M) <= In0(N)$
by (*simp add*: *In0-def subset-refl Scons-mono*)

lemma *In1-mono*: $M<=N ==> In1(M) <= In1(N)$
by (*simp add*: *In1-def subset-refl Scons-mono*)

lemma *Split* [*simp*]: $Split\ c\ (Scons\ M\ N) = c\ M\ N$
by (*simp add*: *Split-def*)

lemma *Case-In0* [*simp*]: $Case\ c\ d\ (In0\ M) = c(M)$
by (*simp add*: *Case-def*)

lemma *Case-In1* [*simp*]: $Case\ c\ d\ (In1\ N) = d(N)$
by (*simp add*: *Case-def*)

lemma *ntrunc-UN1*: $ntrunc\ k\ (UN\ x.\ f(x)) = (UN\ x.\ ntrunc\ k\ (f\ x))$
by (*simp add*: *ntrunc-def*, *blast*)

lemma *Scons-UN1-x*: $Scons\ (UN\ x.\ f\ x)\ M = (UN\ x.\ Scons\ (f\ x)\ M)$
by (*simp add*: *Scons-def*, *blast*)

lemma *Scons-UN1-y*: $Scons\ M\ (UN\ x.\ f\ x) = (UN\ x.\ Scons\ M\ (f\ x))$
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-UN1*: $In0(UN\ x.\ f(x)) = (UN\ x.\ In0(f(x)))$
by (*simp add*: *In0-def Scons-UN1-y*)

lemma *In1-UN1*: $In1(UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
by (*simp add*: *In1-def Scons-UN1-y*)

lemma *dprodI* [*intro*!]:
  [| $(M,M'):r$; $(N,N'):s$ |] ==> $(Scons\ M\ N,\ Scons\ M'\ N') : dprod\ r\ s$
by (*auto simp add*: *dprod-def*)

lemma *dprodE* [*elim*!]:
  [| $c : dprod\ r\ s$;
    !!$x\ y\ x'\ y'$. [| $(x,x') : r$; $(y,y') : s$;
          $c = (Scons\ x\ y,\ Scons\ x'\ y')$ |] ==> 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*, *standard*]

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*

hides popular names

hide (**open**) *type node item*
hide (**open**) *const Push Node Atom Leaf Numb Lim Split Case*

24 Datatypes

24.1 Representing sums

rep-datatype *sum*
distinct *Inl-not-Inr Inr-not-Inl*
inject *Inl-eq Inr-eq*
induction *sum-induct*

lemma *size-sum* [*code func*]:
size (*x* :: ′*a* + ′*b*) = *0* **by** (*cases x*) *auto*

lemma *sum-case-KK*[*simp*]: *sum-case* (%*x*. *a*) (%*x*. *a*) = (%*x*. *a*)
by (*rule ext*) (*simp split*: *sum.split*)

lemma *surjective-sum*: *sum-case* (%*x*::′*a*. *f* (*Inl x*)) (%*y*::′*b*. *f* (*Inr y*)) *s* = *f*(*s*)
apply (*rule-tac s* = *s* **in** *sumE*)
apply (*erule ssubst*)
apply (*rule sum.cases*(*1*))
apply (*erule ssubst*)
apply (*rule sum.cases*(*2*))

done

lemma *sum-case-weak-cong*: $s = t ==> sum\text{-}case\ f\ g\ s = sum\text{-}case\ f\ g\ t$
— Prevents simplification of f and g: much faster.
by *simp*

lemma *sum-case-inject*:
$sum\text{-}case\ f1\ f2 = sum\text{-}case\ g1\ g2 ==> (f1 = g1 ==> f2 = g2 ==> P) ==> P$
proof –
assume a: $sum\text{-}case\ f1\ f2 = sum\text{-}case\ g1\ g2$
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
$Suml :: ('a => 'c) => 'a + 'b => 'c$
$Suml == (\%f.\ sum\text{-}case\ f\ arbitrary)$

$Sumr :: ('b => 'c) => 'a + 'b => 'c$
$Sumr == sum\text{-}case\ arbitrary$

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*)

hide (**open**) *const Suml Sumr*

24.2 The option datatype

datatype *'a option = None | Some 'a*

lemma *not-None-eq* [*iff*]: $(x\ \tilde{}= None) = (EX\ y.\ x = Some\ y)$
by (*induct x*) *auto*

lemma *not-Some-eq* [*iff*]: $(ALL\ y.\ x\ \tilde{}= Some\ y) = (x = None)$
by (*induct x*) *auto*

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

lemma *option-caseE*:
  assumes *c*: (*case x of None => P | Some y => Q y*)
  obtains
    (*None*) *x = None* **and** *P*
  | (*Some*) *y* **where** *x = Some y* **and** *Q y*
  using *c* **by** (*cases x*) *simp-all*

lemma *insert-None-conv-UNIV*: *insert None* (*range Some*) = *UNIV*
  by (*rule set-ext*, *case-tac x*) *auto*

instance *option* :: (*finite*) *finite*
proof
  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

lemma *univ-option* [*noatp*, *code func*]:
  UNIV = *insert* (*None* :: *'a::finite option*) (*image Some UNIV*)
  unfolding *insert-None-conv-UNIV* **..**

24.2.1 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 ⟪ *change-claset* (*fn cs => cs 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*)

lemmas [*code func del*] = *option-map-def*

lemma *option-map-None* [*simp*, *code*]: *option-map f None* = *None*
by (*simp add*: *option-map-def*)

lemma *option-map-Some* [*simp*, *code*]: *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*)
by (*rule ext*) (*simp split*: *sum.split*)

24.2.2 Code generator setup

setup *DatatypeCodegen.setup*

definition
is-none :: $'a$ *option* $\Rightarrow$ *bool* **where**
is-none-none [*code post*, *symmetric*, *code inline*]: *is-none x* $\longleftrightarrow$ *x* = *None*

lemma *is-none-code* [*code*]:
shows *is-none None* $\longleftrightarrow$ *True*
and *is-none* (*Some x*) $\longleftrightarrow$ *False*
unfolding *is-none-none* [*symmetric*] **by** *simp-all*

hide (**open**) *const is-none*

code-type *option*
(*SML* - *option*)
(*OCaml* - *option*)
(*Haskell Maybe* -)

code-const *None* **and** *Some*
(*SML NONE* **and** *SOME*)
(*OCaml None* **and** *Some* -)

(*Haskell Nothing* **and** *Just*)

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

code-const *op* = :: *'a::eq option* ⇒ *'a option* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-reserved *SML*
option NONE SOME

code-reserved *OCaml*
option None Some

code-modulename *SML*
Datatype Nat

code-modulename *OCaml*
Datatype Nat

code-modulename *Haskell*
Datatype Nat

end

25 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *Finite-Set Relation*
begin

25.1 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* ∀ *x y*. (*x*, *y*) ∈ *r --> (y, x)* ∈ *r*)
apply *fast*
apply *fast*
done

25.2 Equivalence classes

lemma *equiv-class-subset*:
equiv A r ==> (*a*, *b*) ∈ *r ==> r*‘‘{*a*} ⊆ *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*) ∈ *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* ∈ *A ==> a* ∈ *r*‘‘{*a*}
by (*unfold equiv-def refl-def*) *blast*

lemma *subset-equiv-class*:
equiv A r ==> r‘‘{*b*} ⊆ *r*‘‘{*a*} *==> b* ∈ *A ==>* (*a*,*b*) ∈ *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* ∈ *A ==>* (*a*, *b*) ∈ *r*
by (*iprover intro*: *equalityD2 subset-equiv-class*)

lemma *equiv-class-nondisjoint*:
equiv A r ==> x ∈ (*r*‘‘{*a*} ∩ *r*‘‘{*b*}) *==>* (*a*, *b*) ∈ *r*
by (*unfold equiv-def trans-def sym-def*) *blast*

lemma *equiv-type*: *equiv A r ==> r* ⊆ *A* × *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*)

25.3 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 $A\ r$ ==> $X \in A//r$ ==> $Y \in A//r$ ==> $X = Y \mid (X \cap Y = \{\})$
apply (*unfold quotient-def*)
apply *clarify*
apply (*rule equiv-class-eq*)
 apply *assumption*
apply (*unfold equiv-def trans-def sym-def*)
apply *blast*
done

lemma *quotient-eqI*:
[|*equiv* $A\ r$; $X \in A//r$; $Y \in A//r$; $x \in X$; $y \in Y$; $(x,y) \in r$|] ==> $X = Y$
apply (*clarify elim*!: *quotientE*)
apply (*rule equiv-class-eq*, *assumption*)
apply (*unfold equiv-def sym-def trans-def*, *blast*)
done

lemma *quotient-eq-iff*:
[|*equiv* $A\ r$; $X \in A//r$; $Y \in A//r$; $x \in X$; $y \in Y$|] ==> $(X = Y) = ((x,y) \in$
$r)$
apply (*rule iffI*)
 prefer *2* **apply** (*blast del*: *equalityI intro*: *quotient-eqI*)
apply (*clarify elim*!: *quotientE*)

apply (*unfold equiv-def sym-def trans-def*, *blast*)
done

lemma *eq-equiv-class-iff2*:
$[\![$ *equiv A r*; $x \in A$; $y \in A$ $]\!] \Longrightarrow (\{x\}//r = \{y\}//r) = ((x,y) : r)$
by(*simp add*:*quotient-def eq-equiv-class-iff*)

lemma *quotient-empty* [*simp*]: $\{\}//r = \{\}$
by(*simp add*: *quotient-def*)

lemma *quotient-is-empty* [*iff*]: $(A//r = \{\}) = (A = \{\})$
by(*simp add*: *quotient-def*)

lemma *quotient-is-empty2* [*iff*]: $(\{\} = A//r) = (A = \{\})$
by(*simp add*: *quotient-def*)

lemma *singleton-quotient*: $\{x\}//r = \{r \text{ `` } \{x\}\}$
by(*simp add*:*quotient-def*)

lemma *quotient-diff1*:
$[\![$ *inj-on* $(\%a.\ \{a\}//r)\ A$; $a \in A$ $]\!] \Longrightarrow (A - \{a\})//r = A//r - \{a\}//r$
apply(*simp add*:*quotient-def inj-on-def*)
apply *blast*
done

25.4 Defining unary operations upon equivalence classes

A congruence-preserving function

locale *congruent* =
fixes *r* **and** *f*
assumes *congruent*: $(y,z) \in r ==> f\ y = f\ z$

abbreviation
RESPECTS :: $('a => 'b) => ('a * 'a)$ *set* => *bool*
(**infixr** *respects 80*) **where**
f respects r == *congruent r f*

lemma *UN-constant-eq*: $a \in A ==> \forall y \in A.\ f\ y = c ==> (\bigcup y \in A.\ f(y))=c$
— lemma required to prove *UN-equiv-class*
by *auto*

lemma *UN-equiv-class*:
equiv A r ==> *f respects r* ==> $a \in A$
==> $(\bigcup x \in r\text{``}\{a\}.\ f\ x) = f\ a$
— Conversion rule
apply (*rule equiv-class-self* [*THEN UN-constant-eq*], *assumption*+)

apply (*unfold equiv-def congruent-def sym-def*)
apply (*blast del*: *equalityI*)
done

lemma *UN-equiv-class-type*:
equiv A r ==> *f respects* r ==> $X \in A//r$ ==>
(!!x. $x \in A$ ==> $f\,x \in B$) ==> $(\bigcup x \in X.\ f\,x) \in B$
apply (*unfold quotient-def*)
apply *clarify*
apply (*subst UN-equiv-class*)
apply *auto*
done

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be !!y. $y \in A$ ==> $f\,y \in B$.

lemma *UN-equiv-class-inject*:
equiv A r ==> *f respects* r ==>
$(\bigcup x \in X.\ f\,x) = (\bigcup y \in Y.\ f\,y)$ ==> $X \in A//r$ ==> $Y \in A//r$
==> (!!x y. $x \in A$ ==> $y \in A$ ==> $f\,x = f\,y$ ==> $(x, y) \in r$)
==> $X = Y$
apply (*unfold quotient-def*)
apply *clarify*
apply (*rule equiv-class-eq*)
apply *assumption*
apply (*subgoal-tac* $f\,x = f\,xa$)
apply *blast*
apply (*erule box-equals*)
apply (*assumption* | *rule UN-equiv-class*)+
done

25.5 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* ==> (*y2*,*z2*) $\in$ *r2* ==> *f y1 y2* = *f z1 z2*

Abbreviation for the common case where the relations are identical

abbreviation
RESPECTS2:: ['a => 'a => 'b, ('a * 'a) *set*] => *bool*
(**infixr** *respects2 80*) **where**
f respects2 r == *congruent2 r r f*

lemma *congruent2-implies-congruent*:
equiv A r1 ==> *congruent2 r1 r2 f* ==> $a \in A$ ==> *congruent r2* (*f a*)
by (*unfold congruent-def congruent2-def equiv-def refl-def*) *blast*

lemma *congruent2-implies-congruent-UN*:
equiv A1 r1 ==> *equiv A2 r2* ==> *congruent2 r1 r2 f* ==> $a \in A2$ ==>
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 ==> *equiv A2 r2* ==> *congruent2 r1 r2 f* ==> $a1 \in A1$ ==> $a2 \in A2$
==> $(\bigcup x1 \in r1``\{a1\}. \bigcup x2 \in r2``\{a2\}. f\ x1\ x2) = f\ a1\ a2$
by (*simp add*: *UN-equiv-class congruent2-implies-congruent*
congruent2-implies-congruent-UN)

lemma *UN-equiv-class-type2*:
equiv A1 r1 ==> *equiv A2 r2* ==> *congruent2 r1 r2 f*
==> $X1 \in A1//r1$ ==> $X2 \in A2//r2$
==> (!!*x1 x2*. $x1 \in A1$ ==> $x2 \in A2$ ==> $f\ x1\ x2 \in B$)
==> $(\bigcup x1 \in X1. \bigcup x2 \in X2. f\ x1\ x2) \in B$
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*:
$(\bigcup (x1, x2) \in X. \bigcup (y1, y2) \in Y. A\ x1\ x2\ y1\ y2) =$
$(\bigcup x \in X. \bigcup y \in Y. (\lambda(x1, x2). (\lambda(y1, y2). A\ x1\ x2\ y1\ y2)\ y)\ x)$
— 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*. $w \in A2$ ==> $(y,z) \in r1$ ==> $f\ y\ w = f\ z\ w$)
==> (!!*y z w*. $w \in A1$ ==> $(y,z) \in r2$ ==> $f\ w\ y = f\ w\ z$)
==> *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*. $y \in A$ ==> $z \in A$ ==> *f y z* = *f z y*
    and *congt*: !!*y z w*. $w \in A$ ==> $(y,z) \in r$ ==> *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

25.6 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*: *finite A* ==> $r \subseteq A \times A$ ==> *finite* $(A//r)$
  — recall *equiv ?A ?r* $\Longrightarrow$ $?r \subseteq ?A \times ?A$
  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 ==> $r \subseteq A \times A$ ==> $X \in A//r$ ==> *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* ==> $\forall X \in A//r$. *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*:
$\llbracket$ *finite A*; *inj-on* $(\lambda x.\ \{x\}\ //\ r)$ *A* $\rrbracket \Longrightarrow$ *card*$(A//r)$ = *card A*
apply(*simp add*:*quotient-def*)
apply(*subst card-UN-disjoint*)
   apply *assumption*
  apply *simp*
 apply(*fastsimp simp add*:*inj-on-def*)

apply (*simp add*:*setsum-constant*)
done

end

26 IntDef: The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *IntDef*
imports *Equiv-Relations Nat*
begin

the equivalence relation underlying the integers

definition
$intrel :: ((nat \times nat) \times (nat \times nat))\ set$
where
$intrel = \{((x,\ y),\ (u,\ v)) \mid x\ y\ u\ v.\ x + v = u + y\ \}$

typedef (*Integ*)
$int = UNIV//intrel$
by (*auto simp add*: *quotient-def*)

instance *int* :: *zero*
Zero-int-def: $0 \equiv Abs\text{-}Integ\ (intrel\ ``\ \{(0,\ 0)\})$ **..**

instance *int* :: *one*
One-int-def: $1 \equiv Abs\text{-}Integ\ (intrel\ ``\ \{(1,\ 0)\})$ **..**

instance *int* :: *plus*
add-int-def: $z + w \equiv Abs\text{-}Integ$
$(\bigcup (x,\ y) \in Rep\text{-}Integ\ z.\ \bigcup (u,\ v) \in Rep\text{-}Integ\ w.$
$intrel\ ``\ \{(x + u,\ y + v)\})$ **..**

instance *int* :: *minus*
minus-int-def:
$-\ z \equiv Abs\text{-}Integ\ (\bigcup (x,\ y) \in Rep\text{-}Integ\ z.\ intrel\ ``\ \{(y,\ x)\})$
diff-int-def: $z - w \equiv z + (-w)$ **..**

instance *int* :: *times*
mult-int-def: $z * w \equiv\ Abs\text{-}Integ$
$(\bigcup (x,\ y) \in Rep\text{-}Integ\ z.\ \bigcup (u,v\ ) \in Rep\text{-}Integ\ w.$
$intrel\ ``\ \{(x*u + y*v,\ x*v + y*u)\})$ **..**

instance *int* :: *ord*
le-int-def:
$z \le w \equiv \exists\, x\ y\ u\ v.\ x+v \le u+y \wedge (x,\ y) \in Rep\text{-}Integ\ z \wedge (u,\ v) \in Rep\text{-}Integ\ w$
less-int-def: $z < w \equiv z \le w \wedge z \neq w$ **..**

lemmas [*code func del*] = *Zero-int-def One-int-def add-int-def*
minus-int-def mult-int-def le-int-def less-int-def

26.1 Construction of the Integers

lemma *intrel-iff* [*simp*]: $(((x,y),(u,v)) \in intrel) = (x+v = u+y)$
by (*simp add*: *intrel-def*)

lemma *equiv-intrel*: *equiv UNIV intrel*
by (*simp add*: *intrel-def equiv-def refl-def sym-def trans-def*)

Reduces equality of equivalence classes to the *intrel* relation: $(intrel \text{ `` } \{x\} = intrel \text{ `` } \{y\}) = ((x,\ y) \in intrel)$

lemmas *equiv-intrel-iff* [*simp*] = *eq-equiv-class-iff* [*OF equiv-intrel UNIV-I UNIV-I*]

All equivalence classes belong to set of representatives

lemma [*simp*]: $intrel\text{``}\{(x,y)\} \in Integ$
by (*auto simp add*: *Integ-def intrel-def quotient-def*)

Reduces equality on abstractions to equality on representatives: $[\![x \in Integ;\ y \in Integ]\!] \Longrightarrow (Abs\text{-}Integ\ x = Abs\text{-}Integ\ y) = (x = y)$

declare *Abs-Integ-inject* [*simp*,*noatp*] *Abs-Integ-inverse* [*simp*,*noatp*]

Case analysis on the representation of an integer as an equivalence class of pairs of naturals.

lemma *eq-Abs-Integ* [*case-names Abs-Integ*, *cases type*: *int*]:
  $(!!x\ y.\ z = Abs\text{-}Integ(intrel\text{``}\{(x,y)\}) ==> P) ==> P$
apply (*rule Abs-Integ-cases* [*of z*])
apply (*auto simp add*: *Integ-def quotient-def*)
done

26.2 Arithmetic Operations

lemma *minus*: $-\ Abs\text{-}Integ(intrel\text{``}\{(x,y)\}) = Abs\text{-}Integ(intrel \text{ `` } \{(y,x)\})$
proof –
  have $(\lambda(x,y).\ intrel\text{``}\{(y,x)\})$ *respects intrel*
    by (*simp add*: *congruent-def*)
  thus *?thesis*
    by (*simp add*: *minus-int-def UN-equiv-class* [*OF equiv-intrel*])
qed

lemma *add*:
  $Abs\text{-}Integ\ (intrel\text{``}\{(x,y)\}) + Abs\text{-}Integ\ (intrel\text{``}\{(u,v)\}) =$
  $Abs\text{-}Integ\ (intrel\text{``}\{(x+u,\ y+v)\})$
proof –
  have $(\lambda z\ w.\ (\lambda(x,y).\ (\lambda(u,v).\ intrel \text{ `` } \{(x+u,\ y+v)\})\ w)\ z)$
    respects2 intrel

```
    by (simp add: congruent2-def)
  thus ?thesis
    by (simp add: add-int-def UN-UN-split-split-eq
                  UN-equiv-class2 [OF equiv-intrel equiv-intrel])
qed
```

Congruence property for multiplication

```
lemma mult-congruent2:
     (%p1 p2. (%(x,y). (%(u,v). intrel``{(x*u + y*v, x*v + y*u)}) p2) p1)
      respects2 intrel
apply (rule equiv-intrel [THEN congruent2-commuteI])
 apply (force simp add: mult-ac, clarify)
apply (simp add: congruent-def mult-ac)
apply (rename-tac u v w x y z)
apply (subgoal-tac u*y + x*y = w*y + v*y  &  u*z + x*z = w*z + v*z)
apply (simp add: mult-ac)
apply (simp add: add-mult-distrib [symmetric])
done

lemma mult:
     Abs-Integ((intrel``{(x,y)})) * Abs-Integ((intrel``{(u,v)})) =
      Abs-Integ(intrel `` {(x*u + y*v, x*v + y*u)})
by (simp add: mult-int-def UN-UN-split-split-eq mult-congruent2
              UN-equiv-class2 [OF equiv-intrel equiv-intrel])
```

The integers form a *comm-ring-1*

```
instance int :: comm-ring-1
proof
  fix i j k :: int
  show (i + j) + k = i + (j + k)
    by (cases i, cases j, cases k) (simp add: add add-assoc)
  show i + j = j + i
    by (cases i, cases j) (simp add: add-ac add)
  show 0 + i = i
    by (cases i) (simp add: Zero-int-def add)
  show − i + i = 0
    by (cases i) (simp add: Zero-int-def minus add)
  show i − j = i + − j
    by (simp add: diff-int-def)
  show (i * j) * k = i * (j * k)
    by (cases i, cases j, cases k) (simp add: mult ring-simps)
  show i * j = j * i
    by (cases i, cases j) (simp add: mult ring-simps)
  show 1 * i = i
    by (cases i) (simp add: One-int-def mult)
  show (i + j) * k = i * k + j * k
    by (cases i, cases j, cases k) (simp add: add mult ring-simps)
  show 0 ≠ (1::int)
    by (simp add: Zero-int-def One-int-def)
```

qed

lemma *int-def*: *of-nat m* = *Abs-Integ* (*intrel* ‘‘ {(*m*, *0*)})
by (*induct m*, *simp-all add*: *Zero-int-def One-int-def add*)

26.3 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 *less*:
  (*Abs-Integ*(*intrel*‘‘{(*x*,*y*)}) < *Abs-Integ*(*intrel*‘‘{(*u*,*v*)})) = (*x*+*v* < *u*+*y*)
by (*simp add*: *less-int-def le order-less-le*)

instance *int* :: *linorder*
proof
  fix *i j k* :: *int*
  show (*i* < *j*) = (*i* ≤ *j* ∧ *i* ≠ *j*)
    by (*simp add*: *less-int-def*)
  show *i* ≤ *i*
    by (*cases i*) (*simp add*: *le*)
  show *i* ≤ *j* ⟹ *j* ≤ *k* ⟹ *i* ≤ *k*
    by (*cases i*, *cases j*, *cases k*) (*simp add*: *le*)
  show *i* ≤ *j* ⟹ *j* ≤ *i* ⟹ *i* = *j*
    by (*cases i*, *cases j*) (*simp add*: *le*)
  show *i* ≤ *j* ∨ *j* ≤ *i*
    by (*cases i*, *cases j*) (*simp add*: *le linorder-linear*)
qed

instance *int* :: *pordered-cancel-ab-semigroup-add*
proof
  fix *i j k* :: *int*
  show *i* ≤ *j* ⟹ *k* + *i* ≤ *k* + *j*
    by (*cases i*, *cases j*, *cases k*) (*simp add*: *le add*)
qed

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on k¿0

lemma *zmult-zless-mono2-lemma*:
    (*i*::*int*)<*j* ==> *0*<*k* ==> *of-nat k* ∗ *i* < *of-nat k* ∗ *j*
apply (*induct k*, *simp*)
apply (*simp add*: *left-distrib*)
apply (*case-tac k*=*0*)
apply (*simp-all add*: *add-strict-mono*)
done

lemma *zero-le-imp-eq-int*: (*0*::*int*) ≤ *k* ==> ∃ *n*. *k* = *of-nat n*
apply (*cases k*)

apply (*auto simp add*: *le add int-def Zero-int-def*)
apply (*rule-tac* $x=x-y$ **in** *exI*, *simp*)
done

lemma *zero-less-imp-eq-int*: $(0::int) < k ==> \exists n>0.\ k = of\text{-}nat\ n$
apply (*cases k*)
apply (*simp add*: *less int-def Zero-int-def*)
apply (*rule-tac* $x=x-y$ **in** *exI*, *simp*)
done

lemma *zmult-zless-mono2*: $[\mid i<j;\ \ (0::int) < k \mid] ==> k*i < k*j$
apply (*drule zero-less-imp-eq-int*)
apply (*auto simp add*: *zmult-zless-mono2-lemma*)
done

instance *int* :: *abs*
  zabs-def: $|i::int| \equiv if\ i < 0\ then\ -\ i\ else\ i$ **..**
instance *int* :: *sgn*
  zsgn-def: $sgn(i::int) \equiv (if\ i=0\ then\ 0\ else\ if\ 0<i\ then\ 1\ else\ -\ 1)$ **..**

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

The integers form an ordered integral domain

instance *int* :: *ordered-idom*
proof
  fix *i j k* :: *int*
  show $i < j \Longrightarrow 0 < k \Longrightarrow k * i < k * j$
    by (*rule zmult-zless-mono2*)
  show $|i| = (if\ i < 0\ then\ -i\ else\ i)$
    by (*simp only*: *zabs-def*)
  show $sgn(i::int) = (if\ i=0\ then\ 0\ else\ if\ 0<i\ then\ 1\ else\ -\ 1)$
    by (*simp only*: *zsgn-def*)
qed

lemma *zless-imp-add1-zle*: $w<z ==> w + (1::int) \le z$
apply (*cases w*, *cases z*)
apply (*simp add*: *less le add One-int-def*)
done

26.4 Magnitude of an Integer, as a Natural Number: *nat*

definition
  $nat :: int \Rightarrow nat$
where
  [*code func del*]: $nat\ z = contents\ (\bigcup (x,\ y) \in Rep\text{-}Integ\ z.\ \{x-y\})$

lemma *nat*: $nat\ (Abs\text{-}Integ\ (intrel``\{(x,y)\})) = x-y$
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*]: $nat\ (of\text{-}nat\ n) = n$
by (*simp add*: *nat int-def*)

lemma *nat-zero* [*simp*]: $nat\ 0 = 0$
by (*simp add*: *Zero-int-def nat*)

lemma *int-nat-eq* [*simp*]: $of\text{-}nat\ (nat\ z) = (if\ 0 \le z\ then\ z\ else\ 0)$
by (*cases z*, *simp add*: *nat le int-def Zero-int-def*)

corollary *nat-0-le*: $0 \le z$ ==> $of\text{-}nat\ (nat\ z) = z$
by *simp*

lemma *nat-le-0* [*simp*]: $z \le 0$ ==> $nat\ z = 0$
by (*cases z*, *simp add*: *nat le Zero-int-def*)

lemma *nat-le-eq-zle*: $0 < w \mid 0 \le z$ ==> $(nat\ w \le nat\ z) = (w \le z)$
apply (*cases w*, *cases z*)
apply (*simp add*: *nat le linorder-not-le* [*symmetric*] *Zero-int-def*, *arith*)
done

An alternative condition is $(0::'a) \le w$

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*: $0 \le w$ ==> $(nat\ w < nat\ z) = (w<z)$
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

lemma *zless-nat-conj* [*simp*]: $(nat\ w < nat\ z) = (0 < z\ \&\ w < z)$
apply (*cases w*, *cases z*)
apply (*simp add*: *nat le Zero-int-def linorder-not-le* [*symmetric*], *arith*)
done

lemma *nonneg-eq-int*:
fixes $z :: int$
assumes $0 \le z$ **and** $\bigwedge m.\ z = of\text{-}nat\ m \Longrightarrow P$
shows P
using *assms* **by** (*blast dest*: *nat-0-le sym*)

lemma *nat-eq-iff*: $(nat\ w = m) = (if\ 0 \le w\ then\ w = of\text{-}nat\ m\ else\ m=0)$
by (*cases w*, *simp add*: *nat le int-def Zero-int-def*, *arith*)

corollary *nat-eq-iff2*: $(m = nat\ w) = (if\ 0 \leq w\ then\ w = of\text{-}nat\ m\ else\ m{=}0)$
by (*simp only*: *eq-commute* [*of m*] *nat-eq-iff*)

lemma *nat-less-iff*: $0 \leq w$ ==> $(nat\ w < m) = (w < of\text{-}nat\ m)$
apply (*cases w*)
apply (*simp add*: *nat le int-def Zero-int-def linorder-not-le* [*symmetric*], *arith*)
done

lemma *int-eq-iff*: $(of\text{-}nat\ m = z) = (m = nat\ z\ \&\ 0 \leq z)$
by (*auto simp add*: *nat-eq-iff2*)

lemma *zero-less-nat-eq* [*simp*]: $(0 < nat\ z) = (0 < z)$
by (*insert zless-nat-conj* [*of 0*], *auto*)

lemma *nat-add-distrib*:
[| $(0{::}int) \leq z$; $0 \leq z'$ |] ==> $nat\ (z{+}z') = nat\ z + nat\ z'$
by (*cases z*, *cases z'*, *simp add*: *nat add le Zero-int-def*)

lemma *nat-diff-distrib*:
[| $(0{::}int) \leq z'$; $z' \leq z$ |] ==> $nat\ (z{-}z') = nat\ z - nat\ z'$
by (*cases z*, *cases z'*,
simp add: *nat add minus diff-minus le Zero-int-def*)

lemma *nat-zminus-int* [*simp*]: $nat\ (-\ (of\text{-}nat\ n)) = 0$
by (*simp add*: *int-def minus nat Zero-int-def*)

lemma *zless-nat-eq-int-zless*: $(m < nat\ z) = (of\text{-}nat\ m < z)$
by (*cases z*, *simp add*: *nat less int-def*, *arith*)

26.5 Lemmas about the Function *of-nat* and Orderings

lemma *negative-zless-0*: $-\ (of\text{-}nat\ (Suc\ n)) < (0\ {::}\ int)$
by (*simp add*: *order-less-le del*: *of-nat-Suc*)

lemma *negative-zless* [*iff*]: $-\ (of\text{-}nat\ (Suc\ n)) < (of\text{-}nat\ m\ {::}\ int)$
by (*rule negative-zless-0* [*THEN order-less-le-trans*], *simp*)

lemma *negative-zle-0*: $-\ of\text{-}nat\ n \leq (0\ {::}\ int)$
by (*simp add*: *minus-le-iff*)

lemma *negative-zle* [*iff*]: $-\ of\text{-}nat\ n \leq (of\text{-}nat\ m\ {::}\ int)$
by (*rule order-trans* [*OF negative-zle-0 of-nat-0-le-iff*])

lemma *not-zle-0-negative* [*simp*]: ~ $(0 \leq -\ (of\text{-}nat\ (Suc\ n)\ {::}\ int))$
by (*subst le-minus-iff*, *simp del*: *of-nat-Suc*)

lemma *int-zle-neg*: $((of\text{-}nat\ n\ {::}\ int) \leq -\ of\text{-}nat\ m) = (n = 0\ \&\ m = 0)$
by (*simp add*: *int-def le minus Zero-int-def*)

lemma *not-int-zless-negative* [*simp*]: ~ ((*of-nat n* :: *int*) < − *of-nat m*)
by (*simp add*: *linorder-not-less*)

lemma *negative-eq-positive* [*simp*]: ((− *of-nat n* :: *int*) = *of-nat m*) = (*n* = *0* & *m* = *0*)
by (*force simp add*: *order-eq-iff* [*of* − *of-nat n*] *int-zle-neg*)

lemma *zle-iff-zadd*: (*w*::*int*) ≤ *z* ⟷ (∃ *n*. *z* = *w* + *of-nat n*)
proof −
  have (*w* ≤ *z*) = (*0* ≤ *z* − *w*)
    by (*simp only*: *le-diff-eq add-0-left*)
  also have ... = (∃ *n*. *z* − *w* = *of-nat n*)
    by (*auto elim*: *zero-le-imp-eq-int*)
  also have ... = (∃ *n*. *z* = *w* + *of-nat n*)
    by (*simp only*: *group-simps*)
  finally show *?thesis* **.**
qed

lemma *zadd-int-left*: *of-nat m* + (*of-nat n* + *z*) = *of-nat* (*m* + *n*) + (*z*::*int*)
by *simp*

lemma *int-Suc0-eq-1*: *of-nat* (*Suc 0*) = (*1*::*int*)
by *simp*

This version is proved for all ordered rings, not just integers! It is proved here because attribute *arith-split* is not available in theory *Ring-and-Field*. But is it really better than just rewriting with *abs-if*?

lemma *abs-split* [*arith-split*,*noatp*]:
  P(*abs*(*a*::′*a*::*ordered-idom*)) = ((*0* ≤ *a* −−> *P a*) & (*a* < *0* −−> *P*(−*a*)))
by (*force dest*: *order-less-le-trans simp add*: *abs-if linorder-not-less*)

26.6 Constants *neg* and *iszero*

definition
  neg :: ′*a*::*ordered-idom* ⇒ *bool*
where
  neg Z ⟷ *Z* < *0*

definition
  iszero :: ′*a*::*semiring-1* ⇒ *bool*
where
  iszero z ⟷ *z* = *0*

lemma *not-neg-int* [*simp*]: ~ *neg* (*of-nat n*)
by (*simp add*: *neg-def*)

lemma *neg-zminus-int* [*simp*]: *neg* (− (*of-nat* (*Suc n*)))
by (*simp add*: *neg-def neg-less-0-iff-less del*: *of-nat-Suc*)

lemmas *neg-eq-less-0* = *neg-def*

lemma *not-neg-eq-ge-0*: $(\sim neg\ x) = (0 \le x)$
by (*simp add*: *neg-def linorder-not-less*)

To simplify inequalities when Numeral1 can get simplified to 1

lemma *not-neg-0*: ~ *neg 0*
by (*simp add*: *One-int-def neg-def*)

lemma *not-neg-1*: ~ *neg 1*
by (*simp add*: *neg-def linorder-not-less zero-le-one*)

lemma *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* ==> *of-nat* (*nat z*) = *z*
by (*simp add*: *linorder-not-less neg-def*)

26.7 Embedding of the Integers into any *ring-1*: *of-int*

context *ring-1*
begin

term *of-nat*

definition
  of-int :: *int* ⇒ *'a*
where
  of-int z = *contents* ($\bigcup$(*i*, *j*) ∈ *Rep-Integ z*. { *of-nat i* − *of-nat j* })
lemmas [*code func del*] = *of-int-def*

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*)

lemma *of-int-1* [*simp*]: *of-int 1* = *1*
by (*simp add*: *of-int One-int-def*)

lemma *of-int-add* [*simp*]: *of-int* (*w*+*z*) = *of-int w* + *of-int z*
by (*cases w*, *cases z*, *simp add*: *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-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 of-nat-mult)
done

Collapse nested embeddings

lemma *of-int-of-nat-eq* [*simp*]: *of-int* (*Nat.of-nat n*) = *of-nat n*
by (*induct n*, *auto*)

end

lemma *of-int-diff* [*simp*]: *of-int* (*w*−*z*) = *of-int w* − *of-int z*
by (*simp add*: *diff-minus*)

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* [*simp*] = *of-int-le-iff* [*of 0*, *simplified*]
lemmas *of-int-le-0-iff* [*simp*] = *of-int-le-iff* [*of - 0*, *simplified*]

lemma *of-int-less-iff* [*simp*]:
(*of-int w* < (*of-int z*::′*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* [*simp*] = *of-int-less-iff* [*of 0*, *simplified*]
lemmas *of-int-less-0-iff* [*simp*] = *of-int-less-iff* [*of - 0*, *simplified*]

Class for unital rings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *ring-char-0* = *ring-1* + *semiring-char-0*

begin

lemma *of-int-eq-iff* [*simp*]:
  of-int w = *of-int* $z \longleftrightarrow w = z$
apply (*cases* w, *cases* z, *simp add*: *of-int*)
apply (*simp only*: *diff-eq-eq diff-add-eq eq-diff-eq*)
apply (*simp only*: *of-nat-add* [*symmetric*] *of-nat-eq-iff*)
done

Special cases where either operand is zero

lemmas *of-int-0-eq-iff* [*simp*] = *of-int-eq-iff* [*of 0*, *simplified*]
lemmas *of-int-eq-0-iff* [*simp*] = *of-int-eq-iff* [*of - 0*, *simplified*]

end

Every *ordered-idom* has characteristic zero.

instance *ordered-idom* $\subseteq$ *ring-char-0* **..**

lemma *of-int-eq-id* [*simp*]: *of-int* = *id*
proof
  fix z **show** *of-int* z = *id* z
    by (*cases* z) (*simp add*: *of-int add minus int-def diff-minus*)
qed

context *ring-1*
begin

lemma *of-nat-nat*: $0 \leq z \Longrightarrow$ *of-nat* (*nat* z) = *of-int* z
  by (*cases* z *rule*: *eq-Abs-Integ*)
   (*simp add*: *nat le of-int Zero-int-def of-nat-diff*)

end

26.8 The Set of Integers

context *ring-1*
begin

definition
  Ints :: *'a set*
where
  Ints = *range of-int*

end

notation (*xsymbols*)
  Ints ($\mathbb{Z}$)

context *ring-1*

begin

lemma *Ints-0* [*simp*]: *0* $\in \mathbb{Z}$
apply (*simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-0* [*symmetric*])
done

lemma *Ints-1* [*simp*]: *1* $\in \mathbb{Z}$
apply (*simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-1* [*symmetric*])
done

lemma *Ints-add* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a + b \in \mathbb{Z}$
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-add* [*symmetric*])
done

lemma *Ints-minus* [*simp*]: $a \in \mathbb{Z} \Longrightarrow -a \in \mathbb{Z}$
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-minus* [*symmetric*])
done

lemma *Ints-mult* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a * b \in \mathbb{Z}$
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-mult* [*symmetric*])
done

lemma *Ints-cases* [*cases set*: *Ints*]:
  assumes $q \in \mathbb{Z}$
  obtains (*of-int*) z **where** $q = $ *of-int* z
  unfolding *Ints-def*
proof −
  from ‹$q \in \mathbb{Z}$› **have** $q \in$ *range of-int* **unfolding** *Ints-def* **.**
  then obtain z **where** $q = $ *of-int* z **..**
  then show *thesis* **..**
qed

lemma *Ints-induct* [*case-names of-int*, *induct set*: *Ints*]:
  $q \in \mathbb{Z} \Longrightarrow (\bigwedge z.\ P\ (\textit{of-int}\ z)) \Longrightarrow P\ q$
  by (*rule Ints-cases*) *auto*

end

lemma *Ints-diff* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a{-}b \in \mathbb{Z}$

apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-diff* [*symmetric*])
done

26.9 *setsum* and *setprod*

By Jeremy Avigad

lemma *of-nat-setsum*: *of-nat* (*setsum f A*) = ($\sum$ *x*∈*A*. *of-nat*(*f x*))
apply (*cases finite A*)
apply (*erule finite-induct*, *auto*)
done

lemma *of-int-setsum*: *of-int* (*setsum f A*) = ($\sum$ *x*∈*A*. *of-int*(*f x*))
apply (*cases finite A*)
apply (*erule finite-induct*, *auto*)
done

lemma *of-nat-setprod*: *of-nat* (*setprod f A*) = ($\prod$ *x*∈*A*. *of-nat*(*f x*))
apply (*cases finite A*)
apply (*erule finite-induct*, *auto simp add*: *of-nat-mult*)
done

lemma *of-int-setprod*: *of-int* (*setprod f A*) = ($\prod$ *x*∈*A*. *of-int*(*f x*))
apply (*cases finite A*)
apply (*erule finite-induct*, *auto*)
done

lemma *setprod-nonzero-nat*:
finite A ==> (∀ *x* ∈ *A*. *f x* ≠ (*0*::*nat*)) ==> *setprod f A* ≠ *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*)

lemmas *int-setsum* = *of-nat-setsum* [**where** *'a*=*int*]
lemmas *int-setprod* = *of-nat-setprod* [**where** *'a*=*int*]

26.10 Further properties

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

lemma *zless-iff-Suc-zadd*:
  $(w :: int) < z \longleftrightarrow (\exists n.\ z = w + \textit{of-nat}\ (\textit{Suc}\ n))$
apply (*cases z*, *cases w*)
apply (*auto simp add*: *less add int-def*)
apply (*rename-tac a b c d*)
apply (*rule-tac x=a+d* $-$ *Suc*(*c+b*) **in** *exI*)
apply *arith*
done

lemma *negD*: $(x :: int) < 0 \Longrightarrow \exists n.\ x = -\ (\textit{of-nat}\ (\textit{Suc}\ n))$
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*) = *of-nat n* ==> *P*; !! *n*. *z* = $-$ (*of-nat* (*Suc n*)) ==> *P* |]
==> *P*
apply (*cases z* < *0*, *blast dest*!: *negD*)
apply (*simp add*: *linorder-not-less del*: *of-nat-Suc*)
apply (*blast dest*: *nat-0-le* [*THEN sym*])
done

theorem *int-induct* [*induct type*: *int*, *case-names nonneg neg*]:
    [|!! *n*. *P* (*of-nat n* :: *int*); !!*n*. *P* ($-$ (*of-nat* (*Suc n*))) |] ==> *P z*
  by (*cases z rule*: *int-cases*) *auto*

Contributed by Brian Huffman

theorem *int-diff-cases*:
  obtains (*diff*) *m n* **where** (*z*::*int*) = *of-nat m* $-$ *of-nat n*
apply (*cases z rule*: *eq-Abs-Integ*)
apply (*rule-tac m=x* **and** *n=y* **in** *diff*)
apply (*simp add*: *int-def diff-def minus add*)
done

26.11 Legacy theorems

lemmas *zminus-zminus* = *minus-minus* [*of z*::*int*, *standard*]
lemmas *zminus-0* = *minus-zero* [**where** *'a=int*]
lemmas *zminus-zadd-distrib* = *minus-add-distrib* [*of z*::*int w*, *standard*]
lemmas *zadd-commute* = *add-commute* [*of z*::*int w*, *standard*]
lemmas *zadd-assoc* = *add-assoc* [*of z1*::*int z2 z3*, *standard*]
lemmas *zadd-left-commute* = *add-left-commute* [*of x*::*int y z*, *standard*]
lemmas *zadd-ac* = *zadd-assoc zadd-commute zadd-left-commute*
lemmas *zmult-ac* = *OrderedGroup.mult-ac*

lemmas *zadd-0* = *OrderedGroup.add-0-left* [*of z*::*int*, *standard*]
lemmas *zadd-0-right* = *OrderedGroup.add-0-left* [*of z*::*int*, *standard*]
lemmas *zadd-zminus-inverse2* = *left-minus* [*of z*::*int*, *standard*]
lemmas *zmult-zminus* = *mult-minus-left* [*of z*::*int w*, *standard*]
lemmas *zmult-commute* = *mult-commute* [*of z*::*int w*, *standard*]
lemmas *zmult-assoc* = *mult-assoc* [*of z1*::*int z2 z3*, *standard*]
lemmas *zadd-zmult-distrib* = *left-distrib* [*of z1*::*int z2 w*, *standard*]
lemmas *zadd-zmult-distrib2* = *right-distrib* [*of w*::*int z1 z2*, *standard*]
lemmas *zdiff-zmult-distrib* = *left-diff-distrib* [*of z1*::*int z2 w*, *standard*]
lemmas *zdiff-zmult-distrib2* = *right-diff-distrib* [*of w*::*int z1 z2*, *standard*]

lemmas *int-distrib* =
zadd-zmult-distrib zadd-zmult-distrib2
zdiff-zmult-distrib zdiff-zmult-distrib2

lemmas *zmult-1* = *mult-1-left* [*of z*::*int*, *standard*]
lemmas *zmult-1-right* = *mult-1-right* [*of z*::*int*, *standard*]

lemmas *zle-refl* = *order-refl* [*of w*::*int*, *standard*]
lemmas *zle-trans* = *order-trans* [**where** *'a*=*int* **and** *x*=*i* **and** *y*=*j* **and** *z*=*k*, *standard*]
lemmas *zle-anti-sym* = *order-antisym* [*of z*::*int w*, *standard*]
lemmas *zle-linear* = *linorder-linear* [*of z*::*int w*, *standard*]
lemmas *zless-linear* = *linorder-less-linear* [**where** *'a* = *int*]

lemmas *zadd-left-mono* = *add-left-mono* [*of i*::*int j k*, *standard*]
lemmas *zadd-strict-right-mono* = *add-strict-right-mono* [*of i*::*int j k*, *standard*]
lemmas *zadd-zless-mono* = *add-less-le-mono* [*of w'*::*int w z' z*, *standard*]

lemmas *int-0-less-1* = *zero-less-one* [**where** *'a*=*int*]
lemmas *int-0-neq-1* = *zero-neq-one* [**where** *'a*=*int*]

lemmas *inj-int* = *inj-of-nat* [**where** *'a*=*int*]
lemmas *int-int-eq* = *of-nat-eq-iff* [**where** *'a*=*int*]
lemmas *zadd-int* = *of-nat-add* [**where** *'a*=*int*, *symmetric*]
lemmas *int-mult* = *of-nat-mult* [**where** *'a*=*int*]
lemmas *zmult-int* = *of-nat-mult* [**where** *'a*=*int*, *symmetric*]
lemmas *int-eq-0-conv* = *of-nat-eq-0-iff* [**where** *'a*=*int* **and** *m*=*n*, *standard*]
lemmas *zless-int* = *of-nat-less-iff* [**where** *'a*=*int*]
lemmas *int-less-0-conv* = *of-nat-less-0-iff* [**where** *'a*=*int* **and** *m*=*k*, *standard*]
lemmas *zero-less-int-conv* = *of-nat-0-less-iff* [**where** *'a*=*int*]
lemmas *zle-int* = *of-nat-le-iff* [**where** *'a*=*int*]
lemmas *zero-zle-int* = *of-nat-0-le-iff* [**where** *'a*=*int*]
lemmas *int-le-0-conv* = *of-nat-le-0-iff* [**where** *'a*=*int* **and** *m*=*n*, *standard*]
lemmas *int-0* = *of-nat-0* [**where** *'a*=*int*]
lemmas *int-1* = *of-nat-1* [**where** *'a*=*int*]
lemmas *int-Suc* = *of-nat-Suc* [**where** *'a*=*int*]
lemmas *abs-int-eq* = *abs-of-nat* [**where** *'a*=*int* **and** *n*=*m*, *standard*]
lemmas *of-int-int-eq* = *of-int-of-nat-eq* [**where** *'a*=*int*]

lemmas *zdiff-int* = *of-nat-diff* [**where** $'a$=*int*, *symmetric*]
lemmas *zless-le* = *less-int-def* [*THEN meta-eq-to-obj-eq*]
lemmas *int-eq-of-nat* = *TrueI*

abbreviation
int :: *nat* ⇒ *int*
where
int ≡ *of-nat*

end

27 Numeral: Arithmetic on Binary Integers

theory *Numeral*
imports *Datatype IntDef*
uses
(*Tools/numeral.ML*)
(*Tools/numeral-syntax.ML*)
begin

27.1 Binary representation

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Tools/twos−compl.ML*, function *int-of-binary*, for the numerical interpretation.

The representation expects that (m *mod* 2) is 0 or 1, even if m is negative; For instance, -5 *div* $2 = -3$ and -5 *mod* $2 = 1$; thus $-5 = (-3)*2 + 1$.

datatype *bit* = *B0* | *B1*

Type *bit* avoids the use of type *bool*, which would make all of the rewrite rules higher-order.

definition
Pls :: *int* **where**
[*code func del*]: *Pls* = *0*

definition
Min :: *int* **where**
[*code func del*]: *Min* = − *1*

definition
Bit :: *int* ⇒ *bit* ⇒ *int* (**infixl** *BIT 90*) **where**
[*code func del*]: *k BIT b* = (*case b of B0* ⇒ *0* | *B1* ⇒ *1*) + *k* + *k*

class *number* = *type* + — for numeric types: nat, int, real, ...
fixes *number-of* :: *int* ⇒ $'a$

use *Tools/numeral.ML*

syntax
  -Numeral :: *num-const* ⇒ *'a*   (-)

use *Tools/numeral-syntax.ML*
setup *NumeralSyntax.setup*

abbreviation
  Numeral0 ≡ *number-of Pls*

abbreviation
  Numeral1 ≡ *number-of* (*Pls BIT B1*)

lemma *Let-number-of* [*simp*]: *Let* (*number-of v*) *f* = *f* (*number-of v*)
  — Unfold all *let*s involving constants
  unfolding *Let-def* **..**

definition
  succ :: *int* ⇒ *int* **where**
  [*code func del*]: *succ k* = *k* + *1*

definition
  pred :: *int* ⇒ *int* **where**
  [*code func del*]: *pred k* = *k* − *1*

lemmas
  max-number-of [*simp*] = *max-def*
    [*of number-of u number-of v*, *standard*, *simp*]
and
  min-number-of [*simp*] = *min-def*
    [*of number-of u number-of v*, *standard*, *simp*]
  — unfolding *minx* and *max* on numerals

lemmas *numeral-simps* =
  succ-def pred-def Pls-def Min-def Bit-def

Removal of leading zeroes

lemma *Pls-0-eq* [*simp*, *code post*]:
  Pls BIT B0 = *Pls*
  unfolding *numeral-simps* **by** *simp*

lemma *Min-1-eq* [*simp*, *code post*]:
  Min BIT B1 = *Min*
  unfolding *numeral-simps* **by** *simp*

27.2 The Functions *succ*, *pred* and *uminus*

lemma *succ-Pls* [*simp*]:
  succ Pls = Pls BIT B1
  unfolding *numeral-simps* **by** *simp*

lemma *succ-Min* [*simp*]:
  succ Min = Pls
  unfolding *numeral-simps* **by** *simp*

lemma *succ-1* [*simp*]:
  succ (*k BIT B1*) = *succ k BIT B0*
  unfolding *numeral-simps* **by** *simp*

lemma *succ-0* [*simp*]:
  succ (*k BIT B0*) = *k BIT B1*
  unfolding *numeral-simps* **by** *simp*

lemma *pred-Pls* [*simp*]:
  pred Pls = Min
  unfolding *numeral-simps* **by** *simp*

lemma *pred-Min* [*simp*]:
  pred Min = Min BIT B0
  unfolding *numeral-simps* **by** *simp*

lemma *pred-1* [*simp*]:
  pred (*k BIT B1*) = *k BIT B0*
  unfolding *numeral-simps* **by** *simp*

lemma *pred-0* [*simp*]:
  pred (*k BIT B0*) = *pred k BIT B1*
  unfolding *numeral-simps* **by** *simp*

lemma *minus-Pls* [*simp*]:
  − *Pls = Pls*
  unfolding *numeral-simps* **by** *simp*

lemma *minus-Min* [*simp*]:
  − *Min = Pls BIT B1*
  unfolding *numeral-simps* **by** *simp*

lemma *minus-1* [*simp*]:
  − (*k BIT B1*) = *pred* (− *k*) *BIT B1*
  unfolding *numeral-simps* **by** *simp*

lemma *minus-0* [*simp*]:
  − (*k BIT B0*) = (− *k*) *BIT B0*
  unfolding *numeral-simps* **by** *simp*

27.3 Binary Addition and Multiplication: *op* + and *op* $*$

lemma *add-Pls* [*simp*]:
Pls + *k* = *k*
unfolding *numeral-simps* **by** *simp*

lemma *add-Min* [*simp*]:
Min + *k* = *pred k*
unfolding *numeral-simps* **by** *simp*

lemma *add-BIT-11* [*simp*]:
(*k BIT B1*) + (*l BIT B1*) = (*k* + *succ l*) *BIT B0*
unfolding *numeral-simps* **by** *simp*

lemma *add-BIT-10* [*simp*]:
(*k BIT B1*) + (*l BIT B0*) = (*k* + *l*) *BIT B1*
unfolding *numeral-simps* **by** *simp*

lemma *add-BIT-0* [*simp*]:
(*k BIT B0*) + (*l BIT b*) = (*k* + *l*) *BIT b*
unfolding *numeral-simps* **by** *simp*

lemma *add-Pls-right* [*simp*]:
k + *Pls* = *k*
unfolding *numeral-simps* **by** *simp*

lemma *add-Min-right* [*simp*]:
k + *Min* = *pred k*
unfolding *numeral-simps* **by** *simp*

lemma *mult-Pls* [*simp*]:
Pls $*$ *w* = *Pls*
unfolding *numeral-simps* **by** *simp*

lemma *mult-Min* [*simp*]:
Min $*$ *k* = − *k*
unfolding *numeral-simps* **by** *simp*

lemma *mult-num1* [*simp*]:
(*k BIT B1*) $*$ *l* = ((*k* $*$ *l*) *BIT B0*) + *l*
unfolding *numeral-simps int-distrib* **by** *simp*

lemma *mult-num0* [*simp*]:
(*k BIT B0*) $*$ *l* = (*k* $*$ *l*) *BIT B0*
unfolding *numeral-simps int-distrib* **by** *simp*

27.4 Converting Numerals to Rings: *number-of*

axclass *number-ring* ⊆ *number*, *comm-ring-1*
number-of-eq: *number-of k* = *of-int k*

self-embedding of the integers

instance *int* :: *number-ring*
  int-number-of-def: *number-of* *w* ≡ *of-int* *w*
  by *intro-classes* (*simp only*: *int-number-of-def*)

lemmas [*code func del*] = *int-number-of-def*

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

lemma *number-of-succ*:
  number-of (*succ k*) = (*1* + *number-of k* ::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-pred*:
  number-of (*pred w*) = (− *1* + *number-of w* ::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-minus*:
  number-of (*uminus w*) = (− (*number-of w*)::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-add*:
  number-of (*v* + *w*) = (*number-of v* + *number-of w*::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-mult*:
  number-of (*v* ∗ *w*) = (*number-of v* ∗ *number-of w*::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

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 B0*) ::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps left-distrib* **by** *simp*

Converting numerals 0 and 1 to their abstract versions.

lemma *numeral-0-eq-0* [*simp*]:
  Numeral0 = (*0*::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *numeral-1-eq-1* [*simp*]:
  Numeral1 = (*1*::'*a*::*number-ring*)
  unfolding *number-of-eq numeral-simps* **by** *simp*

Special-case simplification for small constants.

Unary minus for the abstract constant 1. Cannot be inserted as a simprule until later: it is *number-of-Min* re-oriented!

lemma *numeral-m1-eq-minus-1*:
$(-1::'a::number\text{-}ring) = -\ 1$
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *mult-minus1* [*simp*]:
$-1 * z = -(z::'a::number\text{-}ring)$
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *mult-minus1-right* [*simp*]:
$z * -1 = -(z::'a::number\text{-}ring)$
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *minus-number-of-mult* [*simp*]:
$-\ (number\text{-}of\ w) * z = number\text{-}of\ (uminus\ w) * (z::'a::number\text{-}ring)$
unfolding *number-of-eq* **by** *simp*

Subtraction

lemma *diff-number-of-eq*:
$number\text{-}of\ v - number\text{-}of\ w =$
$(number\text{-}of\ (v + uminus\ w)::'a::number\text{-}ring)$
unfolding *number-of-eq* **by** *simp*

lemma *number-of-Pls*:
$number\text{-}of\ Pls = (0::'a::number\text{-}ring)$
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-Min*:
$number\text{-}of\ Min = (-\ 1::'a::number\text{-}ring)$
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-BIT*:
$number\text{-}of(w\ BIT\ x) = (case\ x\ of\ B0 => 0\ |\ B1 => (1::'a::number\text{-}ring))$
$+ (number\text{-}of\ w) + (number\text{-}of\ w)$
unfolding *number-of-eq numeral-simps* **by** (*simp split*: *bit.split*)

27.5 Equality of Binary Numbers

First version by Norbert Voelker

lemma *eq-number-of-eq*:
$((number\text{-}of\ x::'a::number\text{-}ring) = number\text{-}of\ y) =$
$iszero\ (number\text{-}of\ (x + uminus\ y) :: 'a)$
unfolding *iszero-def number-of-add number-of-minus*
by (*simp add*: *compare-rls*)

lemma *iszero-number-of-Pls*:
$iszero\ ((number\text{-}of\ Pls)::'a::number\text{-}ring)$

```
  unfolding iszero-def numeral-0-eq-0 ..

lemma nonzero-number-of-Min:
  ~ iszero ((number-of Min)::'a::number-ring)
  unfolding iszero-def numeral-m1-eq-minus-1 by simp
```

27.6 Comparisons, for Ordered Rings

```
lemmas double-eq-0-iff = double-zero

lemma le-imp-0-less:
  assumes le: 0 ≤ z
  shows (0::int) < 1 + z
proof −
  have 0 ≤ z by fact
  also have ... < z + 1 by (rule less-add-one)
  also have ... = 1 + z by (simp add: add-ac)
  finally show 0 < 1 + z .
qed

lemma odd-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-double-eq-0-iff:
  assumes in-Ints: a ∈ Ints
  shows (a + a = 0) = (a = (0::'a::ring-char-0))
proof −
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
```

```
  show ?thesis
  proof
    assume a = 0
    thus a + a = 0 by simp
  next
    assume eq: a + a = 0
    hence of-int (z + z) = (of-int 0 :: 'a) by (simp add: a)
    hence z + z = 0 by (simp only: of-int-eq-iff)
    hence z = 0 by (simp only: double-eq-0-iff)
    thus a = 0 by (simp add: a)
  qed
qed

lemma Ints-odd-nonzero:
  assumes in-Ints: a ∈ Ints
  shows 1 + a + a ≠ (0::'a::ring-char-0)
proof −
  from in-Ints have a ∈ range of-int unfolding Ints-def [symmetric] .
  then obtain z where a: a = of-int z ..
  show ?thesis
  proof
    assume eq: 1 + a + a = 0
    hence of-int (1 + z + z) = (of-int 0 :: 'a) by (simp add: a)
    hence 1 + z + z = 0 by (simp only: of-int-eq-iff)
    with odd-nonzero show False by blast
  qed
qed

lemma Ints-number-of:
  (number-of w :: 'a::number-ring) ∈ Ints
  unfolding number-of-eq Ints-def by simp

lemma iszero-number-of-BIT:
  iszero (number-of (w BIT x)::'a) =
   (x = B0 ∧ iszero (number-of w::'a::{ring-char-0,number-ring}))
  by (simp add: iszero-def number-of-eq numeral-simps Ints-double-eq-0-iff
    Ints-odd-nonzero Ints-def split: bit.split)

lemma iszero-number-of-0:
  iszero (number-of (w BIT B0) :: 'a::{ring-char-0,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 B1)::'a::{ring-char-0,number-ring})
  by (simp add: iszero-number-of-BIT)
```

27.7 The Less-Than Relation

lemma *less-number-of-eq-neg*:
  ((*number-of* x::$'a$::{*ordered-idom*,*number-ring*}) < *number-of* y)
  = *neg* (*number-of* (x + *uminus* 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 Pls* ::$'a$::{*ordered-idom*,*number-ring*})
  by (*simp add*: *neg-def numeral-0-eq-0*)

lemma *neg-number-of-Min*:
  neg (*number-of 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*: *of-nat-Suc of-nat-add*
    add: *compare-rls of-nat-1* [*symmetric*] *of-nat-add* [*symmetric*])
qed

The premise involving $\mathbb{Z}$ prevents a = (*1*::$'a$) / (*2*::$'a$).

lemma *Ints-odd-less-0*:
  assumes *in-Ints*: $a \in$ *Ints*
  shows (*1* + a + a < *0*) = (a < (*0*::$'a$::*ordered-idom*))
proof −
  from *in-Ints* **have** $a \in$ *range of-int* **unfolding** *Ints-def* [*symmetric*] **.**
  then obtain z **where** a: a = *of-int* z **..**
  hence ((*1*::$'a$) + a + a < *0*) = (*of-int* (*1* + z + z) < (*of-int 0* :: $'a$))

by (*simp add*: *a*)
also have ... = (z < *0*) **by** (*simp only*: *of-int-less-iff odd-less-0*)
also have ... = (a < *0*) **by** (*simp add*: *a*)
finally show *?thesis* **.**
qed

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 numeral-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* (*y* + *uminus 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*

27.8 Simplification of arithmetic operations on integer constants.

lemmas *arith-extra-simps* [*standard*, *simp*] =
number-of-add [*symmetric*]
number-of-minus [*symmetric*] *numeral-m1-eq-minus-1* [*symmetric*]
number-of-mult [*symmetric*]
diff-number-of-eq abs-number-of

For making a minimal simpset, one must include these default simprules. Also include *simp-thms*.

lemmas *arith-simps* =
bit.distinct

Pls-0-eq Min-1-eq
pred-Pls pred-Min pred-1 pred-0
succ-Pls succ-Min succ-1 succ-0
add-Pls add-Min add-BIT-0 add-BIT-10 add-BIT-11
minus-Pls minus-Min minus-1 minus-0
mult-Pls mult-Min mult-num1 mult-num0
add-Pls-right add-Min-right
abs-zero abs-one arith-extra-simps

Simplification of relational operations

lemmas *rel-simps* [*simp*] =
eq-number-of-eq iszero-0 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

27.9 Simplification of arithmetic when nested to the right.

lemma *add-number-of-left* [*simp*]:
number-of v + (*number-of w* + *z*) =
(*number-of*(*v* + *w*) + *z*::'*a*::*number-ring*)
by (*simp add*: *add-assoc* [*symmetric*])

lemma *mult-number-of-left* [*simp*]:
number-of v ∗ (*number-of w* ∗ *z*) =
(*number-of*(*v* ∗ *w*) ∗ *z*::'*a*::*number-ring*)
by (*simp add*: *mult-assoc* [*symmetric*])

lemma *add-number-of-diff1*:
number-of v + (*number-of w* − *c*) =
number-of(*v* + *w*) − (*c*::'*a*::*number-ring*)
by (*simp add*: *diff-minus add-number-of-left*)

lemma *add-number-of-diff2* [*simp*]:
number-of v + (*c* − *number-of w*) =
number-of (*v* + *uminus w*) + (*c*::'*a*::*number-ring*)
apply (*subst diff-number-of-eq* [*symmetric*])
apply (*simp only*: *compare-rls*)
done

27.10 Configuration of the code generator

instance *int* :: *eq* ..

code-datatype *Pls Min Bit number-of* :: *int* ⇒ *int*

definition

int-aux :: *nat* ⇒ *int* ⇒ *int* **where**
int-aux n i = *int n* + *i*

lemma [*code*]:
int-aux 0 i = *i*
int-aux (*Suc n*) *i* = *int-aux n* (*i* + *1*) — tail recursive
by (*simp add*: *int-aux-def*)+

lemma [*code*, *code unfold*, *code inline del*]:
int n = *int-aux n 0*
by (*simp add*: *int-aux-def*)

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

lemma [*code*]:
nat-aux i n = (*if i* ≤ *0 then n else nat-aux* (*i* − *1*) (*Suc n*)) — tail recursive
by (*auto simp add*: *nat-aux-def nat-eq-iff linorder-not-le order-less-imp-le*
dest: *zless-imp-add1-zle*)

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

lemma *zero-is-num-zero* [*code func*, *code inline*, *symmetric*, *code post*]:
(*0*::*int*) = *Numeral0*
by *simp*

lemma *one-is-num-one* [*code func*, *code inline*, *symmetric*, *code post*]:
(*1*::*int*) = *Numeral1*
by *simp*

code-modulename *SML*
IntDef Integer

code-modulename *OCaml*
IntDef Integer

code-modulename *Haskell*
IntDef Integer

code-modulename *SML*
Numeral Integer

code-modulename *OCaml*
Numeral Integer

code-modulename *Haskell*
Numeral Integer

types-code
int (*int*)
attach (*term-of*) ⟪
val term-of-int = HOLogic.mk-number HOLogic.intT;
⟫
attach (*test*) ⟪
fun gen-int i = one-of [~*1*, *1*] ∗ *random-range 0 i*;
⟫

setup ⟪
let

fun strip-number-of (@{*term Numeral.number-of* :: *int* => *int*} $ *t*) = *t*
| *strip-number-of t* = *t*;

fun numeral-codegen thy defs gr dep module b t =
let val i = *HOLogic.dest-numeral* (*strip-number-of t*)
in
SOME (*fst* (*Codegen.invoke-tycodegen thy defs dep module false* (*gr*, *HOLogic.intT*)),
Pretty.str (*string-of-int i*))
end handle TERM - => *NONE*;

in

Codegen.add-codegen numeral-codegen numeral-codegen

end
⟫

consts-code
number-of :: *int* ⇒ *int* ((-))
0 :: *int* (*0*)
1 :: *int* (*1*)
uminus :: *int* => *int* (~)
op + :: *int* => *int* => *int* ((- +/ -))
op ∗ :: *int* => *int* => *int* ((- ∗/ -))
op ≤ :: *int* => *int* => *bool* ((- <=/ -))
op < :: *int* => *int* => *bool* ((- </ -))

quickcheck-params [*default-type* = *int*]

hide (**open**) *const Pls Min B0 B1 succ pred*

end

28 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 == 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 | 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.
```

28.1 Measure Functions make Wellfounded Relations

28.1.1 ‘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
```

28.1.2 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

lemma *in-inv-image*[*simp*]: ((*x*,*y*) : *inv-image r f*) = ((*f x*, *f y*) : *r*)
  by (*auto simp*:*inv-image-def*)

28.1.3 Finally, All Measures are Wellfounded.

lemma *in-measure*[*simp*]: ((*x*,*y*) : *measure f*) = (*f x* < *f y*)
  by (*simp add*:*measure-def*)

lemma *wf-measure* [*iff*]: *wf* (*measure f*)
apply (*unfold measure-def*)
apply (*rule wf-less-than* [*THEN wf-inv-image*])
done

lemma *measure-induct-rule* [*case-names less*]:
  fixes f :: ′$a \Rightarrow nat$
  assumes *step*: $\bigwedge x.\ (\bigwedge y.\ f\ y < f\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
  shows $P\ a$
proof −
  have *wf* (*measure f*) **..**
  then show *?thesis*
  proof *induct*
    case (*less x*)
    show *?case*
    proof (*rule step*)
      fix *y*
      assume $f\ y < f\ x$
      hence $(y,\ x) \in$ *measure f* **by** *simp*
      thus $P\ y$ **by** (*rule less*)
    qed
  qed
qed

lemma *measure-induct*:
  fixes f :: ′$a \Rightarrow nat$
  shows $(\bigwedge x.\ \forall y.\ f\ y < f\ x \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$
  by (*rule measure-induct-rule* [*of f P a*]) *iprover*

lemma (**in** *linorder*)
  finite-linorder-induct[*consumes 1*, *case-names empty insert*]:
 finite A $\Longrightarrow$ *P* {} $\Longrightarrow$
  (!!*A b*. *finite A* $\Longrightarrow$ *ALL a*:*A*. *a* < *b* $\Longrightarrow$ *P A* $\Longrightarrow$ *P*(*insert b A*))
  $\Longrightarrow$ *P A*
proof (*induct A rule*: *measure-induct*[**where** *f*=*card*])
  fix *A* :: *'a set*
  assume *IH*: *ALL B*. *card B* < *card A* $\longrightarrow$ *finite B* $\longrightarrow$ *P* {} $\longrightarrow$
        ($\forall$ *A b*. *finite A* $\longrightarrow$ ($\forall$ *a*$\in$*A*. *a*<*b*) $\longrightarrow$ *P A* $\longrightarrow$ *P* (*insert b A*))
        $\longrightarrow$ *P B*
  and *finite A* **and** *P* {}
  and *step*: !!*A b*. $[\![$*finite A*; $\forall$ *a*$\in$*A*. *a* < *b*; *P A*$]\!]$ $\Longrightarrow$ *P* (*insert b A*)
  show *P A*
  proof *cases*
    assume *A* = {} **thus** *P A* **using** ‹*P* {}› **by** *simp*
  next
    let *?B* = *A* − {*Max A*} **let** *?A* = *insert* (*Max A*) *?B*
    assume *A* $\neq$ {}
    with ‹*finite A*› **have** *Max A* : *A* **by** *auto*
    hence *A*: *?A* = *A* **using** *insert-Diff-single insert-absorb* **by** *auto*
    note *card-Diff1-less*[*OF* ‹*finite A*› ‹*Max A* : *A*›]
    moreover have *finite ?B* **using** ‹*finite A*› **by** *simp*
    ultimately have *P ?B* **using** ‹*P* {}› *step IH* **by** *blast*
    moreover have $\forall$ *a*$\in$*?B*. *a* < *Max A*
      using *Max-ge*[*OF* ‹*finite A*› ‹*A* $\neq$ {}›] **by** *fastsimp*
    ultimately show *P A*
      using *A insert-Diff-single step*[*OF* ‹*finite ?B*›] **by** *fastsimp*
  qed
qed

28.2 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

lemma *in-lex-prod*[*simp*]:
  (((*a*,*b*),(*a'*,*b'*)): *r* <∗*lex*∗> *s*) = ((*a*,*a'*): *r* $\vee$ (*a* = *a'* $\wedge$ (*b*, *b'*) : *s*))
  by (*auto simp*:*lex-prod-def*)

lexicographic combinations with measure functions

definition

mlex-prod :: ('a ⇒ nat) ⇒ ('a × 'a) set ⇒ ('a × 'a) set (**infixr** <*mlex*> 80)
where
f <*mlex*> R = inv-image (less-than <*lex*> R) (%x. (f x, x))

lemma *wf-mlex*: wf R ⟹ wf (f <*mlex*> R)
unfolding *mlex-prod-def*
by *auto*

lemma *mlex-less*: f x < f y ⟹ (x, y) ∈ f <*mlex*> R
unfolding *mlex-prod-def* **by** *simp*

lemma *mlex-leq*: f x ≤ f y ⟹ (x, y) ∈ R ⟹ (x, y) ∈ f <*mlex*> R
unfolding *mlex-prod-def* **by** *auto*

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*)

28.2.1 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*)

28.2.2 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*)

28.2.3 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

28.3 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*ˆ∗ |] ==> (*f* (*i*+*k*), *f i*) : *r*ˆ∗
apply (*induct-tac k*, *simp-all*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *lemma2*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*ˆ∗; *wf* (*r*ˆ+) |]
  ==> *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*ˆ+)
 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*ˆ∗; *wf* (*r*ˆ+) |]
  ==> *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

29 IntArith: Integer arithmetic

theory *IntArith*
imports *Numeral Wellfounded-Relations*
uses
  ~~/src/Provers/Arith/assoc-fold.ML
  ~~/src/Provers/Arith/cancel-numerals.ML
  ~~/src/Provers/Arith/combine-numerals.ML
  (*int-arith1.ML*)
begin

29.1 Inequality Reasoning for the Arithmetic Simproc

lemma *add-numeral-0*: *Numeral0* + *a* = (*a*::′*a*::*number-ring*)
by *simp*

lemma *add-numeral-0-right*: *a* + *Numeral0* = (*a*::′*a*::*number-ring*)
by *simp*

lemma *mult-numeral-1*: *Numeral1* ∗ *a* = (*a*::′*a*::*number-ring*)
by *simp*

lemma *mult-numeral-1-right*: *a* ∗ *Numeral1* = (*a*::′*a*::*number-ring*)
by *simp*

lemma *divide-numeral-1*: *a* / *Numeral1* = (*a*::′*a*::{*number-ring*,*field*})
by *simp*

lemma *inverse-numeral-1*:
inverse Numeral1 = (*Numeral1*::′*a*::{*number-ring*,*field*})
by *simp*

Theorem lists for the cancellation simprocs. The use of binary numerals for 0 and 1 reduces the number of special cases.

lemmas *add-0s* = *add-numeral-0 add-numeral-0-right*
lemmas *mult-1s* = *mult-numeral-1 mult-numeral-1-right*
mult-minus1 mult-minus1-right

29.2 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*[*simp*] =
add-special diff-special eq-special less-special le-special

lemma *min-max-01*: *min* (*0*::*int*) *1* = *0* & *min* (*1*::*int*) *0* = *0* &
max (*0*::*int*) *1* = *1* & *max* (*1*::*int*) *0* = *1*
by(*simp add*:*min-def max-def*)

lemmas *min-max-special*[*simp*] =
min-max-01
max-def[*of 0*::*int number-of v*, *standard*, *simp*]
min-def[*of 0*::*int number-of v*, *standard*, *simp*]
max-def[*of number-of u 0*::*int*, *standard*, *simp*]
min-def[*of number-of u 0*::*int*, *standard*, *simp*]
max-def[*of 1*::*int number-of v*, *standard*, *simp*]
min-def[*of 1*::*int number-of v*, *standard*, *simp*]
max-def[*of number-of u 1*::*int*, *standard*, *simp*]
min-def[*of number-of u 1*::*int*, *standard*, *simp*]

use *int-arith1.ML*
declaration ⟪ *K int-arith-setup* ⟫

29.3 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 ˆ *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*)

29.4 More Inequality Reasoning

lemma *zless-add1-eq*: ($w < z$ + (*1*::*int*)) = ($w<z$ | $w=z$)
by *arith*

lemma *add1-zle-eq*: (w + (*1*::*int*) $\leq z$) = ($w<z$)
by *arith*

lemma *zle-diff1-eq* [*simp*]: ($w \leq z$ $-$ (*1*::*int*)) = ($w<z$)
by *arith*

lemma *zle-add1-eq-le* [*simp*]: ($w < z$ + (*1*::*int*)) = ($w\leq z$)
by *arith*

lemma *int-one-le-iff-zero-less*: ((*1*::*int*) $\leq z$) = ($0 < z$)
by *arith*

29.5 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*]

lemmas *diff-int-def-symmetric* = *diff-int-def* [*symmetric*, *simp*]

lemma *nat-0*: *nat 0* = *0*
by (*simp add*: *nat-eq-iff*)

lemma *nat-1*: *nat 1* = *Suc 0*
by (*subst nat-eq-iff*, *simp*)

lemma *nat-2*: *nat 2* = *Suc* (*Suc 0*)
by (*subst nat-eq-iff*, *simp*)

lemma *one-less-nat-eq* [*simp*]: (*Suc 0* < *nat z*) = (*1* < *z*)
apply (*insert zless-nat-conj* [*of 1 z*])
apply (*auto simp add*: *nat-1*)
done

This simplifies expressions of the form *int* $n = z$ where z is an integer literal.

lemmas *int-eq-iff-number-of* [*simp*] = *int-eq-iff* [*of* - *number-of v*, *standard*]

lemma *split-nat* [*arith-split*]:
  P(*nat*(*i*::*int*)) = ((∀ *n*. *i* = *int n* ⟶ *P n*) & (*i* < *0* ⟶ *P 0*))
  (**is** *?P* = (*?L* & *?R*))
proof (*cases i* < *0*)
  case *True* **thus** *?thesis* **by** *auto*
next
  case *False*
  have *?P* = *?L*
  proof
    assume *?P* **thus** *?L* **using** *False* **by** *clarsimp*
  next
    assume *?L* **thus** *?P* **using** *False* **by** *simp*
  qed
  with *False* **show** *?thesis* **by** *simp*
qed

context *ring-1*
begin

lemma *of-int-of-nat*:
  of-int k = (*if k* < *0 then* − *of-nat* (*nat* (− *k*)) *else of-nat* (*nat k*))
proof (*cases k* < *0*)
  case *True* **then have** *0* ≤ − *k* **by** *simp*
  then have *of-nat* (*nat* (− *k*)) = *of-int* (− *k*) **by** (*rule of-nat-nat*)

with *True* **show** *?thesis* **by** *simp*
next
case *False* **then show** *?thesis* **by** (*simp add*: *not-less of-nat-nat*)
qed

end

lemma *nat-mult-distrib*: $(0::int) \leq z ==> nat\ (z*z') = nat\ z * nat\ z'$
apply (*cases* $0 \leq z'$)
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*: $z \leq (0::int) ==> nat(z*z') = nat(-z) * nat(-z')$
apply (*rule trans*)
apply (*rule-tac* [*2*] *nat-mult-distrib*, *auto*)
done

lemma *nat-abs-mult-distrib*: $nat\ (abs\ (w * z)) = nat\ (abs\ w) * nat\ (abs\ z)$
apply (*cases* $z{=}0 \mid w{=}0$)
apply (*auto simp add*: *abs-if nat-mult-distrib* [*symmetric*]
nat-mult-distrib-neg [*symmetric*] *mult-less-0-iff*)
done

29.6 Induction principles for int

Well-founded segments of the integers

definition
int-ge-less-than :: *int* => (*int* ∗ *int*) *set*
where
int-ge-less-than $d = \{(z',z).\ d \leq z' \ \&\ z' < z\}$

theorem *wf-int-ge-less-than*: *wf* (*int-ge-less-than d*)
proof −
have *int-ge-less-than d* ⊆ *measure* (%*z*. *nat* (*z*−*d*))
by (*auto simp add*: *int-ge-less-than-def*)
thus *?thesis*
by (*rule wf-subset* [*OF wf-measure*])
qed

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition
int-ge-less-than2 :: *int* => (*int* ∗ *int*) *set*
where
int-ge-less-than2 $d = \{(z',z).\ d \leq z \ \&\ z' < z\}$

theorem *wf-int-ge-less-than2*: *wf* (*int-ge-less-than2 d*)
proof −
  have *int-ge-less-than2 d* $\subseteq$ *measure* (%*z*. *nat* (*1*+*z*−*d*))
    by (*auto simp add*: *int-ge-less-than2-def*)
  thus *?thesis*
    by (*rule wf-subset* [*OF wf-measure*])
qed

theorem *int-ge-induct* [*case-names base step*, *induct set*:*int*]:
  fixes *i* :: *int*
  assumes *ge*: $k \leq i$ **and**
    base: *P k* **and**
    step: $\bigwedge i.\ k \leq i \Longrightarrow P\ i \Longrightarrow P\ (i + 1)$
  shows *P i*
proof −
  { fix *n* **have** $\bigwedge i{::}int.\ n = nat(i-k) \Longrightarrow k \leq i \Longrightarrow P\ i$
    proof (*induct n*)
      case *0*
      hence $i = k$ **by** *arith*
      thus *P i* **using** *base* **by** *simp*
    next
      case (*Suc n*)
      then have $n = nat((i - 1) - k)$ **by** *arith*
      moreover
      have *ki1*: $k \leq i - 1$ **using** *Suc.prems* **by** *arith*
      ultimately
      have $P(i - 1)$ **by**(*rule Suc.hyps*)
      from *step*[*OF ki1 this*] **show** *?case* **by** *simp*
    qed
  }
  with *ge* **show** *?thesis* **by** *fast*
qed

theorem *int-gr-induct*[*case-names base step*,*induct set*:*int*]:
  assumes *gr*: $k < (i{::}int)$ **and**
        base: $P(k+1)$ **and**
        step: $\bigwedge i.\ [\![k < i;\ P\ i]\!] \Longrightarrow P(i+1)$
  shows *P i*
apply(*rule int-ge-induct*[*of k* + *1*])
  using *gr* **apply** *arith*
 apply(*rule base*)
apply (*rule step*, *simp*+)
done

theorem *int-le-induct*[*consumes 1*,*case-names base step*]:
  assumes *le*: $i \leq (k{::}int)$ **and**
        base: $P(k)$ **and**

step: $\bigwedge i.\ [\![i \le k;\ P\ i]\!] \Longrightarrow P(i - 1)$
shows $P\ i$
proof –
{ fix n **have** $\bigwedge i{::}int.\ n = nat(k{-}i) \Longrightarrow i \le k \Longrightarrow P\ i$
proof (*induct n*)
case *0*
hence $i = k$ **by** *arith*
thus $P\ i$ **using** *base* **by** *simp*
next
case (*Suc n*)
hence $n = nat(k - (i{+}1))$ **by** *arith*
moreover
have *ki1*: $i + 1 \le k$ **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: $\bigwedge i.\ [\![i < k;\ P\ i]\!] \Longrightarrow P(i - 1)$
shows $P\ i$
apply(*rule int-le-induct*[*of - k* $-$ *1*])
using *less* **apply** *arith*
apply(*rule base*)
apply (*rule step*, *simp+*)
done

29.7 Intermediate value theorems

lemma *int-val-lemma*:
$(\forall\, i{<}n{::}nat.\ abs(f(i{+}1) - f\ i) \le 1)$ -->
$f\ 0 \le k$ --> $k \le f\ n$ --> $(\exists\, i \le n.\ f\ i = (k{::}int))$
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*:
[| $\forall i.\ m \le i$ & $i < n$ --> $abs(f(i + 1::nat) - f\ i) \le 1$; $m < n$;
$f\ m \le k$; $k \le f\ n$ |] ==> ? $i.\ m \le i$ & $i \le n$ & $f\ i = (k::int)$
apply (*cut-tac* $n = n{-}m$ **and** f = %$i.\ f\ (i{+}m)$ **and** $k = k$
in *int-val-lemma*)
apply *simp*
apply (*erule exE*)
apply (*rule-tac* $x = i{+}m$ **in** *exI*, *arith*)
done

29.8 Products and 1, by T. M. Rasmussen

lemma *zabs-less-one-iff* [*simp*]: $(|z| < 1) = (z = (0::int))$
by *arith*

lemma *abs-zmult-eq-1*: $(|m * n| = 1)$ ==> $|m| = (1::int)$
apply (*cases* $|n|{=}1$)
apply (*simp add*: *abs-mult*)
apply (*rule ccontr*)
apply (*auto simp add*: *linorder-neq-iff abs-mult*)
apply (*subgoal-tac* $2 \le |m|$ & $2 \le |n|$)
prefer *2* **apply** *arith*
apply (*subgoal-tac* $2{*}2 \le |m| * |n|$, *simp*)
apply (*rule mult-mono*, *auto*)
done

lemma *pos-zmult-eq-1-iff-lemma*: $(m * n = 1)$ ==> $m = (1::int) \mid m = -1$
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*: $(m{*}n = (1::int)) = ((m = 1$ & $n = 1) \mid (m = -1$ & $n = -1))$
apply (*rule iffI*)
apply (*frule pos-zmult-eq-1-iff-lemma*)
apply (*simp add*: *mult-commute* [*of m*])
apply (*frule pos-zmult-eq-1-iff-lemma*, *auto*)
done

lemma *infinite-UNIV-int*: ~*finite*(*UNIV*::*int set*)
proof
assume *finite*(*UNIV*::*int set*)
moreover have ~($EX\ i::int.\ 2{*}i = 1$)

by (*auto simp*: *pos-zmult-eq-1-iff*)
ultimately show *False* **using** *finite-UNIV-inj-surj*[*of* %*n*::*int*. *n*+*n*]
by (*simp add*:*inj-on-def surj-def*) (*blast intro*:*sym*)
qed

29.9 Legacy ML bindings

ML ⟪
val of-int-number-of-eq = @{*thm of-int-number-of-eq*};
val nat-0 = @{*thm nat-0*};
val nat-1 = @{*thm nat-1*};
⟫

end

30 Accessible-Part: The accessible part of a relation

theory *Accessible-Part*
imports *Wellfounded-Recursion*
begin

30.1 Inductive definition

Inductive definition of the accessible part *acc r* of a relation; see also [**?**].

inductive-set
acc :: ('*a* ∗ '*a*) *set* => '*a set*
for *r* :: ('*a* ∗ '*a*) *set*
where
accI: (!!*y*. (*y*, *x*) : *r* ==> *y* : *acc r*) ==> *x* : *acc r*

abbreviation
termip :: ('*a* => '*a* => *bool*) => '*a* => *bool* **where**
termip r == *accp* (r^{-1-1})

abbreviation
termi :: ('*a* ∗ '*a*) *set* => '*a set* **where**
termi r == *acc* (r^{-1})

lemmas *accpI* = *accp.accI*

30.2 Induction rules

theorem *accp-induct*:
assumes *major*: *accp r a*
assumes *hyp*: !!*x*. *accp r x* ==> ∀ *y*. *r y x* −−> *P y* ==> *P x*
shows *P a*

```
  apply (rule major [THEN accp.induct])
  apply (rule hyp)
   apply (rule accp.accI)
   apply fast
  apply fast
  done

theorems accp-induct-rule = accp-induct [rule-format, induct set: accp]

theorem accp-downward: accp r b ==> r a b ==> accp r a
  apply (erule accp.cases)
  apply fast
  done

lemma not-accp-down:
  assumes na: ¬ accp R x
  obtains z where R z x and ¬ accp R z
proof −
  assume a: ⋀z. ⟦R z x; ¬ accp R z⟧ ⟹ thesis

  show thesis
  proof (cases ∀ z. R z x ⟶ accp R z)
    case True
    hence ⋀z. R z x ⟹ accp R z by auto
    hence accp R x
      by (rule accp.accI)
    with na show thesis ..
  next
    case False then obtain z where R z x and ¬ accp R z
      by auto
    with a show thesis .
  qed
qed

lemma accp-downwards-aux: r** b a ==> accp r a −−> accp r b
  apply (erule rtranclp-induct)
   apply blast
  apply (blast dest: accp-downward)
  done

theorem accp-downwards: accp r a ==> r** b a ==> accp r b
  apply (blast dest: accp-downwards-aux)
  done

theorem accp-wfPI: ∀ x. accp r x ==> wfP r
  apply (rule wfPUNIVI)
  apply (induct-tac P x rule: accp-induct)
   apply blast
  apply blast
```

```
  done

theorem accp-wfPD: wfP r ==> accp r x
  apply (erule wfP-induct-rule)
  apply (rule accp.accI)
  apply blast
  done

theorem wfP-accp-iff: wfP r = (∀ x. accp r x)
  apply (blast intro: accp-wfPI dest: accp-wfPD)
  done
```

Smaller relations have bigger accessible parts:

```
lemma accp-subset:
  assumes sub: R1 ≤ R2
  shows accp R2 ≤ accp R1
proof
  fix x assume accp R2 x
  then show accp R1 x
  proof (induct x)
    fix x
    assume ih: ⋀y. R2 y x ⟹ accp R1 y
    with sub show accp R1 x
      by (blast intro: accp.accI)
  qed
qed
```

This is a generalized induction theorem that works on subsets of the accessible part.

```
lemma accp-subset-induct:
  assumes subset: D ≤ accp R
    and dcl: ⋀x z. ⟦D x; R z x⟧ ⟹ D z
    and D x
    and istep: ⋀x. ⟦D x; (⋀z. R z x ⟹ P z)⟧ ⟹ P x
  shows P x
proof –
  from subset and ‹D x›
  have accp R x ..
  then show P x using ‹D x›
  proof (induct x)
    fix x
    assume D x
      and ⋀y. R y x ⟹ D y ⟹ P y
    with dcl and istep show P x by blast
  qed
qed
```

Set versions of the above theorems

```
lemmas acc-induct = accp-induct [to-set]
```

lemmas *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

lemmas *acc-downward* = *accp-downward* [*to-set*]

lemmas *not-acc-down* = *not-accp-down* [*to-set*]

lemmas *acc-downwards-aux* = *accp-downwards-aux* [*to-set*]

lemmas *acc-downwards* = *accp-downwards* [*to-set*]

lemmas *acc-wfI* = *accp-wfPI* [*to-set*]

lemmas *acc-wfD* = *accp-wfPD* [*to-set*]

lemmas *wf-acc-iff* = *wfP-accp-iff* [*to-set*]

lemmas *acc-subset* = *accp-subset* [*to-set*]

lemmas *acc-subset-induct* = *accp-subset-induct* [*to-set*]

end

31 FunDef: General recursive function definitions

theory *FunDef*
imports *Accessible-Part*
uses
(*Tools/function-package/fundef-lib.ML*)
(*Tools/function-package/fundef-common.ML*)
(*Tools/function-package/inductive-wrap.ML*)
(*Tools/function-package/context-tree.ML*)
(*Tools/function-package/fundef-core.ML*)
(*Tools/function-package/mutual.ML*)
(*Tools/function-package/pattern-split.ML*)
(*Tools/function-package/fundef-package.ML*)
(*Tools/function-package/auto-term.ML*)
begin

Definitions with default value.

definition
THE-default :: $'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
THE-default $d\ P = (\mathit{if}\ (\exists!x.\ P\ x)\ \mathit{then}\ (\mathit{THE}\ x.\ P\ x)\ \mathit{else}\ d)$

lemma *THE-defaultI'*: $\exists!x.\ P\ x \Longrightarrow P\ (\mathit{THE\text{-}default}\ d\ P)$
by (*simp add*: *theI' THE-default-def*)

lemma *THE-default1-equality*:

  $[\![\exists!x.\ P\ x;\ P\ a]\!] \Longrightarrow$ *THE-default* $d\ P = a$
  by (*simp add*: *the1-equality THE-default-def*)

lemma *THE-default-none*:
  $\neg(\exists!x.\ P\ x) \Longrightarrow$ *THE-default* $d\ P = d$
  by (*simp add*:*THE-default-def*)

lemma *fundef-ex1-existence*:
  assumes *f-def*: $f == (\lambda x{::}'a.$ *THE-default* $(d\ x)\ (\lambda y.\ G\ x\ y))$
  assumes *ex1*: $\exists!y.\ G\ x\ y$
  shows $G\ x\ (f\ x)$
  apply (*simp only*: *f-def*)
  apply (*rule THE-defaultI'*)
  apply (*rule ex1*)
  done

lemma *fundef-ex1-uniqueness*:
  assumes *f-def*: $f == (\lambda x{::}'a.$ *THE-default* $(d\ x)\ (\lambda y.\ G\ x\ y))$
  assumes *ex1*: $\exists!y.\ G\ x\ y$
  assumes *elm*: $G\ x\ (h\ x)$
  shows $h\ x = f\ x$
  apply (*simp only*: *f-def*)
  apply (*rule THE-default1-equality* [*symmetric*])
   apply (*rule ex1*)
  apply (*rule elm*)
  done

lemma *fundef-ex1-iff*:
  assumes *f-def*: $f == (\lambda x{::}'a.$ *THE-default* $(d\ x)\ (\lambda y.\ G\ x\ y))$
  assumes *ex1*: $\exists!y.\ G\ x\ y$
  shows $(G\ x\ y) = (f\ x = y)$
  apply (*auto simp*:*ex1 f-def THE-default1-equality*)
  apply (*rule THE-defaultI'*)
  apply (*rule ex1*)
  done

lemma *fundef-default-value*:
  assumes *f-def*: $f == (\lambda x{::}'a.$ *THE-default* $(d\ x)\ (\lambda y.\ G\ x\ y))$
  assumes *graph*: $\bigwedge x\ y.\ G\ x\ y \Longrightarrow D\ x$
  assumes $\neg\ D\ x$
  shows $f\ x = d\ x$
proof −
  have $\neg(\exists\, y.\ G\ x\ y)$
  proof
    assume $\exists\, y.\ G\ x\ y$
    hence $D\ x$ **using** *graph* **..**
    with $\langle\neg\ D\ x\rangle$ **show** *False* **..**
  qed

hence $\neg(\exists!y.\ G\ x\ y)$ **by** *blast*

thus *?thesis*
unfolding *f-def*
by (*rule THE-default-none*)
qed

definition *in-rel-def* [*simp*]:
in-rel R x y == $(x,\ y) \in R$

lemma *wf-in-rel*:
wf R $\Longrightarrow$ *wfP* (*in-rel R*)
by (*simp add*: *wfP-def*)

use *Tools/function-package/fundef-lib.ML*
use *Tools/function-package/fundef-common.ML*
use *Tools/function-package/inductive-wrap.ML*
use *Tools/function-package/context-tree.ML*
use *Tools/function-package/fundef-core.ML*
use *Tools/function-package/mutual.ML*
use *Tools/function-package/pattern-split.ML*
use *Tools/function-package/auto-term.ML*
use *Tools/function-package/fundef-package.ML*

setup ⟪ *FundefPackage.setup* ⟫

lemma *let-cong* [*fundef-cong*]:
$M = N \Longrightarrow (\bigwedge x.\ x = N \Longrightarrow f\ x = g\ x) \Longrightarrow Let\ M\ f = Let\ N\ g$
unfolding *Let-def* **by** *blast*

lemmas [*fundef-cong*] =
if-cong image-cong INT-cong UN-cong
bex-cong ball-cong imp-cong

lemma *split-cong* [*fundef-cong*]:
$(\bigwedge x\ y.\ (x,\ y) = q \Longrightarrow f\ x\ y = g\ x\ y) \Longrightarrow p = q$
$\Longrightarrow split\ f\ p = split\ g\ q$
by (*auto simp*: *split-def*)

lemma *comp-cong* [*fundef-cong*]:
$f\ (g\ x) = f'\ (g'\ x') \Longrightarrow (f\ o\ g)\ x = (f'\ o\ g')\ x'$
unfolding *o-apply* **.**

end

32 IntDiv: The Division Operators div and mod; the Divides Relation dvd

theory *IntDiv*
imports *IntArith Divides FunDef*
begin

constdefs
quorem :: (*int*∗*int*) ∗ (*int*∗*int*) => *bool*
— definition of quotient and remainder
[*code func*]: *quorem* == %((*a*,*b*), (*q*,*r*)).
$a = b*q + r$ &
(*if* $0 < b$ *then* $0 \leq r$ & $r < b$ *else* $b < r$ & $r \leq 0$)

adjust :: [*int*, *int*∗*int*] => *int*∗*int*
— for the division algorithm
[*code func*]: *adjust b* == %(*q*,*r*). *if* $0 \leq r-b$ *then* $(2*q + 1, r-b)$
else $(2*q, r)$

algorithm for the case $a \geq 0$, $b > 0$

function
posDivAlg :: *int* ⇒ *int* ⇒ *int* × *int*
where
posDivAlg a b =
(*if* ($a < b$ | $b \leq 0$) *then* $(0,a)$
else adjust b (*posDivAlg a* $(2*b)$))
by *auto*
termination by (*relation measure* (%(*a*,*b*). *nat*($a - b + 1$))) *auto*

algorithm for the case $a < 0$, $b > 0$

function
negDivAlg :: *int* ⇒ *int* ⇒ *int* × *int*
where
negDivAlg a b =
(*if* ($0 \leq a+b$ | $b \leq 0$) *then* $(-1,a+b)$
else adjust b (*negDivAlg a* $(2*b)$))
by *auto*
termination by (*relation measure* (%(*a*,*b*). *nat*($- a - b$))) *auto*

algorithm for the general case $b \neq (0::'a)$

constdefs
negateSnd :: *int*∗*int* => *int*∗*int*
[*code func*]: *negateSnd* == %(*q*,*r*). $(q,-r)$

definition
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 $a < (0::'a)$.

where
divAlg = (λ(*a*, *b*). (*if* $0 \le a$ *then*
  if $0 \le b$ *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*
div-def: *a div b* == *fst* (*divAlg* (*a*, *b*))
mod-def: *a mod b* == *snd* (*divAlg* (*a*, *b*)) ..

lemma *divAlg-mod-div*:
divAlg (*p*, *q*) = (*p div q*, *p mod q*)
by (*auto simp add*: *div-def mod-def*)

Here is the division algorithm in ML:

```
fun posDivAlg (a,b) =
  if a<b then (0,a)
  else let val (q,r) = posDivAlg(a, 2*b)
           in  if 0\<le>r-b then (2*q+1, r-b) else (2*q, r)
       end

fun negDivAlg (a,b) =
  if 0\<le>a+b then (~1,a+b)
  else let val (q,r) = negDivAlg(a, 2*b)
           in  if 0\<le>r-b then (2*q+1, r-b) else (2*q, r)
       end;

fun negateSnd (q,r:int) = (q,~r);

fun 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));
```

32.1 Uniqueness and Monotonicity of Quotients and Remainders

lemma *unique-quotient-lemma*:

[| $b*q' + r' \le b*q + r$; $0 \le r'$; $r' < b$; $r < b$ |]
==> $q' \le (q::int)$
apply (*subgoal-tac* $r' + b * (q'-q) \le r$)
prefer *2* **apply** (*simp add*: *right-diff-distrib*)
apply (*subgoal-tac* $0 < b * (1 + q - q')$)
apply (*erule-tac* [*2*] *order-le-less-trans*)
prefer *2* **apply** (*simp add*: *right-diff-distrib right-distrib*)
apply (*subgoal-tac* $b * q' < b * (1 + q)$)
prefer *2* **apply** (*simp add*: *right-diff-distrib right-distrib*)
apply (*simp add*: *mult-less-cancel-left*)
done

lemma *unique-quotient-lemma-neg*:
[| $b*q' + r' \le b*q + r$; $r \le 0$; $b < r$; $b < r'$ |]
==> $q \le (q'::int)$
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')); $b \ne 0$ |]
==> $q = q'$
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')); $b \ne 0$ |]
==> $r = r'$
apply (*subgoal-tac* $q = q'$)
apply (*simp add*: *quorem-def*)
apply (*blast intro*: *unique-quotient*)
done

32.2 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 $0 \le$ *diff then* ($2*q + 1$, *diff*)
else ($2*q$, *r*))
by (*simp add*: *Let-def adjust-def*)

declare *posDivAlg.simps* [*simp del*] use with a simproc to avoid repeatedly proving the premise

lemma *posDivAlg-eqn*:
  $0 < b ==>$
   $posDivAlg\ a\ b = (if\ a{<}b\ then\ (0{,}a)\ else\ adjust\ b\ (posDivAlg\ a\ (2{*}b)))$
by (*rule posDivAlg.simps* [*THEN trans*], *simp*)

Correctness of *posDivAlg*: it computes quotients correctly

theorem *posDivAlg-correct*:
 assumes $0 \leq a$ **and** $0 < b$
 shows *quorem* ((*a*, *b*), *posDivAlg a b*)
using *prems* **apply** (*induct a b rule*: *posDivAlg.induct*)
apply *auto*
apply (*simp add*: *quorem-def*)
apply (*subst posDivAlg-eqn*, *simp add*: *right-distrib*)
apply (*case-tac* $a < b$)
apply *simp-all*
apply (*erule splitE*)
apply (*auto simp add*: *right-distrib Let-def*)
done

32.3 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*:
  $0 < b ==>$
   $negDivAlg\ a\ b =$
    $(if\ 0{\leq}a{+}b\ then\ (-1{,}a{+}b)\ else\ adjust\ b\ (negDivAlg\ a\ (2{*}b)))$
by (*rule negDivAlg.simps* [*THEN trans*], *simp*)

lemma *negDivAlg-correct*:
 assumes $a < 0$ **and** $b > 0$
 shows *quorem* ((*a*, *b*), *negDivAlg a b*)
using *prems* **apply** (*induct a b rule*: *negDivAlg.induct*)
apply (*auto simp add*: *linorder-not-le*)
apply (*simp add*: *quorem-def*)
apply (*subst negDivAlg-eqn*, *assumption*)
apply (*case-tac* $a + b < (0{::}int)$)
apply *simp-all*
apply (*erule splitE*)
apply (*auto simp add*: *right-distrib Let-def*)
done

32.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma *quorem-0*: $b \neq 0$ ==> *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*: $b \neq 0$ ==> *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*])

Tool setup

ML-setup ⟪
local

structure CancelDivMod = *CancelDivModFun*(
struct
  val div-name = @{*const-name Divides.div*};
  val mod-name = @{*const-name Divides.mod*};
  val mk-binop = *HOLogic.mk-binop*;

val mk-sum = *Int-Numeral-Simprocs.mk-sum HOLogic.intT*;
val dest-sum = *Int-Numeral-Simprocs.dest-sum*;
val div-mod-eqs =
map mk-meta-eq [@{*thm zdiv-zmod-equality*},
@{*thm zdiv-zmod-equality2*}];
val trans = *trans*;
val prove-eq-sums =
let
val simps = @{*thm diff-int-def*} :: *Int-Numeral-Simprocs.add-0s* @ @{*thms zadd-ac*}
in NatArithUtils.prove-conv all-tac (*NatArithUtils.simp-all-tac simps*) *end*;
end)

in

val cancel-zdiv-zmod-proc = *NatArithUtils.prep-simproc*
(*cancel-zdiv-zmod*, [(*m*::*int*) + *n*], *K CancelDivMod.proc*)

end;

Addsimprocs [*cancel-zdiv-zmod-proc*]
⟫

lemma *pos-mod-conj* : (*0*::*int*) < *b* ==> *0* ≤ *a mod b* & *a mod b* < *b*
apply (*cut-tac a* = *a* **and** *b* = *b* **in** *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*) ==> *a mod b* ≤ *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*]

32.5 General Properties of div and mod

lemma *quorem-div-mod*: *b* ≠ *0* ==> *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* ≠ *0* |] ==> *a div b* = *q*
by (*simp add*: *quorem-div-mod* [*THEN unique-quotient*])

lemma *quorem-mod*: [| *quorem*((*a*,*b*),(*q*,*r*)); $b \neq 0$ |] ==> *a mod b* = *r*
by (*simp add*: *quorem-div-mod* [*THEN unique-remainder*])

lemma *div-pos-pos-trivial*: [| (*0*::*int*) ≤ *a*; *a* < *b* |] ==> *a div b* = *0*
apply (*rule quorem-div*)
apply (*auto simp add*: *quorem-def*)
done

lemma *div-neg-neg-trivial*: [| *a* ≤ (*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* ≤ *0* |] ==> *a div b* = −*1*
apply (*rule quorem-div*)
apply (*auto simp add*: *quorem-def*)
done

lemma *mod-pos-pos-trivial*: [| (*0*::*int*) ≤ *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* ≤ (*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* ≤ *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

32.6 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*)

32.7 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*)); $a \neq (0::int)$ |] ==> $r = 0$
apply (*frule self-quotient*, *assumption*)
apply (*simp add*: *quorem-def*)
done

lemma *zdiv-self* [*simp*]: $a \neq 0$ ==> *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

32.8 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*: [| $0 < a$; $0 \leq b$ |] ==> *a div b* = *fst* (*posDivAlg a b*)
by (*simp add*: *div-def divAlg-def*)

lemma *mod-pos-pos*: [| $0 < a$; $0 \leq b$ |] ==> *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

lemma *quoremI*:
 $[\![a == b * q + r;\ \mathit{if}\ 0 < b\ \mathit{then}\ 0 \leq r \wedge r < b\ \mathit{else}\ b < r \wedge r \leq 0]\!]$
  $\Longrightarrow$ *quorem* $((a,\ b),\ (q,\ r))$
 unfolding *quorem-def* **by** *simp*

lemmas *quorem-div-eq* = *quoremI* [*THEN quorem-div*, *THEN eq-reflection*]
lemmas *quorem-mod-eq* = *quoremI* [*THEN quorem-mod*, *THEN eq-reflection*]
lemmas *arithmetic-simps* =
 arith-simps
 add-special
 OrderedGroup.add-0-left
 OrderedGroup.add-0-right
 mult-zero-left
 mult-zero-right
 mult-1-left
 mult-1-right

ML ⟪
local
 infix ==;
 val op == = *Logic.mk-equals*;
 fun plus m n = @{*term plus* :: *int* ⇒ *int* ⇒ *int*} \$ *m* \$ *n*;
 fun mult m n = @{*term times* :: *int* ⇒ *int* ⇒ *int*} \$ *m* \$ *n*;

 val binary-ss = *HOL-basic-ss addsimps* @{*thms arithmetic-simps*};

```
  fun prove ctxt prop =
    Goal.prove ctxt [] [] prop (fn - => ALLGOALS (full-simp-tac binary-ss));

  fun binary-proc proc ss ct =
    (case Thm.term-of ct of
      - $ t $ u =>
      (case try (pairself (`(snd o HOLogic.dest-number))) (t, u) of
        SOME args => proc (Simplifier.the-context ss) args
      | NONE => NONE)
    | - => NONE);
in

fun divmod-proc rule = binary-proc (fn ctxt => fn ((m, t), (n, u)) =>
  if n = 0 then NONE
  else
    let val (k, l) = Integer.div-mod m n;
        fun mk-num x = HOLogic.mk-number HOLogic.intT x;
    in SOME (rule OF [prove ctxt (t == plus (mult u (mk-num k)) (mk-num l))])
    end);

end;
»
```

simproc-setup *binary-int-div* (*number-of m div number-of n* :: *int*) =
  « *K* (*divmod-proc* (@{*thm quorem-div-eq*})) »

simproc-setup *binary-int-mod* (*number-of m mod number-of n* :: *int*) =
  « *K* (*divmod-proc* (@{*thm quorem-mod-eq*})) »

lemmas *div-pos-pos-number-of* =
  div-pos-pos [*of number-of v number-of w*, *standard*]

lemmas *div-neg-pos-number-of* =
  div-neg-pos [*of number-of v number-of w*, *standard*]

lemmas *div-pos-neg-number-of* =
  div-pos-neg [*of number-of v number-of w*, *standard*]

lemmas *div-neg-neg-number-of* =
  div-neg-neg [*of number-of v number-of w*, *standard*]

lemmas *mod-pos-pos-number-of* =
  mod-pos-pos [*of number-of v number-of w*, *standard*]

lemmas *mod-neg-pos-number-of* =
  mod-neg-pos [*of number-of v number-of w*, *standard*]

lemmas *mod-pos-neg-number-of* =
mod-pos-neg [*of number-of v number-of w*, *standard*]

lemmas *mod-neg-neg-number-of* =
mod-neg-neg [*of number-of v number-of w*, *standard*]

lemmas *posDivAlg-eqn-number-of* [*simp*] =
posDivAlg-eqn [*of number-of v number-of w*, *standard*]

lemmas *negDivAlg-eqn-number-of* [*simp*] =
negDivAlg-eqn [*of number-of v number-of w*, *standard*]

Special-case simplification

lemma *zmod-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* [*simp*] =
div-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *div-pos-neg-1-number-of* [*simp*] =
div-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *mod-pos-pos-1-number-of* [*simp*] =
mod-pos-pos [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *mod-pos-neg-1-number-of* [*simp*] =
mod-pos-neg [*OF int-0-less-1*, *of number-of w*, *standard*]

lemmas *posDivAlg-eqn-1-number-of* [*simp*] =

posDivAlg-eqn [*of* **concl**: *1 number-of w*, *standard*]

lemmas *negDivAlg-eqn-1-number-of* [*simp*] =
negDivAlg-eqn [*of* **concl**: *1 number-of w*, *standard*]

32.9 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*: $[\mid a \leq a';\ 0 < (b::int) \mid] ==> a\ div\ b \leq a'\ div\ b$
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*: $[\mid a \leq a';\ (b::int) < 0 \mid] ==> a'\ div\ b \leq a\ div\ b$
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

32.10 Monotonicity in the Second Argument (Divisor)

lemma *q-pos-lemma*:
$[\mid 0 \leq b'*q' + r';\ r' < b';\ 0 < b' \mid] ==> 0 \leq (q'::int)$
apply (*subgoal-tac* $0 < b'* (q' + 1)$)
apply (*simp add*: *zero-less-mult-iff*)
apply (*simp add*: *right-distrib*)
done

lemma *zdiv-mono2-lemma*:
$[\mid b*q + r = b'*q' + r';\ 0 \leq b'*q' + r';$
$r' < b';\ 0 \leq r;\ 0 < b';\ b' \leq b \mid]$
$==> q \leq (q'::int)$
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*:
$[\mid (0::int) \leq a;\ 0 < b';\ b' \leq b \mid] ==> a\ div\ b \leq a\ div\ b'$
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

32.11 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* [*dest!*] = *zmod-eq-0-iff* [*THEN iffD1*]

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

lemma *zmod-zdiff1-eq*: **fixes** *a::int*
shows $(a - b)$ *mod* $c = (a$ *mod* $c - b$ *mod* $c)$ *mod* c (**is** *?l* = *?r*)
proof –

have *?l* = (*c* + (*a mod c* − *b mod c*)) *mod c*
using *zmod-zadd1-eq*[*of a* −*b c*] **by**(*simp add*:*ring-simps zmod-zminus1-eq-if*)
also have ... = *?r* **by** *simp*
finally show *?thesis* **.**
qed

32.12 Proving *a div* (*b* ∗ *c*) = *a div b div c*

first, four lemmas to bound the remainder for the cases b¡0 and b¿0

lemma *zmult2-lemma-aux1*: [| (*0*::*int*) < *c*; *b* < *r*; *r* ≤ *0* |] ==> *b*∗*c* < *b*∗(*q mod c*) + *r*
apply (*subgoal-tac b* ∗ (*c* − *q mod c*) < *r* ∗ *1*)
apply (*simp add*: *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* ≤ *0* |] ==> *b* ∗ (*q mod c*) + *r* ≤ *0*
apply (*subgoal-tac b* ∗ (*q mod c*) ≤ *0*)
apply *arith*
apply (*simp add*: *mult-le-0-iff*)
done

lemma *zmult2-lemma-aux3*: [| (*0*::*int*) < *c*; *0* ≤ *r*; *r* < *b* |] ==> *0* ≤ *b* ∗ (*q mod c*) + *r*
apply (*subgoal-tac 0* ≤ *b* ∗ (*q mod c*))
apply *arith*
apply (*simp add*: *zero-le-mult-iff*)
done

lemma *zmult2-lemma-aux4*: [| (*0*::*int*) < *c*; *0* ≤ *r*; *r* < *b* |] ==> *b* ∗ (*q mod c*) + *r* < *b* ∗ *c*
apply (*subgoal-tac r* ∗ *1* < *b* ∗ (*c* − *q mod c*))
apply (*simp add*: *right-diff-distrib*)
apply (*rule order-less-le-trans*)
apply (*erule mult-strict-right-mono*)
apply (*rule-tac* [*2*] *mult-left-mono*)
apply (*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* ≠ *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

32.13 Cancellation of Common Factors in div

lemma *zdiv-zmult-zmult1-aux1*:
$[|\ (0::int) < b;\ \ c \neq 0\ |] ==> (c*a)\ div\ (c*b) = a\ div\ b$
by (*subst zdiv-zmult2-eq*, *auto*)

lemma *zdiv-zmult-zmult1-aux2*:
$[|\ b < (0::int);\ \ c \neq 0\ |] ==> (c*a)\ div\ (c*b) = a\ div\ b$
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*: $c \neq (0::int) ==> (c*a)\ div\ (c*b) = a\ div\ b$
apply (*case-tac* $b = 0$, *simp*)
apply (*auto simp add*: *linorder-neq-iff zdiv-zmult-zmult1-aux1 zdiv-zmult-zmult1-aux2*)
done

lemma *zdiv-zmult-zmult1-if* [*simp*]:
$(k*m)\ div\ (k*n) = (if\ k = (0::int)\ then\ 0\ else\ m\ div\ n)$
by (*simp add*:*zdiv-zmult-zmult1*)

32.14 Distribution of Factors over mod

lemma *zmod-zmult-zmult1-aux1*:
$[|\ (0::int) < b;\ \ c \neq 0\ |] ==> (c*a)\ mod\ (c*b) = c * (a\ mod\ b)$
by (*subst zmod-zmult2-eq*, *auto*)

lemma *zmod-zmult-zmult1-aux2*:
$[|\ b < (0::int);\ \ c \neq 0\ |] ==> (c*a)\ mod\ (c*b) = c * (a\ mod\ b)$
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

lemma *zmod-zmod-cancel*:
assumes *n dvd m* **shows** (*k*::*int*) *mod m mod n* = *k mod n*
proof −
  from ‹*n dvd m*› **obtain** *r* **where** $m = n*r$ **by**(*auto simp*:*dvd-def*)
  have *k mod n* = $(m * (k$ *div* $m) + k$ *mod* $m)$ *mod n*
    using *zmod-zdiv-equality*[*of k m*] **by** *simp*
  also have ... = $(m * (k$ *div* $m)$ *mod* $n + k$ *mod* m *mod* $n)$ *mod n*
    by(*subst zmod-zadd1-eq*, *rule refl*)
  also have $m * (k$ *div* $m)$ *mod* $n = 0$ **using** ‹$m = n*r$›
    by(*simp add*:*mult-ac*)
  finally show *?thesis* **by** *simp*
qed

32.15 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* $k) = (\forall\, i\, j.\ 0 \le j\ \&\ j<k\ \&\ n = k*i + j$ −−> $P\ i\ j)$
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* $k) = (\forall\, i\, j.\ k<j\ \&\ j \le 0\ \&\ n = k*i + j$ −−> $P\ i\ j)$
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]
```

32.16 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*: $a \le (0::int) ==> (1 + 2*b)\ div\ (2*a) = (b+1)\ div\ a$
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*: $\sim\ 0 \le x ==> x \le (0::int)$
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 numeral-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

32.17 Computing mod by Shifting (proofs resemble those for div)

lemma *pos-zmod-mult-2*:
$(0::int) \le a ==> (1 + 2*b)\ mod\ (2*a) = 1 + 2 * (b\ mod\ a)$
apply (*case-tac* $a = 0$, *simp*)
apply (*subgoal-tac* $1 < a * 2$)
prefer *2* **apply** *arith*
apply (*subgoal-tac* $2*(1 + b\ mod\ a) \le 2*a$)
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* $0 \le b\ mod\ a$, *arith*, *simp*)

done

lemma *neg-zmod-mult-2*:
$a \le (0::int) ==> (1 + 2*b) \; mod \; (2*a) = 2 * ((b+1) \; mod \; a) - 1$
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* $(0::int) \le$ *number-of w*
then $2 *$ (*number-of v mod number-of w*) + 1
else $2 *$ ((*number-of v* + (1::*int*)) *mod number-of w*) − 1)
apply (*simp only*: *number-of-eq numeral-simps 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

32.18 Quotients of Signs

lemma *div-neg-pos-less0*: $[| \; a < (0::int); \; 0 < b \; |] ==> a \; div \; b < 0$
apply (*subgoal-tac* $a \; div \; b \le -1$, *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*: $[| \; (0::int) \le a; \; b < 0 \; |] ==> a \; div \; b \le 0$
by (*drule zdiv-mono1-neg*, *auto*)

lemma *pos-imp-zdiv-nonneg-iff*: $(0::int) < b ==> (0 \le a \; div \; b) = (0 \le a)$
apply *auto*
apply (*drule-tac* [2] *zdiv-mono1*)
apply (*auto simp add*: *linorder-neq-iff*)
apply (*simp* (*no-asm-use*) *add*: *linorder-not-less* [*symmetric*])
apply (*blast intro*: *div-neg-pos-less0*)
done

lemma *neg-imp-zdiv-nonneg-iff*:
$b < (0::int) ==> (0 \le a \; div \; b) = (a \le (0::int))$
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*)

32.19 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*)

instance *int* :: *dvd-mod*
by *default* (*simp add*: *zdvd-iff-zmod-eq-0*)

lemmas *zdvd-iff-zmod-eq-0-number-of* [*simp*] =
zdvd-iff-zmod-eq-0 [*of number-of x number-of y*, *standard*]

lemma *zdvd-0-right* [*iff*]: (*m::int*) *dvd 0*
by (*simp add*: *dvd-def*)

lemma *zdvd-0-left* [*iff*,*noatp*]: (*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-abs1*: (|*i::int*| *dvd j*) = (*i dvd j*)
apply (*cases i* > *0*, *simp*)
apply (*simp add*: *dvd-def*)
apply (*rule iffI*)

apply (*erule exE*)
apply (*rule-tac x*=− *k* **in** *exI*, *simp*)
apply (*erule exE*)
apply (*rule-tac x*=− *k* **in** *exI*, *simp*)
done
lemma *zdvd-abs2*: ((*i*::*int*) *dvd* |*j*|) = (*i dvd j*)
apply (*cases j* > *0*, *simp*)
apply (*simp add*: *dvd-def*)
apply (*rule iffI*)
apply (*erule exE*)
apply (*rule-tac x*=− *k* **in** *exI*, *simp*)
apply (*erule exE*)
apply (*rule-tac x*=− *k* **in** *exI*, *simp*)
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-dvd-eq*: **assumes** *anz*:*a* ≠ *0* **and** *ab*: (*a*::*int*) *dvd b* **and** *ba*:*b dvd a*
shows |*a*| = |*b*|
proof−
from *ab* **obtain** *k* **where** *k*:*b* = *a*∗*k* **unfolding** *dvd-def* **by** *blast*
from *ba* **obtain** *k′* **where** *k′*:*a* = *b*∗*k′* **unfolding** *dvd-def* **by** *blast*
from *k k′* **have** *a* = *a*∗*k*∗*k′* **by** *simp*
with *mult-cancel-left1*[**where** *c*=*a* **and** *b*=*k*∗*k′*]
have *kk′*:*k*∗*k′* = *1* **using** *anz* **by** (*simp add*: *mult-assoc*)
hence *k* = *1* ∧ *k′* = *1* ∨ *k* = −*1* ∧ *k′* = −*1* **by** (*simp add*: *zmult-eq-1-iff*)
thus *?thesis* **using** *k k′* **by** *auto*
qed

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* ==> *m* < *n* ==> ¬ *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 *zmult-div-cancel*: (*n*::*int*) $*$ (*m div n*) = *m* − (*m mod n*)
  using *zmod-zdiv-equality*[**where** *a*=*m* **and** *b*=*n*]
  by (*simp add*: *ring-simps*)

lemma *zdvd-mult-div-cancel*:(*n*::*int*) *dvd m* ⟹ *n* $*$ (*m div n*) = *m*
apply (*subgoal-tac m mod n* = *0*)
 apply (*simp add*: *zmult-div-cancel*)
apply (*simp only*: *zdvd-iff-zmod-eq-0*)
done

lemma *zdvd-mult-cancel*: **assumes** *d*:*k* $*$ *m dvd k* $*$ *n* **and** *kz*:*k* ≠ (*0*::*int*)
  shows *m dvd n*
proof−
  from *d* **obtain** *h* **where** *h*: *k*$*$*n* = *k*$*$*m* $*$ *h* **unfolding** *dvd-def* **by** *blast*
  {assume *n* ≠ *m*$*$*h* **hence** *k*$*$ *n* ≠ *k*$*$ (*m*$*$*h*) **using** *kz* **by** *simp*
   with *h* **have** *False* **by** (*simp add*: *mult-assoc*)**}**
  hence *n* = *m* $*$ *h* **by** *blast*
  thus *?thesis* **by** *blast*
qed

lemma *zdvd-zmult-cancel-disj*[*simp*]:
  (*k*$*$*m*) *dvd* (*k*$*$*n*) = (*k*=*0* | *m dvd* (*n*::*int*))
by (*auto simp*: *zdvd-zmult-mono dest*: *zdvd-mult-cancel*)

theorem *ex-nat*: (∃ *x*::*nat*. *P x*) = (∃ *x*::*int*. *0* <= *x* ∧ *P* (*nat x*))
apply (*simp split add*: *split-nat*)
apply (*rule iffI*)
apply (*erule exE*)
apply (*rule-tac x* = *int x* **in** *exI*)
apply *simp*
apply (*erule exE*)
apply (*rule-tac x* = *nat x* **in** *exI*)
apply (*erule conjE*)

apply (*erule-tac x = nat x* **in** *allE*)
apply *simp*
done

theorem *zdvd-int*: (*x dvd y*) = (*int x dvd int y*)
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*: *neq0-conv 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

lemma *zdvd1-eq*[*simp*]: (*x*::*int*) *dvd 1* = (|*x*| = *1*)
proof
assume *d*: *x dvd 1* **hence** *int* (*nat* |*x*|) *dvd int* (*nat 1*) **by** (*simp add*: *zdvd-abs1*)
hence *nat* |*x*| *dvd 1* **by** (*simp add*: *zdvd-int*)
hence *nat* |*x*| = *1* **by** *simp*
thus |*x*| = *1* **by** (*cases x* < *0*, *auto*)
next
assume |*x*|=*1* **thus** *x dvd 1*
by(*cases x* < *0*,*simp-all add*: *minus-equation-iff zdvd-iff-zmod-eq-0*)
qed
lemma *zdvd-mult-cancel1*:
assumes *mp*:*m* $\neq$(*0*::*int*) **shows** (*m* $*$ *n dvd m*) = (|*n*| = *1*)
proof
assume *n1*: |*n*| = *1* **thus** *m* $*$ *n dvd m*
by (*cases n* >*0*, *auto simp add*: *zdvd-zminus2-iff minus-equation-iff*)
next
assume *H*: *m* $*$ *n dvd m* **hence** *H2*: *m* $*$ *n dvd m* $*$ *1* **by** *simp*
from *zdvd-mult-cancel*[*OF H2 mp*] **show** |*n*| = *1* **by** (*simp only*: *zdvd1-eq*)
qed

lemma *int-dvd-iff*: (*int m dvd z*) = (*m dvd nat* (*abs z*))
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*: [| $z\ dvd\ n$; $0 < n$ |] ==> $z \le (n::int)$
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

32.20 Integer Powers

instance *int* :: *power* **..**

primrec
$p\ \hat{}\ 0 = 1$
$p\ \hat{}\ (Suc\ n) = (p::int) * (p\ \hat{}\ n)$

instance *int* :: *recpower*
proof
fix z :: *int*
fix n :: *nat*

show $z\hat{}0 = 1$ **by** *simp*
show $z\hat{}(Suc\ n) = z * (z\hat{}n)$ **by** *simp*
qed

lemma *of-int-power*:
$of\text{-}int\ (z\ \hat{}\ n) = (of\text{-}int\ z\ \hat{}\ n :: 'a::\{recpower,\ ring\text{-}1\})$
by (*induct n*) (*simp-all add*: *power-Suc*)

lemma *zpower-zmod*: $((x::int)\ mod\ m)\hat{}y\ mod\ m = x\hat{}y\ mod\ m$
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*: $x\hat{}(y+z) = ((x\hat{}y)*(x\hat{}z)::int)$
by (*rule Power.power-add*)

lemma *zpower-zpower*: $(x\hat{}y)\hat{}z = (x\hat{}(y*z)::int)$
by (*rule Power.power-mult* [*symmetric*])

lemma *zero-less-zpower-abs-iff* [*simp*]:
$(0 < (abs\ x)\hat{}n) = (x \neq (0::int) \mid n=0)$
apply (*induct n*)
apply (*auto simp add*: *zero-less-mult-iff*)
done

lemma *zero-le-zpower-abs* [*simp*]: $(0::int) <= (abs\ x)\hat{}n$
apply (*induct n*)
apply (*auto simp add*: *zero-le-mult-iff*)
done

lemma *int-power*: $int\ (m\hat{}n) = (int\ m)\ \hat{}\ n$
by (*rule of-nat-power*)

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 of-nat-mult*)
done

lemma *zmod-int*: $int\ (a\ mod\ b) = (int\ a)\ mod\ (int\ b)$
apply (*subst split-mod*, *auto*)
apply (*subst split-zmod*, *auto*)

```
apply (rule-tac a=int (b * i) + int j and b=int b and q=int i and q'=ia
      in unique-remainder)
apply (auto simp add: IntDiv.quorem-def of-nat-mult)
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
```

by Brian Huffman

```
lemma zminus-zmod: - ((x::int) mod m) mod m = - x mod m
by (simp only: zmod-zminus1-eq-if mod-mod-trivial)

lemma zdiff-zmod-left: (x mod m - y) mod m = (x - y) mod (m::int)
by (simp only: diff-def zmod-zadd-left-eq [symmetric])

lemma zdiff-zmod-right: (x - y mod m) mod m = (x - y) mod (m::int)
proof -
  have (x + - (y mod m) mod m) mod m = (x + - y mod m) mod m
    by (simp only: zminus-zmod)
  hence (x + - (y mod m)) mod m = (x + - y) mod m
    by (simp only: zmod-zadd-right-eq [symmetric])
  thus (x - y mod m) mod m = (x - y) mod m
    by (simp only: diff-def)
qed

lemmas zmod-simps =
  IntDiv.zmod-zadd-left-eq  [symmetric]
  IntDiv.zmod-zadd-right-eq [symmetric]
  IntDiv.zmod-zmult1-eq     [symmetric]
  IntDiv.zmod-zmult1-eq'    [symmetric]
  IntDiv.zpower-zmod
  zminus-zmod zdiff-zmod-left zdiff-zmod-right
```

code generator setup

```
code-modulename SML
  IntDiv Integer

code-modulename OCaml
  IntDiv Integer

code-modulename Haskell
  IntDiv Integer
```

end

33 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*
nat-number-of-def [*code inline*]: *number-of v* == *nat* (*number-of* (*v*::*int*)) **..**

abbreviation (*xsymbols*)
square :: *'a*::*power* => *'a* ((-^{2}) [*1000*] *999*) **where**
x^2 == *x*ˆ*2*

notation (*latex* **output**)
square ((-^{2}) [*1000*] *999*)

notation (*HTML* **output**)
square ((-^{2}) [*1000*] *999*)

33.1 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 of-nat-eq-iff* [**where** *'a=int*, *THEN iffD1*], *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*
of-nat-add [*symmetric*] *of-nat-mult* [*symmetric*]
del: *of-nat-add of-nat-mult*)
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 int-int-eq* [*THEN iffD1*], *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*
of-nat-add [*symmetric*] *of-nat-mult* [*symmetric*]
del: *of-nat-add of-nat-mult*)
done

Suggested by Matthias Daum

lemma *int-div-less-self*: $[\![0 < x;\ 1 < k]\!] \Longrightarrow x\ div\ k < (x::int)$
apply (*subgoal-tac nat x div nat k* < *nat x*)
apply (*simp* (*asm-lr*) *add*: *nat-div-distrib* [*symmetric*])
apply (*rule Divides.div-less-dividend*, *simp-all*)
done

33.2 Function *int*: Coercion from Type *nat* to *int*

lemma *int-nat-number-of* [*simp*]:
int (*number-of v*) =
(*if neg* (*number-of v* :: *int*) *then 0*
else (*number-of v* :: *int*))
by (*simp del*: *nat-number-of*
add: *neg-nat nat-number-of-def not-neg-nat add-assoc*)

33.2.1 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* (*Numeral.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* (*Numeral.succ v*))
apply (*cut-tac n* = *0* **in** *Suc-nat-number-of-add*)
apply (*simp cong del*: *if-weak-cong*)
done

33.2.2 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 (*v* + *v'*))
by (*force dest*!: *neg-nat*
    simp del: *nat-number-of*
    simp add: *nat-number-of-def nat-add-distrib* [*symmetric*])

33.2.3 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*)
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* (*v* + *uminus 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*)

33.2.4 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* ($v * v'$))
by (*force dest*!: *neg-nat*
   simp del: *nat-number-of*
   simp add: *nat-number-of-def nat-mult-distrib* [*symmetric*])

33.2.5 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*])

33.2.6 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*])

33.2.7 Divisibility

lemmas *dvd-eq-mod-eq-0-number-of* =
 dvd-eq-mod-eq-0 [*of number-of x number-of y*, *standard*]

declare *dvd-eq-mod-eq-0-number-of* [*simp*]

ML
⟪
val nat-number-of-def = *thmnat-number-of-def*;

val nat-number-of = thmnat-number-of;
val nat-numeral-0-eq-0 = thmnat-numeral-0-eq-0;
val nat-numeral-1-eq-1 = thmnat-numeral-1-eq-1;
val numeral-1-eq-Suc-0 = thmnumeral-1-eq-Suc-0;
val numeral-2-eq-2 = thmnumeral-2-eq-2;
val nat-div-distrib = thmnat-div-distrib;
val nat-mod-distrib = thmnat-mod-distrib;
val int-nat-number-of = thmint-nat-number-of;
val Suc-nat-eq-nat-zadd1 = thmSuc-nat-eq-nat-zadd1;
val Suc-nat-number-of-add = thmSuc-nat-number-of-add;
val Suc-nat-number-of = thmSuc-nat-number-of;
val add-nat-number-of = thmadd-nat-number-of;
val diff-nat-eq-if = thmdiff-nat-eq-if;
val diff-nat-number-of = thmdiff-nat-number-of;
val mult-nat-number-of = thmmult-nat-number-of;
val div-nat-number-of = thmdiv-nat-number-of;
val mod-nat-number-of = thmmod-nat-number-of;
⟫

33.3 Comparisons

33.3.1 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* (*v* + *uminus 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

33.3.2 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* (*uminus v'*) :: *int*)
else neg (*number-of* (*v* + *uminus 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 add*: *Pls-def*)

lemmas *numerals* = *nat-numeral-0-eq-0 nat-numeral-1-eq-1 numeral-2-eq-2*

33.4 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*: $(a::'a::recpower)^2 = a * a$
  by (*simp add*: *numeral-2-eq-2 Power.power-Suc*)

lemma *zero-power2* [*simp*]: $(0::'a::\{semiring\text{-}1,recpower\})^2 = 0$
  by (*simp add*: *power2-eq-square*)

lemma *one-power2* [*simp*]: $(1::'a::\{semiring\text{-}1,recpower\})^2 = 1$
  by (*simp add*: *power2-eq-square*)

lemma *power3-eq-cube*: $(x::'a::recpower)$ ^ $3 = x * x * x$
  apply (*subgoal-tac* 3 = *Suc* (*Suc* (*Suc* 0)))
  apply (*erule ssubst*)
  apply (*simp add*: *power-Suc mult-ac*)
  apply (*unfold nat-number-of-def*)
  apply (*subst nat-eq-iff*)
  apply *simp*
done

Squares of literal numerals will be evaluated.

lemmas *power2-eq-square-number-of* =
    power2-eq-square [*of number-of w*, *standard*]
declare *power2-eq-square-number-of* [*simp*]

lemma *zero-le-power2*[*simp*]: $0 \leq (a^2::'a::\{ordered\text{-}idom,recpower\})$
  by (*simp add*: *power2-eq-square*)

lemma *zero-less-power2*[*simp*]:
    $(0 < a^2) = (a \neq (0::'a::\{ordered\text{-}idom,recpower\}))$
  by (*force simp add*: *power2-eq-square zero-less-mult-iff linorder-neq-iff*)

lemma *power2-less-0*[*simp*]:
  fixes $a :: 'a::\{ordered\text{-}idom,recpower\}$
  shows ~ $(a^2 < 0)$
by (*force simp add*: *power2-eq-square mult-less-0-iff*)

lemma *zero-eq-power2*[*simp*]:

$(a^2 = 0) = (a = (0::'a::\{ordered\text{-}idom,recpower\}))$
by (*force simp add*: *power2-eq-square mult-eq-0-iff*)

lemma *abs-power2*[*simp*]:
$abs(a^2) = (a^2::'a::\{ordered\text{-}idom,recpower\})$
by (*simp add*: *power2-eq-square abs-mult abs-mult-self*)

lemma *power2-abs*[*simp*]:
$(abs\ a)^2 = (a^2::'a::\{ordered\text{-}idom,recpower\})$
by (*simp add*: *power2-eq-square abs-mult-self*)

lemma *power2-minus*[*simp*]:
$(-\ a)^2 = (a^2::'a::\{comm\text{-}ring\text{-}1,recpower\})$
by (*simp add*: *power2-eq-square*)

lemma *power2-le-imp-le*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 \le y^2;\ 0 \le y]\!] \Longrightarrow x \le y$
unfolding *numeral-2-eq-2* **by** (*rule power-le-imp-le-base*)

lemma *power2-less-imp-less*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 < y^2;\ 0 \le y]\!] \Longrightarrow x < y$
by (*rule power-less-imp-less-base*)

lemma *power2-eq-imp-eq*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 = y^2;\ 0 \le x;\ 0 \le y]\!] \Longrightarrow x = y$
unfolding *numeral-2-eq-2* **by** (*erule* (*2*) *power-eq-imp-eq-base*, *simp*)

lemma *power-minus1-even*[*simp*]: $(-\ 1)\ \hat{}\ (2*n) = (1::'a::\{comm\text{-}ring\text{-}1,recpower\})$
apply (*induct n*)
apply (*auto simp add*: *power-Suc power-add*)
done

lemma *power-even-eq*: $(a::'a::recpower)\ \hat{}\ (2*n) = (a\hat{}n)\hat{}2$
by (*subst mult-commute*) (*simp add*: *power-mult*)

lemma *power-odd-eq*: $(a::int)\ \hat{}\ Suc(2*n) = a * (a\hat{}n)\hat{}2$
by (*simp add*: *power-even-eq*)

lemma *power-minus-even* [*simp*]:
$(-a)\ \hat{}\ (2*n) = (a::'a::\{comm\text{-}ring\text{-}1,recpower\})\ \hat{}\ (2*n)$
by (*simp add*: *power-minus1-even power-minus* [*of a*])

lemma *zero-le-even-power'*[*simp*]:
$0 \le (a::'a::\{ordered\text{-}idom,recpower\})\ \hat{}\ (2*n)$
proof (*induct n*)
case *0*

```
    show ?case by (simp add: zero-le-one)
next
  case (Suc n)
    have a ^ (2 * Suc n) = (a*a) * a ^ (2*n)
      by (simp add: mult-ac power-add power2-eq-square)
    thus ?case
      by (simp add: prems zero-le-mult-iff)
qed

lemma odd-power-less-zero:
    (a::'a::{ordered-idom,recpower}) < 0 ==> a ^ Suc(2*n) < 0
proof (induct n)
  case 0
  then show ?case by (simp add: Power.power-Suc)
next
  case (Suc n)
  have a ^ Suc (2 * Suc n) = (a*a) * a ^ 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 ≤ a  ^ Suc(2*n) ==> 0 ≤ (a::'a::{ordered-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
```

33.4.1 Nat

```
lemma Suc-pred': 0 < n ==> n = Suc(n - 1)
by (simp add: numerals)


lemmas expand-Suc = Suc-pred' [of number-of v, standard]
```

33.4.2 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* ˆ *m* :: *nat*) = (*if m=0 then 1 else p* ∗ (*p* ˆ (*m* − *1*)))
apply (*case-tac m*)
apply (*simp-all add*: *numerals*)
done

33.5 Comparisons involving (0::nat)

Simplification already does $n < (0::'a)$, $n \le (0::'a)$ and $(0::'a) \le n$.

lemma *eq-number-of-0* [*simp*]:
(*number-of v* = (*0*::*nat*)) =
(*if neg* (*number-of v* :: *int*) *then True else iszero* (*number-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-of v*) =
(*if neg* (*number-of v* :: *int*) *then True else iszero* (*number-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* (*uminus v*) :: *int*)
by (*simp del*: *nat-numeral-0-eq-0 add*: *nat-numeral-0-eq-0* [*symmetric*] *Pls-def*)

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*)

33.6 Comparisons involving *Suc*

lemma *eq-number-of-Suc* [*simp*]:
(*number-of v* = *Suc n*) =
(*let pv* = *number-of* (*Numeral.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 (Numeral.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 (Numeral.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 (Numeral.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 (Numeral.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 (Numeral.pred v) in
      if neg pv then False else n <= nat pv)
by (simp add: Let-def less-number-of-Suc linorder-not-less [symmetric])


lemma 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-left
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*

33.7 Max and Min Combined with *Suc*

lemma *max-number-of-Suc* [*simp*]:
max (*Suc n*) (*number-of v*) =
(*let pv* = *number-of* (*Numeral.pred v*) *in*
if neg pv then Suc n else Suc(*max n* (*nat pv*)))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
split add: *split-if nat.split*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply *auto*

done

lemma *max-Suc-number-of* [*simp*]:
  max (*number-of v*) (*Suc n*) =
    (*let pv* = *number-of* (*Numeral.pred v*) *in*
     if neg pv then Suc n else Suc(*max* (*nat pv*) *n*))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
    split add: *split-if nat.split*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply *auto*
done

lemma *min-number-of-Suc* [*simp*]:
  min (*Suc n*) (*number-of v*) =
    (*let pv* = *number-of* (*Numeral.pred v*) *in*
     if neg pv then 0 else Suc(*min n* (*nat pv*)))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
    split add: *split-if nat.split*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply *auto*
done

lemma *min-Suc-number-of* [*simp*]:
  min (*number-of v*) (*Suc n*) =
    (*let pv* = *number-of* (*Numeral.pred v*) *in*
     if neg pv then 0 else Suc(*min* (*nat pv*) *n*))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
    split add: *split-if nat.split*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply *auto*
done

33.8 Literal arithmetic involving powers

lemma *nat-power-eq*: (*0*::*int*) <= *z* ==> *nat* (*z*^*n*) = *nat z* ^ *n*
apply (*induct n*)
apply (*simp-all* (*no-asm-simp*) *add*: *nat-mult-distrib*)
done

lemma *power-nat-number-of*:
  (*number-of v* :: *nat*) ^ *n* =
   (*if neg* (*number-of v* :: *int*) *then 0*^*n else nat* ((*number-of v* :: *int*) ^ *n*))
by (*simp only*: *simp-thms neg-nat not-neg-eq-ge-0 nat-number-of-def nat-power-eq*
   split add: *split-if cong*: *imp-cong*)

lemmas *power-nat-number-of-number-of* = *power-nat-number-of* [*of - number-of w*, *standard*]
declare *power-nat-number-of-number-of* [*simp*]

For arbitrary rings

lemma *power-number-of-even*:
  fixes z :: *'a*::{*number-ring*,*recpower*}
  shows z ^ *number-of* (*w BIT bit.B0*) = (*let* $w = z$ ^ (*number-of* w) *in* $w * w$)
unfolding *Let-def nat-number-of-def number-of-BIT bit.cases*
apply (*rule-tac* x = *number-of* w **in** *spec*, *clarify*)
apply (*case-tac* (*0*::*int*) <= x)
apply (*auto simp add*: *nat-mult-distrib power-even-eq power2-eq-square*)
done

lemma *power-number-of-odd*:
  fixes z :: *'a*::{*number-ring*,*recpower*}
  shows z ^ *number-of* (*w BIT bit.B1*) = (*if* (*0*::*int*) <= *number-of* w
    then (*let* $w = z$ ^ (*number-of* w) *in* $z * w * w$) *else 1*)
unfolding *Let-def nat-number-of-def number-of-BIT bit.cases*
apply (*rule-tac* x = *number-of* w **in** *spec*, *auto*)
apply (*simp only*: *nat-add-distrib nat-mult-distrib*)
apply *simp*
apply (*auto simp add*: *nat-add-distrib nat-mult-distrib power-even-eq power2-eq-square neg-nat power-Suc*)
done

lemmas *zpower-number-of-even* = *power-number-of-even* [**where** *'a*=*int*]
lemmas *zpower-number-of-odd* = *power-number-of-odd* [**where** *'a*=*int*]

lemmas *power-number-of-even-number-of* [*simp*] =
  power-number-of-even [*of number-of v*, *standard*]

lemmas *power-number-of-odd-number-of* [*simp*] =
  power-number-of-odd [*of number-of v*, *standard*]

ML
⟪
val numerals = *thmsnumerals*;
val numeral-ss = *simpset*() *addsimps numerals*;

val nat-bin-arith-setup =
 LinArith.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*]})
⟫

declaration ⟪ *K nat-bin-arith-setup* ⟫

declare *split-div*[*of - - number-of k*, *standard*, *arith-split*]
declare *split-mod*[*of - - number-of k*, *standard*, *arith-split*]

lemma *nat-number-of-Pls*: *Numeral0* = (*0*::*nat*)
  by (*simp add*: *number-of-Pls nat-number-of-def*)

lemma *nat-number-of-Min*: *number-of Numeral.Min* = (*0*::*nat*)
  apply (*simp only*: *number-of-Min nat-number-of-def nat-zminus-int*)
  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 power-Suc*)

lemma *power-m1-odd*: (-1) ˆ $Suc(2{*}n) = (-1{::}'a{::}\{number\text{-}ring,recpower\})$
by (*simp add*: *power-mult power-Suc*)

33.9 Literal arithmetic and *of-nat*

lemma *of-nat-double*:
$0 \le x$ ==> *of-nat* (*nat* ($2 * x$)) = *of-nat* (*nat* x) + *of-nat* (*nat* x)
by (*simp only*: *mult-2 nat-add-distrib of-nat-add*)

lemma *nat-numeral-m1-eq-0*: $-1 = (0{::}nat)$
by (*simp only*: *nat-number-of-def*)

lemma *of-nat-number-of-lemma*:
of-nat (*number-of v* :: *nat*) =
(*if* $0 \le$ (*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*)

33.10 Lemmas for the Combination and Cancellation Simprocs

lemma *nat-number-of-add-left*:
number-of v + (*number-of v'* + (k::*nat*)) =
(*if neg* (*number-of v* :: *int*) *then number-of v'* + k
else if neg (*number-of v'* :: *int*) *then number-of v* + k
else number-of (v + v') + k)
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 (v * v') * k)
by *simp*

33.10.1 For *combine-numerals*

lemma *left-add-mult-distrib*: $i{*}u + (j{*}u + k) = (i{+}j){*}u + (k{::}nat)$
by (*simp add*: *add-mult-distrib*)

33.10.2 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*)

33.10.3 For *cancel-numeral-factors*

lemma *nat-mult-le-cancel1*: $(0::nat) < k ==> (k*m <= k*n) = (m<=n)$
by *auto*

lemma *nat-mult-less-cancel1*: $(0::nat) < k ==> (k*m < k*n) = (m<n)$
by *auto*

lemma *nat-mult-eq-cancel1*: $(0::nat) < k ==> (k*m = k*n) = (m=n)$
by *auto*

lemma *nat-mult-div-cancel1*: $(0::nat) < k ==> (k*m)\ div\ (k*n) = (m\ div\ n)$
by *auto*

lemma *nat-mult-dvd-cancel-disj*[*simp*]:
$(k*m)\ dvd\ (k*n) = (k=0 \mid m\ dvd\ (n::nat))$
by(*auto simp*: *dvd-eq-mod-eq-0 mod-mult-distrib2*[*symmetric*])

lemma *nat-mult-dvd-cancel1*: $0 < k \Longrightarrow (k*m)\ dvd\ (k*n::nat) = (m\ dvd\ n)$

by(*auto*)

33.10.4 For *cancel-factor*

lemma *nat-mult-le-cancel-disj*: $(k*m <= k*n) = ((0::nat) < k --> m<=n)$
by *auto*

lemma *nat-mult-less-cancel-disj*: $(k*m < k*n) = ((0::nat) < k \& m<n)$
by *auto*

lemma *nat-mult-eq-cancel-disj*: $(k*m = k*n) = (k = (0::nat) \mid m=n)$
by *auto*

lemma *nat-mult-div-cancel-disj*[*simp*]:
$(k*m)$ *div* $(k*n) =$ (*if* $k = (0::nat)$ *then* 0 *else* m *div* n)
by (*simp add*: *nat-mult-div-cancel1*)

33.11 legacy ML bindings

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 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;
⟫

end

34 Groebner-Basis: Semiring normalization and Groebner Bases

theory *Groebner-Basis*

imports *NatBin*
uses
Tools/Groebner-Basis/misc.ML
Tools/Groebner-Basis/normalizer-data.ML
(*Tools/Groebner-Basis/normalizer.ML*)
(*Tools/Groebner-Basis/groebner.ML*)
begin

34.1 Semiring normalization

setup *NormalizerData.setup*

locale *gb-semiring* =
fixes *add mul pwr r0 r1*
assumes *add-a*:(*add x* (*add y z*) = *add* (*add x y*) *z*)
and *add-c*: *add x y* = *add y x* **and** *add-0*:*add r0 x* = *x*
and *mul-a*:*mul x* (*mul y z*) = *mul* (*mul x y*) *z* **and** *mul-c*:*mul x y* = *mul y x*
and *mul-1*:*mul r1 x* = *x* **and** *mul-0*:*mul r0 x* = *r0*
and *mul-d*:*mul x* (*add y z*) = *add* (*mul x y*) (*mul x z*)
and *pwr-0*:*pwr x 0* = *r1* **and** *pwr-Suc*:*pwr x* (*Suc n*) = *mul x* (*pwr x n*)
begin

lemma *mul-pwr*:*mul* (*pwr x p*) (*pwr x q*) = *pwr x* (*p* + *q*)
proof (*induct p*)
case *0*
then show *?case* **by** (*auto simp add*: *pwr-0 mul-1*)
next
case *Suc*
from *this* [*symmetric*] **show** *?case*
by (*auto simp add*: *pwr-Suc mul-1 mul-a*)
qed

lemma *pwr-mul*: *pwr* (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*)
proof (*induct q arbitrary*: *x y*, *auto simp add*:*pwr-0 pwr-Suc mul-1*)
fix *q x y*
assume $\bigwedge$*x y*. *pwr* (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*)
have *mul* (*mul x y*) (*mul* (*pwr x q*) (*pwr y q*)) = *mul x* (*mul y* (*mul* (*pwr x q*) (*pwr y q*)))
by (*simp add*: *mul-a*)
also have ... = (*mul* (*mul y* (*mul* (*pwr y q*) (*pwr x q*))) *x*) **by** (*simp add*: *mul-c*)
also have ... = (*mul* (*mul y* (*pwr y q*)) (*mul* (*pwr x q*) *x*)) **by** (*simp add*: *mul-a*)
finally show *mul* (*mul x y*) (*mul* (*pwr x q*) (*pwr y q*)) =
mul (*mul x* (*pwr x q*)) (*mul y* (*pwr y q*)) **by** (*simp add*: *mul-c*)
qed

lemma *pwr-pwr*: *pwr* (*pwr x p*) *q* = *pwr x* (*p* ∗ *q*)
proof (*induct p arbitrary*: *q*)
case *0*

show *?case* **using** *pwr-Suc mul-1 pwr-0* **by** (*induct q*) *auto*
next
case *Suc*
thus *?case* **by** (*auto simp add*: *mul-pwr* [*symmetric*] *pwr-mul pwr-Suc*)
qed

34.1.1 Declaring the abstract theory

lemma *semiring-ops*:
includes *meta-term-syntax*
shows *TERM* (*add x y*) **and** *TERM* (*mul x y*) **and** *TERM* (*pwr x n*)
and *TERM r0* **and** *TERM r1*
by *rule+*

lemma *semiring-rules*:
add (*mul a m*) (*mul b m*) = *mul* (*add a b*) *m*
add (*mul a m*) *m* = *mul* (*add a r1*) *m*
add m (*mul a m*) = *mul* (*add a r1*) *m*
add m m = *mul* (*add r1 r1*) *m*
add r0 a = *a*
add a r0 = *a*
mul a b = *mul b a*
mul (*add a b*) *c* = *add* (*mul a c*) (*mul b c*)
mul r0 a = *r0*
mul a r0 = *r0*
mul r1 a = *a*
mul a r1 = *a*
mul (*mul lx ly*) (*mul rx ry*) = *mul* (*mul lx rx*) (*mul ly ry*)
mul (*mul lx ly*) (*mul rx ry*) = *mul lx* (*mul ly* (*mul rx ry*))
mul (*mul lx ly*) (*mul rx ry*) = *mul rx* (*mul* (*mul lx ly*) *ry*)
mul (*mul lx ly*) *rx* = *mul* (*mul lx rx*) *ly*
mul (*mul lx ly*) *rx* = *mul lx* (*mul ly rx*)
mul lx (*mul rx ry*) = *mul* (*mul lx rx*) *ry*
mul lx (*mul rx ry*) = *mul rx* (*mul lx ry*)
add (*add a b*) (*add c d*) = *add* (*add a c*) (*add b d*)
add (*add a b*) *c* = *add a* (*add b c*)
add a (*add c d*) = *add c* (*add a d*)
add (*add a b*) *c* = *add* (*add a c*) *b*
add a c = *add c a*
add a (*add c d*) = *add* (*add a c*) *d*
mul (*pwr x p*) (*pwr x q*) = *pwr x* (*p* + *q*)
mul x (*pwr x q*) = *pwr x* (*Suc q*)
mul (*pwr x q*) *x* = *pwr x* (*Suc q*)
mul x x = *pwr x 2*
pwr (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*)
pwr (*pwr x p*) *q* = *pwr x* (*p* * *q*)
pwr x 0 = *r1*
pwr x 1 = *x*
mul x (*add y z*) = *add* (*mul x y*) (*mul x z*)

pwr x (*Suc q*) = *mul x* (*pwr x q*)
pwr x (*2∗n*) = *mul* (*pwr x n*) (*pwr x n*)
pwr x (*Suc* (*2∗n*)) = *mul x* (*mul* (*pwr x n*) (*pwr x n*))
proof −
show *add* (*mul a m*) (*mul b m*) = *mul* (*add a b*) *m* **using** *mul-d mul-c* **by** *simp*
next show*add* (*mul a m*) *m* = *mul* (*add a r1*) *m* **using** *mul-d mul-c mul-1* **by** *simp*
next show *add m* (*mul a m*) = *mul* (*add a r1*) *m* **using** *mul-c mul-d mul-1 add-c* **by** *simp*
next show *add m m* = *mul* (*add r1 r1*) *m* **using** *mul-c mul-d mul-1* **by** *simp*
next show *add r0 a* = *a* **using** *add-0* **by** *simp*
next show *add a r0* = *a* **using** *add-0 add-c* **by** *simp*
next show *mul a b* = *mul b a* **using** *mul-c* **by** *simp*
next show *mul* (*add a b*) *c* = *add* (*mul a c*) (*mul b c*) **using** *mul-c mul-d* **by** *simp*
next show *mul r0 a* = *r0* **using** *mul-0* **by** *simp*
next show *mul a r0* = *r0* **using** *mul-0 mul-c* **by** *simp*
next show *mul r1 a* = *a* **using** *mul-1* **by** *simp*
next show *mul a r1* = *a* **using** *mul-1 mul-c* **by** *simp*
next show *mul* (*mul lx ly*) (*mul rx ry*) = *mul* (*mul lx rx*) (*mul ly ry*)
using *mul-c mul-a* **by** *simp*
next show *mul* (*mul lx ly*) (*mul rx ry*) = *mul lx* (*mul ly* (*mul rx ry*))
using *mul-a* **by** *simp*
next
have *mul* (*mul lx ly*) (*mul rx ry*) = *mul* (*mul rx ry*) (*mul lx ly*) **by** (*rule mul-c*)
also have ... = *mul rx* (*mul ry* (*mul lx ly*)) **using** *mul-a* **by** *simp*
finally
show *mul* (*mul lx ly*) (*mul rx ry*) = *mul rx* (*mul* (*mul lx ly*) *ry*)
using *mul-c* **by** *simp*
next show *mul* (*mul lx ly*) *rx* = *mul* (*mul lx rx*) *ly* **using** *mul-c mul-a* **by** *simp*
next
show *mul* (*mul lx ly*) *rx* = *mul lx* (*mul ly rx*) **by** (*simp add*: *mul-a*)
next show *mul lx* (*mul rx ry*) = *mul* (*mul lx rx*) *ry* **by** (*simp add*: *mul-a*)
next show *mul lx* (*mul rx ry*) = *mul rx* (*mul lx ry*) **by** (*simp add*: *mul-a*,*simp add*: *mul-c*)
next show *add* (*add a b*) (*add c d*) = *add* (*add a c*) (*add b d*)
using *add-c add-a* **by** *simp*
next show *add* (*add a b*) *c* = *add a* (*add b c*) **using** *add-a* **by** *simp*
next show *add a* (*add c d*) = *add c* (*add a d*)
apply (*simp add*: *add-a*) **by** (*simp only*: *add-c*)
next show *add* (*add a b*) *c* = *add* (*add a c*) *b* **using** *add-a add-c* **by** *simp*
next show *add a c* = *add c a* **by** (*rule add-c*)
next show *add a* (*add c d*) = *add* (*add a c*) *d* **using** *add-a* **by** *simp*
next show *mul* (*pwr x p*) (*pwr x q*) = *pwr x* (*p* + *q*) **by** (*rule mul-pwr*)
next show *mul x* (*pwr x q*) = *pwr x* (*Suc q*) **using** *pwr-Suc* **by** *simp*
next show *mul* (*pwr x q*) *x* = *pwr x* (*Suc q*) **using** *pwr-Suc mul-c* **by** *simp*
next show *mul x x* = *pwr x 2* **by** (*simp add*: *nat-number pwr-Suc pwr-0 mul-1 mul-c*)
next show *pwr* (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*) **by** (*rule pwr-mul*)

next show *pwr* (*pwr x p*) *q* = *pwr x* (*p* ∗ *q*) **by** (*rule pwr-pwr*)
next show *pwr x 0* = *r1* **using** *pwr-0* **.**
next show *pwr x 1* = *x* **by** (*simp add*: *nat-number pwr-Suc pwr-0 mul-1 mul-c*)
next show *mul x* (*add y z*) = *add* (*mul x y*) (*mul x z*) **using** *mul-d* **by** *simp*
next show *pwr x* (*Suc q*) = *mul x* (*pwr x q*) **using** *pwr-Suc* **by** *simp*
next show *pwr x* (*2* ∗ *n*) = *mul* (*pwr x n*) (*pwr x n*) **by** (*simp add*: *nat-number mul-pwr*)
next show *pwr x* (*Suc* (*2* ∗ *n*)) = *mul x* (*mul* (*pwr x n*) (*pwr x n*))
  by (*simp add*: *nat-number pwr-Suc mul-pwr*)
qed

lemma *axioms* [*normalizer*
  semiring ops: *semiring-ops*
  semiring rules: *semiring-rules*]:
 gb-semiring add mul pwr r0 r1 **by** *fact*

end

interpretation *class-semiring*: *gb-semiring*
  [*op* + *op* ∗ *op* ^ *0*::*'a*::{*comm-semiring-1*, *recpower*} *1*]
 by *unfold-locales* (*auto simp add*: *ring-simps power-Suc*)

lemmas *nat-arith* =
 add-nat-number-of diff-nat-number-of mult-nat-number-of eq-nat-number-of less-nat-number-of

lemma *not-iszero-Numeral1*: ¬ *iszero* (*Numeral1*::*'a*::*number-ring*)
 by (*simp add*: *numeral-1-eq-1*)
lemmas *comp-arith* = *Let-def arith-simps nat-arith rel-simps if-False*
 if-True add-0 add-Suc add-number-of-left mult-number-of-left
 numeral-1-eq-1[*symmetric*] *Suc-eq-add-numeral-1*
 numeral-0-eq-0[*symmetric*] *numerals*[*symmetric*] *not-iszero-1*
 iszero-number-of-1 iszero-number-of-0 nonzero-number-of-Min
 iszero-number-of-Pls iszero-0 not-iszero-Numeral1

lemmas *semiring-norm* = *comp-arith*

ML ⟪
local

open Conv;

fun numeral-is-const ct =
 can HOLogic.dest-number (*Thm.term-of ct*);

fun int-of-rat x =
 (*case Rat.quotient-of-rat x of* (*i*, *1*) => *i*
 | - => *error int-of-rat*: *bad int*);

val numeral-conv =
  Simplifier.rewrite (*HOL-basic-ss addsimps* @{*thms semiring-norm*}) *then-conv*
  Simplifier.rewrite (*HOL-basic-ss addsimps*
   (@{*thms numeral-1-eq-1*} @ @{*thms numeral-0-eq-0*} @ @{*thms numerals*(*1*−*2*)}));

in

fun normalizer-funs key =
  NormalizerData.funs key
   {*is-const* = *fn phi* => *numeral-is-const*,
    dest-const = *fn phi* => *fn ct* =>
      Rat.rat-of-int (*snd*
        (*HOLogic.dest-number* (*Thm.term-of ct*)
          handle TERM - => *error ring-dest-const*)),
    mk-const = *fn phi* => *fn cT* => *fn x* => *Numeral.mk-cnumber cT* (*int-of-rat*
x),
    conv = *fn phi* => *K numeral-conv*}

end
⟫

declaration ⟪ *normalizer-funs* @{*thm class-semiring.axioms*} ⟫

locale *gb-ring* = *gb-semiring* +
  fixes *sub* :: $'a \Rightarrow 'a \Rightarrow 'a$
    and *neg* :: $'a \Rightarrow 'a$
  assumes *neg-mul*: *neg x* = *mul* (*neg r1*) *x*
    and *sub-add*: *sub x y* = *add x* (*neg y*)
begin

lemma *ring-ops*:
  includes *meta-term-syntax*
  shows *TERM* (*sub x y*) **and** *TERM* (*neg x*) **.**

lemmas *ring-rules* = *neg-mul sub-add*

lemma *axioms* [*normalizer*
  semiring ops: *semiring-ops*
  semiring rules: *semiring-rules*
  ring ops: *ring-ops*
  ring rules: *ring-rules*]:
  gb-ring add mul pwr r0 r1 sub neg **by** *fact*

end

interpretation *class-ring*: *gb-ring* [*op* + *op* ∗ *op* ˆ
    0::*'a*::{*comm-semiring-1*,*recpower*,*number-ring*} *1 op* − *uminus*]

by *unfold-locales simp-all*

declaration ⟪ *normalizer-funs* @{*thm class-ring.axioms*} ⟫

use *Tools/Groebner-Basis/normalizer.ML*

method-setup *sring-norm* = ⟪
Method.ctxt-args (*fn ctxt* => *Method.SIMPLE-METHOD′* (*Normalizer.semiring-normalize-tac ctxt*))
⟫ *semiring normalizer*

locale *gb-field* = *gb-ring* +
fixes *divide* :: ′*a* ⇒ ′*a* ⇒ ′*a*
and *inverse*:: ′*a* ⇒ ′*a*
assumes *divide*: *divide x y* = *mul x* (*inverse y*)
and *inverse*: *inverse x* = *divide r1 x*
begin

lemma *axioms* [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*]:
gb-field add mul pwr r0 r1 sub neg divide inverse **by** *fact*

end

34.2 Groebner Bases

locale *semiringb* = *gb-semiring* +
assumes *add-cancel*: *add* (*x*::′*a*) *y* = *add x z* ⟷ *y* = *z*
and *add-mul-solve*: *add* (*mul w y*) (*mul x z*) =
add (*mul w z*) (*mul x y*) ⟷ *w* = *x* ∨ *y* = *z*
begin

lemma *noteq-reduce*: *a* ≠ *b* ∧ *c* ≠ *d* ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
proof−
have *a* ≠ *b* ∧ *c* ≠ *d* ⟷ ¬ (*a* = *b* ∨ *c* = *d*) **by** *simp*
also have ... ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
using *add-mul-solve* **by** *blast*
finally show *a* ≠ *b* ∧ *c* ≠ *d* ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
by *simp*
qed

lemma *add-scale-eq-noteq*: ⟦*r* ≠ *r0* ; (*a* = *b*) ∧ ~(*c* = *d*)⟧

$\Longrightarrow$ *add a* (*mul r c*) $\neq$ *add b* (*mul r d*)
proof(*clarify*)
  assume *nz*: *r*$\neq$ *r0* **and** *cnd*: *c*$\neq$*d*
    and *eq*: *add b* (*mul r c*) = *add b* (*mul r d*)
  hence *mul r c* = *mul r d* **using** *cnd add-cancel* **by** *simp*
  hence *add* (*mul r0 d*) (*mul r c*) = *add* (*mul r0 c*) (*mul r d*)
    using *mul-0 add-cancel* **by** *simp*
  thus *False* **using** *add-mul-solve nz cnd* **by** *simp*
qed

lemma *add-r0-iff*: *x* = *add x a* $\longleftrightarrow$ *a* = *r0*
proof−
  have *a* = *r0* $\longleftrightarrow$ *add x a* = *add x r0* **by** (*simp add*: *add-cancel*)
  thus *x* = *add x a* $\longleftrightarrow$ *a* = *r0* **by** (*auto simp add*: *add-c add-0*)
qed

declare *axioms* [*normalizer del*]

lemma *axioms* [*normalizer*
  semiring ops: *semiring-ops*
  semiring rules: *semiring-rules*
  idom rules: *noteq-reduce add-scale-eq-noteq*]:
  semiringb add mul pwr r0 r1 **by** *fact*

end

locale *ringb* = *semiringb* + *gb-ring* +
  assumes *subr0-iff*: *sub x y* = *r0* $\longleftrightarrow$ *x* = *y*
begin

declare *axioms* [*normalizer del*]

lemma *axioms* [*normalizer*
  semiring ops: *semiring-ops*
  semiring rules: *semiring-rules*
  ring ops: *ring-ops*
  ring rules: *ring-rules*
  idom rules: *noteq-reduce add-scale-eq-noteq*
  ideal rules: *subr0-iff add-r0-iff*]:
  ringb add mul pwr r0 r1 sub neg **by** *fact*

end

lemma *no-zero-divirors-neq0*:
  assumes *az*: (*a*::'*a*::*no-zero-divisors*) $\neq$ *0*
    and *ab*: *a*∗*b* = *0* **shows** *b* = *0*
proof −
  { assume *bz*: *b* $\neq$ *0*

```
    from no-zero-divisors [OF az bz] ab have False by blast }
  thus b = 0 by blast
qed

interpretation class-ringb: ringb
  [op + op ∗ op ˆ 0::'a::{idom,recpower,number-ring} 1 op − uminus]
proof(unfold-locales, simp add: ring-simps power-Suc, auto)
  fix w x y z ::'a::{idom,recpower,number-ring}
  assume p: w ∗ y + x ∗ z = w ∗ z + x ∗ y and ynz: y ≠ z
  hence ynz': y − z ≠ 0 by simp
  from p have w ∗ y + x∗ z − w∗z − x∗y = 0 by simp
  hence w∗ (y − z) − x ∗ (y − z) = 0 by (simp add: ring-simps)
  hence (y − z) ∗ (w − x) = 0 by (simp add: ring-simps)
  with no-zero-divirors-neq0 [OF ynz']
  have w − x = 0 by blast
  thus w = x by simp
qed

declaration ⟪ normalizer-funs @{thm class-ringb.axioms} ⟫

interpretation natgb: semiringb
  [op + op ∗ op ˆ 0::nat 1]
proof (unfold-locales, simp add: ring-simps power-Suc)
  fix w x y z ::nat
  { assume p: w ∗ y + x ∗ z = w ∗ z + x ∗ y and ynz: y ≠ z
    hence y < z ∨ y > z by arith
    moreover {
       assume lt:y <z hence ∃ k. z = y + k ∧ k > 0 by (rule-tac x=z − y in
exI, auto)
      then obtain k where kp: k>0 and yz:z = y + k by blast
      from p have (w ∗ y + x ∗y) + x∗k = (w ∗ y + x∗y) + w∗k by (simp add:
yz ring-simps)
      hence x∗k = w∗k by simp
      hence w = x using kp by (simp add: mult-cancel2) }
    moreover {
     assume lt: y >z hence ∃ k. y = z + k ∧ k>0 by (rule-tac x=y − z in exI,
auto)
      then obtain k where kp: k>0 and yz:y = z + k by blast
      from p have (w ∗ z + x ∗z) + w∗k = (w ∗ z + x∗z) + x∗k by (simp add:
yz ring-simps)
      hence w∗k = x∗k by simp
      hence w = x using kp by (simp add: mult-cancel2)}
    ultimately have w=x by blast }
  thus (w ∗ y + x ∗ z = w ∗ z + x ∗ y) = (w = x ∨ y = z) by auto
qed

declaration ⟪ normalizer-funs @{thm natgb.axioms} ⟫

locale fieldgb = ringb + gb-field
```

begin

declare *axioms* [*normalizer del*]

lemma *axioms* [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*
idom rules: *noteq-reduce add-scale-eq-noteq*
ideal rules: *subr0-iff add-r0-iff*]:
fieldgb add mul pwr r0 r1 sub neg divide inverse **by** *unfold-locales*
end

lemmas *bool-simps* = *simp-thms*(*1–34*)
lemma *dnf*:
$(P \ \& \ (Q \mid R)) = ((P\&Q) \mid (P\&R))$ $((Q \mid R) \ \& \ P) = ((Q\&P) \mid (R\&P))$
$(P \wedge Q) = (Q \wedge P)$ $(P \vee Q) = (Q \vee P)$
by *blast+*

lemmas *weak-dnf-simps* = *dnf bool-simps*

lemma *nnf-simps*:
$(\neg(P \wedge Q)) = (\neg P \vee \neg Q)$ $(\neg(P \vee Q)) = (\neg P \wedge \neg Q)$ $(P \longrightarrow Q) = (\neg P \vee Q)$
$(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg Q))$ $(\neg \neg(P)) = P$
by *blast+*

lemma *PFalse*:
$P \equiv False \Longrightarrow \neg P$
$\neg P \Longrightarrow (P \equiv False)$
by *auto*

use *Tools/Groebner-Basis/groebner.ML*

method-setup *algebra* =

```
⟪
let
 fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ()
 val addN = add
 val delN = del
 val any-keyword = keyword addN || keyword delN
 val thms = Scan.repeat (Scan.unless any-keyword Attrib.multi-thm) >> flat;
in
fn src => Method.syntax
    ((Scan.optional (keyword addN |-- thms) []) --
    (Scan.optional (keyword delN |-- thms) [])) src
 #> (fn ((add-ths, del-ths), ctxt) =>
      Method.SIMPLE-METHOD' (Groebner.algebra-tac add-ths del-ths ctxt))
```

end
⟫ *solve polynomial equations over (semi)rings and ideal membership problems using Groebner bases*

end

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

theory *Dense-Linear-Order*
imports *Finite-Set*
uses
Tools/Qelim/qelim.ML
Tools/Qelim/langford-data.ML
Tools/Qelim/ferrante-rackoff-data.ML
(*Tools/Qelim/langford.ML*)
(*Tools/Qelim/ferrante-rackoff.ML*)
begin

setup *Langford-Data.setup*
setup *Ferrante-Rackoff-Data.setup*

context *linorder*
begin

lemma *less-not-permute*: $\neg\ (x < y \wedge y < x)$ **by** (*simp add*: *not-less linear*)

lemma *gather-simps*:
shows
$(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge x < u \wedge P\ x) \longleftrightarrow (\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in (insert\ u\ U).\ x < y) \wedge P\ x)$
and $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge l < x \wedge P\ x) \longleftrightarrow (\exists x.\ (\forall y \in (insert\ l\ L).\ y < x) \wedge (\forall y \in U.\ x < y) \wedge P\ x)$
$(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge x < u) \longleftrightarrow (\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in (insert\ u\ U).\ x < y))$
and $(\exists x.\ (\forall y \in L.\ y < x) \wedge (\forall y \in U.\ x < y) \wedge l < x) \longleftrightarrow (\exists x.\ (\forall y \in (insert\ l\ L).\ y < x) \wedge (\forall y \in U.\ x < y))$ **by** *auto*

lemma
gather-start: $(\exists x.\ P\ x) \equiv (\exists x.\ (\forall y \in \{\}.\ y < x) \wedge (\forall y \in \{\}.\ x < y) \wedge P\ x)$
by *simp*

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

lemma *minf-lt*: $\exists z\ .\ \forall x.\ x < z \longrightarrow (x < t \longleftrightarrow True)$ **by** *auto*
lemma *minf-gt*: $\exists z\ .\ \forall x.\ x < z \longrightarrow (t < x \longleftrightarrow False)$
by (*simp add*: *not-less*) (*rule exI*[**where** $x=t$], *auto simp add*: *less-le*)

lemma *minf-le*: $\exists z. \forall x. x < z \longrightarrow (x \leq t \longleftrightarrow True)$ **by** (*auto simp add*: *less-le*)
lemma *minf-ge*: $\exists z. \forall x. x < z \longrightarrow (t \leq x \longleftrightarrow False)$
by (*auto simp add*: *less-le not-less not-le*)
lemma *minf-eq*: $\exists z. \forall x. x < z \longrightarrow (x = t \longleftrightarrow False)$ **by** *auto*
lemma *minf-neq*: $\exists z. \forall x. x < z \longrightarrow (x \neq t \longleftrightarrow True)$ **by** *auto*
lemma *minf-P*: $\exists z. \forall x. x < z \longrightarrow (P \longleftrightarrow P)$ **by** *blast*

Theorems for $\exists z. \forall x. x < z \longrightarrow (P\ x \longleftrightarrow P_{+\infty})$

lemma *pinf-gt*: $\exists z . \forall x. z < x \longrightarrow (t < x \longleftrightarrow True)$ **by** *auto*
lemma *pinf-lt*: $\exists z . \forall x. z < x \longrightarrow (x < t \longleftrightarrow False)$
by (*simp add*: *not-less*) (*rule exI*[**where** *x*=*t*], *auto simp add*: *less-le*)

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

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

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

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

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

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

lemma *lin-dense-ge*: $t \in U \Longrightarrow \forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t< u \longrightarrow t \notin U) \wedge l< x \wedge x < u \wedge t \leq x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow t \leq y)$
proof(*clarsimp*)

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

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

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

lemma *finite-set-intervals*:
assumes px: $P\ x$ **and** lx: $l \leq x$ **and** xu: $x \leq u$ **and** $linS$: $l\in S$
and $uinS$: $u \in S$ **and** fS:*finite* S **and** lS: $\forall\ x\in\ S.\ l \leq x$ **and** Su: $\forall\ x\in\ S.\ x \leq u$
shows $\exists\ a \in S.\ \exists\ b \in S.\ (\forall\ y.\ a < y \wedge y < b \longrightarrow y \notin S) \wedge a \leq x \wedge x \leq b \wedge P\ x$
proof−
let $?Mx = \{y.\ y\in\ S \wedge y \leq x\}$
let $?xM = \{y.\ y\in\ S \wedge x \leq y\}$

let *?a* = *Max ?Mx*
let *?b* = *Min ?xM*
have *MxS*: *?Mx* ⊆ *S* **by** *blast*
hence *fMx*: *finite ?Mx* **using** *fS finite-subset* **by** *auto*
from *lx linS* **have** *linMx*: *l* ∈ *?Mx* **by** *blast*
hence *Mxne*: *?Mx* ≠ {} **by** *blast*
have *xMS*: *?xM* ⊆ *S* **by** *blast*
hence *fxM*: *finite ?xM* **using** *fS finite-subset* **by** *auto*
from *xu uinS* **have** *linxM*: *u* ∈ *?xM* **by** *blast*
hence *xMne*: *?xM* ≠ {} **by** *blast*
have *ax*:*?a* ≤ *x* **using** *Mxne fMx* **by** *auto*
have *xb*:*x* ≤ *?b* **using** *xMne fxM* **by** *auto*
have *?a* ∈ *?Mx* **using** *Max-in*[*OF fMx Mxne*] **by** *simp* **hence** *ainS*: *?a* ∈ *S* **using** *MxS* **by** *blast*
have *?b* ∈ *?xM* **using** *Min-in*[*OF fxM xMne*] **by** *simp* **hence** *binS*: *?b* ∈ *S* **using** *xMS* **by** *blast*
have *noy*:∀ *y*. *?a* < *y* ∧ *y* < *?b* ⟶ *y* ∉ *S*
proof(*clarsimp*)
fix *y* **assume** *ay*: *?a* < *y* **and** *yb*: *y* < *?b* **and** *yS*: *y* ∈ *S*
from *yS* **have** *y*∈ *?Mx* ∨ *y*∈ *?xM* **by** (*auto simp add*: *linear*)
moreover {**assume** *y* ∈ *?Mx* **hence** *y* ≤ *?a* **using** *Mxne fMx* **by** *auto* **with** *ay* **have** *False* **by** (*simp add*: *not-le*[*symmetric*])}
moreover {**assume** *y* ∈ *?xM* **hence** *?b* ≤ *y* **using** *xMne fxM* **by** *auto* **with** *yb* **have** *False* **by** (*simp add*: *not-le*[*symmetric*])}
ultimately show *False* **by** *blast*
qed
from *ainS binS noy ax xb px* **show** *?thesis* **by** *blast*
qed

lemma *finite-set-intervals2*:
assumes *px*: *P x* **and** *lx*: *l* ≤ *x* **and** *xu*: *x* ≤ *u* **and** *linS*: *l*∈ *S*
and *uinS*: *u* ∈ *S* **and** *fS*:*finite S* **and** *lS*: ∀ *x*∈ *S*. *l* ≤ *x* **and** *Su*: ∀ *x*∈ *S*. *x* ≤ *u*
shows (∃ *s*∈ *S*. *P s*) ∨ (∃ *a* ∈ *S*. ∃ *b* ∈ *S*. (∀ *y*. *a* < *y* ∧ *y* < *b* ⟶ *y* ∉ *S*) ∧ *a* < *x* ∧ *x* < *b* ∧ *P x*)
proof−
from *finite-set-intervals*[**where** *P*=*P*, *OF px lx xu linS uinS fS lS Su*]
obtain *a* **and** *b* **where**
as: *a*∈ *S* **and** *bs*: *b*∈ *S* **and** *noS*:∀ *y*. *a* < *y* ∧ *y* < *b* ⟶ *y* ∉ *S*
and *axb*: *a* ≤ *x* ∧ *x* ≤ *b* ∧ *P x* **by** *auto*
from *axb* **have** *x*= *a* ∨ *x*= *b* ∨ (*a* < *x* ∧ *x* < *b*) **by** (*auto simp add*: *le-less*)
thus *?thesis* **using** *px as bs noS* **by** *blast*
qed

end

36 The classical QE after Langford for dense linear orders

context *dense-linear-order*
begin

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

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

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

lemma *exists-neq*: $\exists (x::'a).\ x \neq t$ $\exists (x::'a).\ t \neq x$
  using *gt-ex*[*of t*] **by** *auto*

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

lemma *axiom*: *dense-linear-order* $(op \leq)$ $(op <)$ **by** *fact*
lemma *atoms*:
  includes *meta-term-syntax*
  shows *TERM* $(less :: 'a \Rightarrow \text{-})$
    and *TERM* $(less\text{-}eq :: 'a \Rightarrow \text{-})$
    and *TERM* $(op = :: 'a \Rightarrow \text{-})$ **.**

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

end

lemma *dnf*:
  $(P\ \&\ (Q \mid R)) = ((P\&Q) \mid (P\&R))$
  $((Q \mid R)\ \&\ P) = ((Q\&P) \mid (R\&P))$
  by *blast+*

lemmas *weak-dnf-simps* = *simp-thms dnf*

lemma *nnf-simps*:
  $(\neg(P \wedge Q)) = (\neg P \vee \neg Q)$ $(\neg(P \vee Q)) = (\neg P \wedge \neg Q)$ $(P \longrightarrow Q) = (\neg P \vee Q)$
  $(P = Q) = ((P \wedge Q) \vee (\neg P \wedge \neg\ Q))$ $(\neg\ \neg(P)) = P$
  by *blast+*

lemma *ex-distrib*: $(\exists x.\ P\ x \vee Q\ x) \longleftrightarrow ((\exists x.\ P\ x) \vee (\exists x.\ Q\ x))$ **by** *blast*

lemmas *dnf-simps* = *weak-dnf-simps nnf-simps ex-distrib*

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

37 Contructive dense linear orders yield QE for linear arithmetic over ordered Fields – see *Arith-Tools.thy*

Linear order without upper bounds

class *linorder-no-ub* = *linorder* +
  assumes *gt-ex*: $\exists y.\ x < y$

begin

lemma *ge-ex*: $\exists y.\ x \leq y$ **using** *gt-ex* **by** *auto*

Theorems for $\exists z.\ \forall x.\ z < x \longrightarrow (P\ x \longleftrightarrow P_{+\infty})$

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

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

lemma *pinf-ex*: **assumes** *ex*:$\exists z.\ \forall x.\ z < x \longrightarrow (P\ x \longleftrightarrow P1)$ **and** *p1*: *P1* **shows** $\exists\ x.\ P\ x$
proof−
  from *ex* **obtain** *z* **where** *z*: $\forall x.\ z < x \longrightarrow (P\ x \longleftrightarrow P1)$ **by** *blast*
  from *gt-ex* **obtain** *x* **where** *x*: $z < x$ **by** *blast*
  from *z x p1* **show** *?thesis* **by** *blast*
qed

end

Linear order without upper bounds

class *linorder-no-lb* = *linorder* +

assumes *lt-ex*: $\exists y.\ y < x$
begin

lemma *le-ex*: $\exists y.\ y \leq x$ **using** *lt-ex* **by** *auto*

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

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

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

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

end

class *constr-dense-linear-order* = *linorder-no-lb* + *linorder-no-ub* +
  fixes *between*
  assumes *between-less*: $x < y \Longrightarrow x < between\ x\ y \wedge between\ x\ y < y$
    and *between-same*: $between\ x\ x = x$
begin

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

lemma *rinf-U*:
  assumes *fU*: *finite U*
  and *lin-dense*: $\forall x\ l\ u.\ (\forall\ t.\ l < t \wedge t< u \longrightarrow t \notin U) \wedge l< x \wedge x < u \wedge P\ x \longrightarrow (\forall\ y.\ l < y \wedge y < u \longrightarrow P\ y\ )$
  and *nmpiU*: $\forall x.\ \neg\ MP \wedge \neg PP \wedge P\ x \longrightarrow (\exists\ u\in\ U.\ \exists\ u' \in U.\ u \leq x \wedge x \leq u')$
  and *nmi*: $\neg\ MP$ **and** *npi*: $\neg\ PP$ **and** *ex*: $\exists\ x.\ \ P\ x$
  shows $\exists\ u\in\ U.\ \exists\ u' \in\ U.\ P\ (between\ u\ u')$
proof−
  from *ex* **obtain** *x* **where** *px*: *P x* **by** *blast*
  from *px nmi npi nmpiU* **have** $\exists\ u\in\ U.\ \exists\ u' \in\ U.\ u \leq x \wedge x \leq u'$ **by** *auto*
  then obtain *u* **and** *u'* **where** *uU*:$u\in\ U$ **and** *uU'*: $u' \in U$ **and** *ux*:$u \leq x$ **and** *xu'*:$x \leq u'$ **by** *auto*
  from *uU* **have** *Une*: $U \neq \{\}$ **by** *auto*
  let *?l* = *Min U*
  let *?u* = *Max U*
  have *linM*: $?l \in U$ **using** *fU Une* **by** *simp*
  have *uinM*: $?u \in U$ **using** *fU Une* **by** *simp*
  have *lM*: $\forall\ t\in\ U.\ ?l \leq t$ **using** *Une fU* **by** *auto*
  have *Mu*: $\forall\ t\in\ U.\ t \leq\ ?u$ **using** *Une fU* **by** *auto*
  have *th*:$?l \leq u$ **using** *uU Une lM* **by** *auto*
  from *order-trans*[*OF th ux*] **have** *lx*: $?l \leq x$ **.**
  have *th*: $u' \leq\ ?u$ **using** *uU' Une Mu* **by** *simp*
  from *order-trans*[*OF xu' th*] **have** *xu*: $x \leq\ ?u$ **.**
  from *finite-set-intervals2*[**where** *P=P*,*OF px lx xu linM uinM fU lM Mu*]
  have $(\exists\ s\in\ U.\ P\ s) \vee$
     $(\exists\ t1\in\ U.\ \exists\ t2 \in\ U.\ (\forall\ y.\ t1 < y \wedge y < t2 \longrightarrow y \notin U) \wedge t1 < x \wedge x < t2 \wedge P\ x)$ **.**
  moreover **{ fix** *u* **assume** *um*: $u\in U$ **and** *pu*: *P u*
    have $between\ u\ u = u$ **by** (*simp add*: *between-same*)
    with *um pu* **have** *P* (*between u u*) **by** *simp*
    with *um* **have** *?thesis* **by** *blast***}**
  moreover{
    assume $\exists\ t1\in\ U.\ \exists\ t2 \in\ U.\ (\forall\ y.\ t1 < y \wedge y < t2 \longrightarrow y \notin U) \wedge t1 < x \wedge x < t2 \wedge P\ x$
      then obtain *t1* **and** *t2* **where** *t1M*: $t1 \in U$ **and** *t2M*: $t2\in\ U$
        and *noM*: $\forall\ y.\ t1 < y \wedge y < t2 \longrightarrow y \notin U$ **and** *t1x*: $t1 < x$ **and** *xt2*: $x < t2$ **and** *px*: *P x*

by *blast*
from *less-trans*[*OF t1x xt2*] **have** *t1t2*: $t1 < t2$ **.**
let $?u = between\ t1\ t2$
from *between-less t1t2* **have** *t1lu*: $t1 < ?u$ **and** *ut2*: $?u < t2$ **by** *auto*
from *lin-dense noM t1x xt2 px t1lu ut2* **have** $P\ ?u$ **by** *blast*
with *t1M t2M* **have** *?thesis* **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

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

lemmas *minf-thms* = *minf-conj minf-disj minf-eq minf-neq minf-lt minf-le minf-gt minf-ge minf-P*
lemmas *pinf-thms* = *pinf-conj pinf-disj pinf-eq pinf-neq pinf-lt pinf-le pinf-gt pinf-ge pinf-P*

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

lin-dense-lt lin-dense-le lin-dense-gt lin-dense-ge lin-dense-P

lemma *ferrack-axiom*: *constr-dense-linear-order less-eq less between* **by** *fact*
lemma *atoms*:
  includes *meta-term-syntax*
  shows *TERM* (*less* :: 'a ⇒ -)
    and *TERM* (*less-eq* :: 'a ⇒ -)
    and *TERM* (*op* = :: 'a ⇒ -) **.**

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

declaration ⟪

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

⟫

end

use *Tools/Qelim/ferrante-rackoff.ML*

method-setup *ferrack* = ⟪

Method.ctxt-args (*Method.SIMPLE-METHOD′ o FerranteRackoff.dlo-tac*)
⟫ *Ferrante and Rackoff's algorithm for quantifier elimination in dense linear orders*

end

38 Arith-Tools: Setup of arithmetic tools

theory *Arith-Tools*
imports *Groebner-Basis Dense-Linear-Order*
uses
~~/src/Provers/Arith/cancel-numeral-factor.ML
~~/src/Provers/Arith/extract-common-term.ML
int-factor-simprocs.ML
nat-simprocs.ML
begin

38.1 Simprocs for the Naturals

declaration ⟪ *K nat-simprocs-setup* ⟫

38.1.1 For simplifying $Suc\ m - K$ and $K - Suc\ m$

Where K above is a literal

lemma *Suc-diff-eq-diff-pred*: *Numeral0* < *n* ==> *Suc m* − *n* = *m* − (*n* − *Numeral1*)
by (*simp add*: *numeral-0-eq-0 numeral-1-eq-1 split add*: *nat-diff-split*)

Now just instantiating n to *number-of* v does the right simplification, but with some redundant inequality tests.

lemma *neg-number-of-pred-iff-0*:
neg (*number-of* (*Numeral.pred v*)::*int*) = (*number-of v* = (*0*::*nat*))
apply (*subgoal-tac neg* (*number-of* (*Numeral.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* (*uminus v*)::*int*) ==>
Suc m − (*number-of v*) = *m* − (*number-of* (*Numeral.pred v*))
apply (*subst Suc-diff-eq-diff-pred*)
apply *simp*
apply (*simp del*: *nat-numeral-1-eq-1*)
apply (*auto simp only*: *diff-nat-number-of less-0-number-of* [*symmetric*]
neg-number-of-pred-iff-0)
done

lemma *diff-Suc-eq-diff-pred*: $m - Suc\ n = (m - 1) - n$
by (*simp add*: *numerals split add*: *nat-diff-split*)

38.1.2 For *nat-case* and *nat-rec*

lemma *nat-case-number-of* [*simp*]:
nat-case a f (*number-of v*) =
(*let pv* = *number-of* (*Numeral.pred v*) *in*
if neg pv then a else f (*nat pv*))
by (*simp split add*: *nat.split add*: *Let-def neg-number-of-pred-iff-0*)

lemma *nat-case-add-eq-if* [*simp*]:
nat-case a f ((*number-of v*) + *n*) =
(*let pv* = *number-of* (*Numeral.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-pred-iff-0)
done

lemma *nat-rec-number-of* [*simp*]:
nat-rec a f (*number-of v*) =
(*let pv* = *number-of* (*Numeral.pred v*) *in*
if neg pv then a else f (*nat pv*) (*nat-rec a f* (*nat pv*)))
apply (*case-tac* (*number-of v*) ::*nat*)
apply (*simp-all* (*no-asm-simp*) *add*: *Let-def neg-number-of-pred-iff-0*)
apply (*simp split add*: *split-if-asm*)
done

lemma *nat-rec-add-eq-if* [*simp*]:
nat-rec a f (*number-of v* + *n*) =
(*let pv* = *number-of* (*Numeral.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-pred-iff-0)
done

38.1.3 Various Other Lemmas

Evens and Odds, for Mutilated Chess Board

Lemmas for specialist use, NOT as default simprules

lemma *nat-mult-2*: $2 * z = (z+z::nat)$

proof –
  have $2{*}z = (1 + 1){*}z$ **by** *simp*
  also have ... $= z{+}z$ **by** (*simp add*: *left-distrib*)
  finally show *?thesis* **.**
qed

lemma *nat-mult-2-right*: $z * 2 = (z{+}z{::}nat)$
by (*subst mult-commute*, *rule nat-mult-2*)

Case analysis on $n < (2{::}'a)$

lemma *less-2-cases*: $(n{::}nat) < 2 \implies n = 0 \mid n = Suc\ 0$
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

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*]

38.1.4 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*,*noatp*]

lemmas *divide-less-0-iff-number-of* =
divide-less-0-iff [*of number-of w*, *standard*]
declare *divide-less-0-iff-number-of* [*simp*,*noatp*]

lemmas *zero-le-divide-iff-number-of* =
zero-le-divide-iff [*of number-of w*, *standard*]
declare *zero-le-divide-iff-number-of* [*simp*,*noatp*]

lemmas *divide-le-0-iff-number-of* =
  divide-le-0-iff [*of number-of w*, *standard*]
declare *divide-le-0-iff-number-of* [*simp*,*noatp*]

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*,*noatp*]

lemmas *le-minus-iff-number-of* =
  le-minus-iff [*of number-of v*, *standard*]
declare *le-minus-iff-number-of* [*simp*,*noatp*]

lemmas *equation-minus-iff-number-of* =
  equation-minus-iff [*of number-of v*, *standard*]
declare *equation-minus-iff-number-of* [*simp*,*noatp*]

lemmas *minus-less-iff-number-of* =
  minus-less-iff [*of - number-of v*, *standard*]
declare *minus-less-iff-number-of* [*simp*,*noatp*]

lemmas *minus-le-iff-number-of* =
  minus-le-iff [*of - number-of v*, *standard*]
declare *minus-le-iff-number-of* [*simp*,*noatp*]

lemmas *minus-equation-iff-number-of* =
  minus-equation-iff [*of - number-of v*, *standard*]
declare *minus-equation-iff-number-of* [*simp*,*noatp*]

To Simplify Inequalities Where One Side is the Constant 1

lemma *less-minus-iff-1* [*simp*,*noatp*]:
 fixes $b::'b::\{ordered\text{-}idom,number\text{-}ring\}$
 shows $(1 < - b) = (b < -1)$
by *auto*

lemma *le-minus-iff-1* [*simp*,*noatp*]:
 fixes $b::'b::\{ordered\text{-}idom,number\text{-}ring\}$
 shows $(1 \leq - b) = (b \leq -1)$
by *auto*

lemma *equation-minus-iff-1* [*simp*,*noatp*]:
  fixes b::$'b$::*number-ring*
  shows $(1 = - b) = (b = -1)$
by (*subst equation-minus-iff*, *auto*)

lemma *minus-less-iff-1* [*simp*,*noatp*]:
  fixes a::$'b$::{*ordered-idom*,*number-ring*}
  shows $(- a < 1) = (-1 < a)$
by *auto*

lemma *minus-le-iff-1* [*simp*,*noatp*]:
  fixes a::$'b$::{*ordered-idom*,*number-ring*}
  shows $(- a \leq 1) = (-1 \leq a)$
by *auto*

lemma *minus-equation-iff-1* [*simp*,*noatp*]:
  fixes a::$'b$::*number-ring*
  shows $(- a = 1) = (a = -1)$
by (*subst minus-equation-iff*, *auto*)

Cancellation of constant factors in comparisons ($<$ and $\leq$)

lemmas *mult-less-cancel-left-number-of* =
    mult-less-cancel-left [*of number-of v*, *standard*]
declare *mult-less-cancel-left-number-of* [*simp*,*noatp*]

lemmas *mult-less-cancel-right-number-of* =
    mult-less-cancel-right [*of - number-of v*, *standard*]
declare *mult-less-cancel-right-number-of* [*simp*,*noatp*]

lemmas *mult-le-cancel-left-number-of* =
    mult-le-cancel-left [*of number-of v*, *standard*]
declare *mult-le-cancel-left-number-of* [*simp*,*noatp*]

lemmas *mult-le-cancel-right-number-of* =
    mult-le-cancel-right [*of - number-of v*, *standard*]
declare *mult-le-cancel-right-number-of* [*simp*,*noatp*]

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*]

38.1.5 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

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*, *standard*]
declare *half-gt-zero* [*simp*]

lemma *nat-dvd-not-less*:
[| $0 < m$; $m < n$ |] ==> ¬ *n dvd* (*m*::*nat*)
by (*unfold dvd-def*) *auto*

ML ⟪
val divide-minus1 = @{*thm divide-minus1*};

val minus1-divide = @{*thm minus1-divide*};
⟫

38.2 Groebner Bases for fields

interpretation *class-fieldgb*:
fieldgb[*op* + *op* ∗ *op* ˆ *0*::′*a*::{*field*,*recpower*,*number-ring*} *1* *op* − *uminus* *op* / *inverse*] **apply** (*unfold-locales*) **by** (*simp-all add*: *divide-inverse*)

lemma *divide-Numeral1*: (*x*::′*a*::{*field*,*number-ring*}) / *Numeral1* = *x* **by** *simp*
lemma *divide-Numeral0*: (*x*::′*a*::{*field*,*number-ring*, *division-by-zero*}) / *Numeral0* = *0*
by *simp*
lemma *mult-frac-frac*: ((*x*::′*a*::{*field*,*division-by-zero*}) / *y*) ∗ (*z* / *w*) = (*x*∗*z*) / (*y*∗*w*)
by *simp*
lemma *mult-frac-num*: ((*x*::′*a*::{*field*, *division-by-zero*}) / *y*) ∗ *z* = (*x*∗*z*) / *y*
by *simp*
lemma *mult-num-frac*: ((*x*::′*a*::{*field*, *division-by-zero*}) / *y*) ∗ *z* = (*x*∗*z*) / *y*
by *simp*

lemma *Numeral1-eq1-nat*: (*1*::*nat*) = *Numeral1* **by** *simp*

lemma *add-frac-num*: *y*≠ *0* ⟹ (*x*::′*a*::{*field*, *division-by-zero*}) / *y* + *z* = (*x* + *z*∗*y*) / *y*
by (*simp add*: *add-divide-distrib*)
lemma *add-num-frac*: *y*≠ *0* ⟹ *z* + (*x*::′*a*::{*field*, *division-by-zero*}) / *y* = (*x* + *z*∗*y*) / *y*
by (*simp add*: *add-divide-distrib*)

ML⟪
```
local
 val zr = @{cpat 0}
 val zT = ctyp-of-term zr
 val geq = @{cpat op =}
 val eqT = Thm.dest-ctyp (ctyp-of-term geq) |> hd
 val add-frac-eq = mk-meta-eq @{thm add-frac-eq}
 val add-frac-num = mk-meta-eq @{thm add-frac-num}
 val add-num-frac = mk-meta-eq @{thm add-num-frac}

 fun prove-nz ss T t =
    let
      val z = instantiate-cterm ([(zT,T)],[]) zr
      val eq = instantiate-cterm ([(eqT,T)],[]) geq
      val th = Simplifier.rewrite (ss addsimps simp-thms)
           (Thm.capply @{cterm Trueprop} (Thm.capply @{cterm Not}
                  (Thm.capply (Thm.capply eq t) z)))
    in equal-elim (symmetric th) TrueI
```

```
   end

 fun proc phi ss ct =
  let
   val ((x,y),(w,z)) =
       (Thm.dest-binop #> (fn (a,b) => (Thm.dest-binop a, Thm.dest-binop b)))
ct
   val - = map (HOLogic.dest-number o term-of) [x,y,z,w]
   val T = ctyp-of-term x
   val [y-nz, z-nz] = map (prove-nz ss T) [y, z]
   val th = instantiate' [SOME T] (map SOME [y,z,x,w]) add-frac-eq
  in SOME (implies-elim (implies-elim th y-nz) z-nz)
  end
  handle CTERM - => NONE | TERM - => NONE | THM - => NONE

 fun proc2 phi ss ct =
  let
   val (l,r) = Thm.dest-binop ct
   val T = ctyp-of-term l
  in (case (term-of l, term-of r) of
     (Const(@{const-name HOL.divide},-)$-$-, -) =>
       let val (x,y) = Thm.dest-binop l val z = r
           val - = map (HOLogic.dest-number o term-of) [x,y,z]
           val ynz = prove-nz ss T y
          in SOME (implies-elim (instantiate' [SOME T] (map SOME [y,x,z])
add-frac-num) ynz)
       end
    | (-, Const (@{const-name HOL.divide},-)$-$-) =>
       let val (x,y) = Thm.dest-binop r val z = l
           val - = map (HOLogic.dest-number o term-of) [x,y,z]
           val ynz = prove-nz ss T y
          in SOME (implies-elim (instantiate' [SOME T] (map SOME [y,z,x])
add-num-frac) ynz)
       end
    | - => NONE)
  end
  handle CTERM - => NONE | TERM - => NONE | THM - => NONE

 fun is-number (Const(@{const-name HOL.divide},-)$a$b) = is-number a andalso
is-number b
   | is-number t = can HOLogic.dest-number t

 val is-number = is-number o term-of

 fun proc3 phi ss ct =
  (case term-of ct of
   Const(@{const-name HOL.less},-)$(Const(@{const-name HOL.divide},-)$-$-)$-
=>
      let
```

```
      val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-less-eq}
    in SOME (mk-meta-eq th) end
 | Const(@{const-name HOL.less-eq},-)$(Const(@{const-name HOL.divide},-)$-$-)$-
=>
    let
      val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-le-eq}
    in SOME (mk-meta-eq th) end
 | Const(op =,-)$(Const(@{const-name HOL.divide},-)$-$-)$- =>
    let
      val ((a,b),c) = Thm.dest-binop ct |>> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm divide-eq-eq}
    in SOME (mk-meta-eq th) end
 | Const(@{const-name HOL.less},-)$-$(Const(@{const-name HOL.divide},-)$-$-)
=>
   let
    val (a,(b,c)) = Thm.dest-binop ct ||> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm less-divide-eq}
    in SOME (mk-meta-eq th) end
 | Const(@{const-name HOL.less-eq},-)$-$(Const(@{const-name HOL.divide},-)$-$-)
=>
   let
    val (a,(b,c)) = Thm.dest-binop ct ||> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm le-divide-eq}
    in SOME (mk-meta-eq th) end
 | Const(op =,-)$-$(Const(@{const-name HOL.divide},-)$-$-) =>
   let
    val (a,(b,c)) = Thm.dest-binop ct ||> Thm.dest-binop
      val - = map is-number [a,b,c]
      val T = ctyp-of-term c
      val th = instantiate' [SOME T] (map SOME [a,b,c]) @{thm eq-divide-eq}
    in SOME (mk-meta-eq th) end
 | - => NONE)
 handle TERM - => NONE | CTERM - => NONE | THM - => NONE

val add-frac-frac-simproc =
      make-simproc {lhss = [@{cpat (?x::?'a::field)/?y + (?w::?'a::field)/?z}],
                    name = add-frac-frac-simproc,
```

```
                  proc = proc, identifier = []}

val add-frac-num-simproc =
        make-simproc {lhss = [@{cpat (?x::?'a::field)/?y + ?z}, @{cpat ?z +
(?x::?'a::field)/?y}],
                  name = add-frac-num-simproc,
                  proc = proc2, identifier = []}

val ord-frac-simproc =
  make-simproc
   {lhss = [@{cpat (?a::(?'a::{field, ord}))/?b < ?c},
          @{cpat (?a::(?'a::{field, ord}))/?b ≤ ?c},
          @{cpat ?c < (?a::(?'a::{field, ord}))/?b},
          @{cpat ?c ≤ (?a::(?'a::{field, ord}))/?b},
          @{cpat ?c = ((?a::(?'a::{field, ord}))/?b)},
          @{cpat ((?a::(?'a::{field, ord}))/ ?b) = ?c}],
          name = ord-frac-simproc, proc = proc3, identifier = []}

val nat-arith = map thm [add-nat-number-of, diff-nat-number-of,
             mult-nat-number-of, eq-nat-number-of, less-nat-number-of]

val comp-arith = (map thm [Let-def, if-False, if-True, add-0,
               add-Suc, add-number-of-left, mult-number-of-left,
               Suc-eq-add-numeral-1])@
               (map (fn s => thm s RS sym) [numeral-1-eq-1, numeral-0-eq-0])
               @ arith-simps@ nat-arith @ rel-simps
val ths = [@{thm mult-numeral-1}, @{thm mult-numeral-1-right},
         @{thm divide-Numeral1},
         @{thm Ring-and-Field.divide-zero}, @{thm divide-Numeral0},
         @{thm divide-divide-eq-left}, @{thm mult-frac-frac},
         @{thm mult-num-frac}, @{thm mult-frac-num},
         @{thm mult-frac-frac}, @{thm times-divide-eq-right},
         @{thm times-divide-eq-left}, @{thm divide-divide-eq-right},
         @{thm diff-def}, @{thm minus-divide-left},
         @{thm Numeral1-eq1-nat}, @{thm add-divide-distrib} RS sym]

local
open Conv
in
val comp-conv = (Simplifier.rewrite
(HOL-basic-ss addsimps @{thms Groebner-Basis.comp-arith}
             addsimps ths addsimps comp-arith addsimps simp-thms
             addsimprocs field-cancel-numeral-factors
              addsimprocs [add-frac-frac-simproc, add-frac-num-simproc,
                          ord-frac-simproc]
               addcongs [@{thm if-weak-cong}]))
then-conv (Simplifier.rewrite (HOL-basic-ss addsimps
  [@{thm numeral-1-eq-1},@{thm numeral-0-eq-0}] @ @{thms numerals(1-2)}))
end
```

```
fun numeral-is-const ct =
  case term-of ct of
   Const (@{const-name HOL.divide},-) $ a $ b =>
     numeral-is-const (Thm.dest-arg1 ct) andalso numeral-is-const (Thm.dest-arg
ct)
 | Const (@{const-name HOL.uminus},-)$t => numeral-is-const (Thm.dest-arg
ct)
 | t => can HOLogic.dest-number t

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

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

in
 val field-comp-conv = comp-conv;
 val fieldgb-declaration =
  NormalizerData.funs @{thm class-fieldgb.axioms}
   {is-const = K numeral-is-const,
    dest-const = K dest-const,
    mk-const = mk-const,
    conv = K (K comp-conv)}
end;
»
```

declaration« *fieldgb-declaration* »

38.3 Ferrante and Rackoff algorithm over ordered fields

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

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

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

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

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

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

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

lemma *neg-prod-sum-le*: $(c::'a::ordered\text{-}field) < 0 \Longrightarrow ((c*x + t <= 0) == (x >= (- 1/c)*t))$

proof−
  assume H: $c < 0$
  have $c{*}x + t <= 0 = (c{*}x <= -t)$ **by** (*subst le-iff-diff-le-0* [*of* $c{*}x$ $-t$], *simp*)
  also have ... = $(-t/c <= x)$ **by** (*simp only*: *neg-divide-le-eq*[*OF H*] *ring-simps*)
  also have ... = $((-\ 1/c){*}t <= x)$ **by** *simp*
  finally show $(c{*}x + t <= 0) == (x >= (-\ 1/c){*}t)$ **by** *simp*
qed

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

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

lemma *nz-prod-eq*:$(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow ((c{*}x = 0) == (x = 0))$ **by** *simp*
lemma *nz-prod-sum-eq*: $(c::'a::ordered\text{-}field) \neq 0 \Longrightarrow ((c{*}x + t = 0) == (x = (-\ 1/c){*}t))$
proof−
  assume H: $c \neq 0$
  have $c{*}x + t = 0 = (c{*}x = -t)$ **by** (*subst eq-iff-diff-eq-0* [*of* $c{*}x$ $-t$], *simp*)
  also have ... = $(x = -t/c)$ **by** (*simp only*: *nonzero-eq-divide-eq*[*OF H*] *ring-simps*)
  finally show $(c{*}x + t = 0) == (x = (-\ 1/c){*}t)$ **by** *simp*
qed
lemma *sum-eq*:$((x::'a::pordered\text{-}ab\text{-}group\text{-}add) + t = 0) == (x = -\ t)$
  using *eq-diff-eq*[**where** $a= x$ **and** $b=t$ **and** $c=0$] **by** *simp*

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

declaration⟪
let
fun earlier [] *x y* = *false*
        | *earlier* (*h*::*t*) *x y* =

```
  if h aconvc y then false else if h aconvc x then true else earlier t x y;

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

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

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

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

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


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

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

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

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

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

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

```
    val rth = transitive nth (xnormalize-conv ctxt vs (Thm.rhs-of nth))
  in rth end
| @{term Not} $(Const(op =,-)$a$b) => Conv.arg-conv (field-isolate-conv phi ctxt
vs) ct
| - => reflexive ct
end;

fun classfield-whatis phi =
 let
  fun h x t =
   case term-of t of
     Const(op =, -)$y$z => if term-of x aconv y then Ferrante-Rackoff-Data.Eq
                          else Ferrante-Rackoff-Data.Nox
  | @{term Not}$(Const(op =, -)$y$z) => if term-of x aconv y then Ferrante-Rackoff-Data.NEq
                          else Ferrante-Rackoff-Data.Nox
   | Const(@{const-name HOL.less},-)$y$z =>
       if term-of x aconv y then Ferrante-Rackoff-Data.Lt
        else if term-of x aconv z then Ferrante-Rackoff-Data.Gt
        else Ferrante-Rackoff-Data.Nox
   | Const (@{const-name HOL.less-eq},-)$y$z =>
         if term-of x aconv y then Ferrante-Rackoff-Data.Le
         else if term-of x aconv z then Ferrante-Rackoff-Data.Ge
         else Ferrante-Rackoff-Data.Nox
   | - => Ferrante-Rackoff-Data.Nox
 in h end;
fun class-field-ss phi =
   HOL-basic-ss addsimps ([@{thm linorder-not-less}, @{thm linorder-not-le}])
   addsplits [@{thm abs-split},@{thm split-max}, @{thm split-min}]

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

end

39 SetInterval: Set intervals

theory *SetInterval*
imports *IntArith*
begin

context *ord*
begin
definition
  lessThan :: $'a => 'a$ *set* ((*1*{..<-})) **where**
  {..<u} == {x. $x < u$}

definition
atMost :: $'a$ => $'a$ *set* ((*1*{..-})) **where**
{..*u*} == {*x*. *x* ≤ *u*}

definition
greaterThan :: $'a$ => $'a$ *set* ((*1*{-<..})) **where**
{*l*<..} == {*x*. *l*<*x*}

definition
atLeast :: $'a$ => $'a$ *set* ((*1*{-..})) **where**
{*l*..} == {*x*. *l*≤*x*}

definition
greaterThanLessThan :: $'a$ => $'a$ => $'a$ *set* ((*1*{-<..<-})) **where**
{*l*<..<*u*} == {*l*<..} *Int* {..<*u*}

definition
atLeastLessThan :: $'a$ => $'a$ => $'a$ *set* ((*1*{-..<-})) **where**
{*l*..<*u*} == {*l*..} *Int* {..<*u*}

definition
greaterThanAtMost :: $'a$ => $'a$ => $'a$ *set* ((*1*{-<..-})) **where**
{*l*<..*u*} == {*l*<..} *Int* {..*u*}

definition
atLeastAtMost :: $'a$ => $'a$ => $'a$ *set* ((*1*{-..-})) **where**
{*l*..*u*} == {*l*..} *Int* {..*u*}

end

A note of warning when using {..<*n*} on type *nat*: it is equivalent to {*0*..<*n*} but some lemmas involving {*m*..<*n*} may not exist in {..<*n*}-form as well.

syntax
@*UNION-le* :: *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*$_{-} \le$ -)/ -) *10*)
@*UNION-less* :: *nat* ⇒ *nat* => $'b$ *set* => $'b$ *set* ((*3*⋃(*00*$_{-} <$ -)/ -) *10*)
@*INTER-le* :: *nat* ⇒ *nat* => $'b$ *set* => $'b$ *set* ((*3*⋂(*00*$_{-} \le$ -)/ -) *10*)

@INTER-less :: *nat* ⇒ *nat* => *'b set* => *'b set* $((3\bigcap(00\_ < \_)/\ \_)\ 10)$

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*

39.1 Various equivalences

lemma (**in** *ord*) *lessThan-iff* [*iff*]: (*i*: *lessThan k*) = (*i*<*k*)
by (*simp add*: *lessThan-def*)

lemma *Compl-lessThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*lessThan k* = *atLeast k*
apply (*auto simp add*: *lessThan-def atLeast-def*)
done

lemma *single-Diff-lessThan* [*simp*]: !!*k*:: *'a*::*order*. {*k*} − *lessThan k* = {*k*}
by *auto*

lemma (**in** *ord*) *greaterThan-iff* [*iff*]: (*i*: *greaterThan k*) = (*k*<*i*)
by (*simp add*: *greaterThan-def*)

lemma *Compl-greaterThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*greaterThan k* = *atMost k*
apply (*simp add*: *greaterThan-def atMost-def le-def*, *auto*)
done

lemma *Compl-atMost* [*simp*]: !!*k*:: *'a*::*linorder*. −*atMost k* = *greaterThan k*
apply (*subst Compl-greaterThan* [*symmetric*])
apply (*rule double-complement*)
done

lemma (**in** *ord*) *atLeast-iff* [*iff*]: (*i*: *atLeast k*) = (*k*<=*i*)
by (*simp add*: *atLeast-def*)

lemma *Compl-atLeast* [*simp*]:
!!*k*:: *'a*::*linorder*. −*atLeast k* = *lessThan k*
apply (*simp add*: *lessThan-def atLeast-def le-def*, *auto*)
done

lemma (**in** *ord*) *atMost-iff* [*iff*]: (*i*: *atMost k*) = (*i*<=*k*)
by (*simp add*: *atMost-def*)

lemma *atMost-Int-atLeast*: !!*n*:: *'a*::*order*. *atMost n Int atLeast n* = {*n*}
by (*blast intro*: *order-antisym*)

39.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-subset-iff* [*iff*]:
  $(atLeast\ x \subseteq atLeast\ y) = (y \le (x::'a::order))$
by (*blast intro*: *order-trans*)

lemma *atLeast-eq-iff* [*iff*]:
  $(atLeast\ x = atLeast\ y) = (x = (y::'a::linorder))$
by (*blast intro*: *order-antisym order-trans*)

lemma *greaterThan-subset-iff* [*iff*]:
  $(greaterThan\ x \subseteq greaterThan\ y) = (y \le (x::'a::linorder))$
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*]: $(atMost\ x \subseteq atMost\ y) = (x \le (y::'a::order))$
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*]:
  $(lessThan\ x \subseteq lessThan\ y) = (x \le (y::'a::linorder))$
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

39.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [*simp*,*noatp*]:
 $(i : \{l<..<u\}) = (l < i \ \&\ i < u)$
by (*simp add*: *greaterThanLessThan-def*)

lemma *atLeastLessThan-iff* [*simp*,*noatp*]:
(i : $\{l..<u\}$) = (l <= i & i < u)
by (*simp add*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-iff* [*simp*,*noatp*]:
(i : $\{l<..u\}$) = (l < i & i <= u)
by (*simp add*: *greaterThanAtMost-def*)

lemma *atLeastAtMost-iff* [*simp*,*noatp*]:
(i : $\{l..u\}$) = (l <= i & i <= u)
by (*simp add*: *atLeastAtMost-def*)

The above four lemmas could be declared as iffs. If we do so, a call to blast in Hyperreal/Star.ML, lemma *STAR-Int* seems to take forever (more than one hour).

end

39.3.1 Emptyness and singletons

context *order*
begin

lemma *atLeastAtMost-empty* [*simp*]: n < m ==> $\{m..n\}$ = {}
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

lemma *atLeastLessThan-empty*[*simp*]: $n \leq m$ ==> $\{m..<n\}$ = {}
by (*auto simp add*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-empty*[*simp*]:$l \leq k$ ==> $\{k<..l\}$ = {}
by(*auto simp*:*greaterThanAtMost-def greaterThan-def atMost-def*)

lemma *greaterThanLessThan-empty*[*simp*]:$l \leq k$ ==> $\{k<..l\}$ = {}
by(*auto simp*:*greaterThanLessThan-def greaterThan-def lessThan-def*)

lemma *atLeastAtMost-singleton* [*simp*]: $\{a..a\}$ = $\{a\}$
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

end

39.4 Intervals of natural numbers

39.4.1 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*

39.4.2 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*

39.4.3 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*

39.4.4 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*

39.4.5 The Constant *atLeastLessThan*

The orientation of the following rule is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

lemma *atLeast0LessThan*: {*0*::*nat*..<*n*} = {..<*n*}
by(*simp add*:*lessThan-def atLeastLessThan-def*)

declare *atLeast0LessThan*[*symmetric*, *code unfold*]

lemma *atLeastLessThan0*: {*m*..<*0*::*nat*} = {}
by (*simp add*: *atLeastLessThan-def*)

39.4.6 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* ≤ *Suc n* ⟹ {*m*..*Suc n*} = *insert* (*Suc n*) {*m*..*n*}
by (*auto simp add*: *atLeastAtMost-def*)

39.4.7 Image

lemma *image-add-atLeastAtMost*:
  (%*n*::*nat*. *n*+*k*) ' {*i*..*j*} = {*i*+*k*..*j*+*k*} (**is** *?A* = *?B*)
proof
  show *?A* ⊆ *?B* **by** *auto*
next
  show *?B* ⊆ *?A*

proof
fix n **assume** a: n : $?B$
hence $n - k : \{i..j\}$ **by** *auto*
moreover have $n = (n - k) + k$ **using** a **by** *auto*
ultimately show n : $?A$ **by** *blast*
qed
qed

lemma *image-add-atLeastLessThan*:
$(\%n::nat.\ n+k)$ ' $\{i..<j\} = \{i+k..<j+k\}$ (**is** $?A = ?B$)
proof
show $?A \subseteq ?B$ **by** *auto*
next
show $?B \subseteq ?A$
proof
fix n **assume** a: n : $?B$
hence $n - k : \{i..<j\}$ **by** *auto*
moreover have $n = (n - k) + k$ **using** a **by** *auto*
ultimately show n : $?A$ **by** *blast*
qed
qed

corollary *image-Suc-atLeastAtMost*[*simp*]:
Suc ' $\{i..j\}$ = $\{Suc\ i..Suc\ j\}$
using *image-add-atLeastAtMost*[**where** $k=1$] **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

39.4.8 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

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

lemma *subset-card-intvl-is-intvl*:
A <= {*k*..<*k*+*card A*} $\Longrightarrow$ *A* = {*k*..<*k*+*card A*} (**is** *PROP ?P*)
proof *cases*
assume *finite A*
thus *PROP ?P*
proof(*induct A rule*:*finite-linorder-induct*)
case *empty* **thus** *?case* **by** *auto*
next
case (*insert A b*)
moreover hence *b* ~: *A* **by** *auto*
moreover have *A* <= {*k*..<*k*+*card A*} **and** *b* = *k*+*card A*
using ‹*b* ~: *A*› *insert* **by** *fastsimp*+
ultimately show *?case* **by** *auto*
qed
next
assume ~*finite A* **thus** *PROP ?P* **by** *simp*
qed

39.4.9 Cardinality

lemma *card-lessThan* [*simp*]: *card* {..<*u*} = *u*
by (*induct u*, *simp-all add*: *lessThan-Suc*)

lemma *card-atMost* [*simp*]: *card* {..*u*} = *Suc u*
by (*simp add*: *lessThan-Suc-atMost* [*THEN sym*])

lemma *card-atLeastLessThan* [*simp*]: *card* {*l*..<*u*} = *u* − *l*
apply (*subgoal-tac card* {*l*..<*u*} = *card* {..<*u*−*l*})

apply (*erule ssubst*, *rule card-lessThan*)
apply (*subgoal-tac* (%*x*. *x* + *l*) ‘ {..<*u*−*l*} = {*l*..<*u*})
apply (*erule subst*)
apply (*rule card-image*)
apply (*simp add*: *inj-on-def*)
apply (*auto simp add*: *image-def atLeastLessThan-def lessThan-def*)
apply (*rule-tac x* = *x* − *l* **in** *exI*)
apply *arith*
done

lemma *card-atLeastAtMost* [*simp*]: *card* {*l*..*u*} = *Suc u* − *l*
by (*subst atLeastLessThanSuc-atLeastAtMost* [*THEN sym*], *simp*)

lemma *card-greaterThanAtMost* [*simp*]: *card* {*l*<..*u*} = *u* − *l*
by (*subst atLeastSucAtMost-greaterThanAtMost* [*THEN sym*], *simp*)

lemma *card-greaterThanLessThan* [*simp*]: *card* {*l*<..<*u*} = *u* − *Suc l*
by (*subst atLeastSucLessThan-greaterThanLessThan* [*THEN sym*], *simp*)

39.5 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*)

39.5.1 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*)

39.5.2 Cardinality

lemma *card-atLeastZeroLessThan-int*: *card* $\{(0::int)..<u\}$ = *nat u*
apply (*case-tac* $0 \leq u$)
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*)

39.6 Lemmas useful with the summation operator setsum

For examples, see Algebra/poly/UnivPoly2.thy

39.6.1 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*

39.6.2 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*

39.6.3 Some Differences

lemma *ivl-diff* [*simp*]:
$i \le n \Longrightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$
by(*auto*)

39.6.4 Some Subset Conditions

lemma *ivl-subset* [*simp*,*noatp*]:
$(\{i..<j\} \subseteq \{m..<n\}) = (j \le i \mid m \le i \,\&\, j \le (n::'a::linorder))$
apply(*auto simp*:*linorder-not-le*)
apply(*rule ccontr*)
apply(*insert linorder-le-less-linear*[*of i n*])
apply(*clarsimp simp*:*linorder-not-le*)
apply(*fastsimp*)
done

39.7 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*
(($3\sum_{-\,=\,-}^{-}$ -) [*0*,*0*,*0*,*10*] *10*)
-from-upto-setsum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(($3\sum_{-\,=\,-}^{<-}$ -) [*0*,*0*,*0*,*10*] *10*)
-upt-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(($3\sum_{-\,<\,-}$ -) [*0*,*0*,*10*] *10*)
-upto-setsum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(($3\sum_{-\,\leq\,-}$ -) [*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
$\sum x{\in}\{a..b\}.\ e$	$\sum x = a..b.\ e$	$\sum_{x\,=\,a}^{b} e$
$\sum x{\in}\{a..{<}b\}.\ e$	$\sum x = a..{<}b.\ e$	$\sum_{x\,=\,a}^{<b} e$
$\sum x{\in}\{..b\}.\ e$	$\sum x{\leq}b.\ e$	$\sum_{x\,\leq\,b} e$
$\sum x{\in}\{..{<}b\}.\ e$	$\sum x{<}b.\ e$	$\sum_{x\,<\,b} e$

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

39.8 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*])

lemma *setsum-head*:
  fixes *n* :: *nat*
  assumes *mn*: $m <= n$
  shows $(\sum x\in\{m..n\}.\ P\ x) = P\ m + (\sum x\in\{m<..n\}.\ P\ x)$ (**is** *?lhs* = *?rhs*)
proof −
  from *mn*
  have $\{m..n\} = \{m\} \cup \{m<..n\}$
    by (*auto intro*: *ivl-disj-un-singleton*)
  hence *?lhs* = $(\sum x\in\{m\} \cup \{m<..n\}.\ P\ x)$
    by (*simp add*: *atLeast0LessThan*)
  also have ... = *?rhs* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *setsum-head-upt*:
  fixes *m*::*nat*
  assumes *m*: $0 < m$
  shows $(\sum x<m.\ P\ x) = P\ 0 + (\sum x\in\{1..<m\}.\ P\ x)$
proof −
  have $(\sum x<m.\ P\ x) = (\sum x\in\{0..<m\}.\ P\ x)$
    by (*simp add*: *atLeast0LessThan*)
  also
  from *m*
  have ... = $(\sum x\in\{0..m - 1\}.\ P\ x)$
    by (*cases m*) (*auto simp add*: *atLeastLessThanSuc-atLeastAtMost*)
  also
  have ... = $P\ 0 + (\sum x\in\{0<..m - 1\}.\ P\ x)$
    by (*simp add*: *setsum-head*)
  also
  from *m*
  have $\{0<..m - 1\} = \{1..<m\}$
    by (*cases m*) (*auto simp add*: *atLeastLessThanSuc-atLeastAtMost*)
  finally show *?thesis* **.**
qed

39.9 The formula for geometric sums

lemma *geometric-sum*:
  $x \sim= 1 ==> (\sum i=0..<n.\ x\ \hat{}\ i) =$

$(x \,\hat{}\, n - 1) / (x - 1{::}'a{::}\{field, recpower\})$
by (*induct n*) (*simp-all add:field-simps power-Suc*)

39.10 The formula for arithmetic sums

lemma *gauss-sum*:
$((1{::}'a{::}comm\text{-}semiring\text{-}1) + 1)*(\sum i\in\{1..n\}.\ of\text{-}nat\ i) =$
$of\text{-}nat\ n*((of\text{-}nat\ n)+1)$
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case* **by** (*simp add*: *ring-simps*)
qed

theorem *arith-series-general*:
$((1{::}'a{::}comm\text{-}semiring\text{-}1) + 1) * (\sum i\in\{..<n\}.\ a + of\text{-}nat\ i * d) =$
$of\text{-}nat\ n * (a + (a + of\text{-}nat(n - 1)*d))$
proof *cases*
assume *ngt1*: $n > 1$
let $?I = \lambda i.\ of\text{-}nat\ i$ **and** $?n = of\text{-}nat\ n$
have
$(\sum i\in\{..<n\}.\ a+?I\ i*d) =$
$((\sum i\in\{..<n\}.\ a) + (\sum i\in\{..<n\}.\ ?I\ i*d))$
by (*rule setsum-addf*)
also from *ngt1* **have** $\ldots = ?n*a + (\sum i\in\{..<n\}.\ ?I\ i*d)$ **by** *simp*
also from *ngt1* **have** $\ldots = (?n*a + d*(\sum i\in\{1..<n\}.\ ?I\ i))$
by (*simp add*: *setsum-right-distrib setsum-head-upt mult-ac*)
also have $(1+1)*\ldots = (1+1)*?n*a + d*(1+1)*(\sum i\in\{1..<n\}.\ ?I\ i)$
by (*simp add*: *left-distrib right-distrib*)
also from *ngt1* **have** $\{1..<n\} = \{1..n - 1\}$
by (*cases n*) (*auto simp*: *atLeastLessThanSuc-atLeastAtMost*)
also from *ngt1*
have $(1+1)*?n*a + d*(1+1)*(\sum i\in\{1..n - 1\}.\ ?I\ i) = ((1+1)*?n*a + d*?I$
$(n - 1)*?I\ n)$
by (*simp only*: *mult-ac gauss-sum* [*of* $n - 1$])
(*simp add*: *mult-ac trans* [*OF add-commute of-nat-Suc* [*symmetric*]])
finally show *?thesis* **by** (*simp add*: *mult-ac add-ac right-distrib*)
next
assume $\neg(n > 1)$
hence $n = 1 \lor n = 0$ **by** *auto*
thus *?thesis* **by** (*auto simp*: *mult-ac right-distrib*)
qed

lemma *arith-series-nat*:
$Suc\ (Suc\ 0) * (\sum i\in\{..<n\}.\ a+i*d) = n * (a + (a+(n - 1)*d))$
proof −
have

$((1::nat) + 1) * (\sum i \in \{..<n::nat\}.\ a + of\text{-}nat(i)*d) =$
$of\text{-}nat(n) * (a + (a + of\text{-}nat(n - 1)*d))$
by (*rule arith-series-general*)
thus *?thesis* **by** (*auto simp add*: *of-nat-id*)
qed

lemma *arith-series-int*:
$(2::int) * (\sum i \in \{..<n\}.\ a + of\text{-}nat\ i * d) =$
$of\text{-}nat\ n * (a + (a + of\text{-}nat(n - 1)*d))$
proof –
have
$((1::int) + 1) * (\sum i \in \{..<n\}.\ a + of\text{-}nat\ i * d) =$
$of\text{-}nat(n) * (a + (a + of\text{-}nat(n - 1)*d))$
by (*rule arith-series-general*)
thus *?thesis* **by** *simp*
qed

lemma *sum-diff-distrib*:
fixes $P::nat \Rightarrow nat$
shows
$\forall x.\ Q\ x \leq P\ x \Longrightarrow$
$(\sum x<n.\ P\ x) - (\sum x<n.\ Q\ x) = (\sum x<n.\ P\ x - Q\ x)$
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*)

let $?lhs = (\sum x<n.\ P\ x) - (\sum x<n.\ Q\ x)$
let $?rhs = \sum x<n.\ P\ x - Q\ x$

from *Suc* **have** $?lhs = ?rhs$ **by** *simp*
moreover
from *Suc* **have** $?lhs + P\ n - Q\ n = ?rhs + (P\ n - Q\ n)$ **by** *simp*
moreover
from *Suc* **have**
$(\sum x<n.\ P\ x) + P\ n - ((\sum x<n.\ Q\ x) + Q\ n) = ?rhs + (P\ n - Q\ n)$
by (*subst diff-diff-left*[*symmetric*],
subst diff-add-assoc2)
(*auto simp*: *diff-add-assoc2 intro*: *setsum-mono*)
ultimately
show *?case* **by** *simp*
qed

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

40 Presburger: Decision Procedure for Presburger Arithmetic

theory *Presburger*
imports *Arith-Tools SetInterval*
uses
Tools/Qelim/cooper-data.ML
Tools/Qelim/generated-cooper.ML
(*Tools/Qelim/cooper.ML*)
(*Tools/Qelim/presburger.ML*)
begin

setup *CooperData.setup*

40.1 The $-\infty$ and $+\infty$ Properties

lemma *minf*:
$[\![\exists (z ::'a::linorder).\forall x<z.\ P\ x = P'\ x;\ \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z ::'a::linorder).\forall x<z.\ P\ x = P'\ x;\ \exists z.\forall x<z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x<z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x = t) = False$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \neq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x < t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \leq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x > t) = False$
$\exists (z ::'a::\{linorder\}).\forall x<z.(x \geq t) = False$
$\exists z.\forall (x::'a::\{linorder,plus,Divides.div\})<z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'a::\{linorder,plus,Divides.div\})<z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x<z.\ F = F$
by ((*erule exE*, *erule exE*,*rule-tac x=min z za* **in** *exI*,*simp*)+, (*rule-tac x=t* **in** *exI*,*fastsimp*)+) *simp-all*

lemma *pinf*:
$[\![\exists (z ::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z ::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x = t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \neq t) = True$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x < t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \leq t) = False$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x > t) = True$
$\exists (z ::'a::\{linorder\}).\forall x>z.(x \geq t) = True$
$\exists z.\forall (x::'a::\{linorder,plus,Divides.div\})>z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'a::\{linorder,plus,Divides.div\})>z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x>z.\ F = F$
by ((*erule exE*, *erule exE*,*rule-tac x=max z za* **in** *exI*,*simp*)+,(*rule-tac x=t* **in** *exI*,*fastsimp*)+) *simp-all*

lemma *inf-period*:
$[\![\forall\, x\, k.\ P\ x = P\ (x - k*D);\ \forall\, x\, k.\ Q\ x = Q\ (x - k*D)]\!]$
$\Longrightarrow \forall\, x\, k.\ (P\ x \wedge Q\ x) = (P\ (x - k*D) \wedge Q\ (x - k*D))$
$[\![\forall\, x\, k.\ P\ x = P\ (x - k*D);\ \forall\, x\, k.\ Q\ x = Q\ (x - k*D)]\!]$
$\Longrightarrow \forall\, x\, k.\ (P\ x \vee Q\ x) = (P\ (x - k*D) \vee Q\ (x - k*D))$
$(d::'a::\{comm\text{-}ring,Divides.div\})\ dvd\ D \Longrightarrow \forall\, x\, k.\ (d\ dvd\ x + t) = (d\ dvd\ (x - k*D) + t)$
$(d::'a::\{comm\text{-}ring,Divides.div\})\ dvd\ D \Longrightarrow \forall\, x\, k.\ (\neg d\ dvd\ x + t) = (\neg d\ dvd\ (x - k*D) + t)$
$\forall\, x\, k.\ F = F$
by *simp-all*
(*clarsimp simp add*: *dvd-def*, *rule iffI*, *clarsimp*,*rule-tac* $x = kb - ka*k$ **in** *exI*, *simp add*: *ring-simps*, *clarsimp*,*rule-tac* $x = kb + ka*k$ **in** *exI*,*simp add*: *ring-simps*)+

40.2 The A and B sets

lemma *bset*:
$[\![\forall x.(\forall j \in \{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)\ ;$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x - D) \wedge Q\ (x - D))$
$[\![\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)\ ;$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \vee Q\ x) \longrightarrow (P(x - D) \vee Q\ (x - D))$
$[\![D{>}0;\ t - 1{\in}\ B]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$
$D{>}0 \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$
$D{>}0 \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t))$
$[\![D{>}0\ ;\ t - 1 \in B]\!] \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x - D) + t))$
$d\ dvd\ D \Longrightarrow (\forall (x::int).(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (\neg d\ dvd\ x{+}t) \longrightarrow (\neg\ d\ dvd\ (x - D) + t))$
$\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow F \longrightarrow F$
proof (*blast*, *blast*)
assume *dp*: $D > 0$ **and** *tB*: $t - 1{\in}\ B$
show $(\forall x.(\forall j{\in}\{1\ ..\ D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** $x{=}1$],*erule ballE*[**where** $x{=}t - 1$])

using *dp tB* **by** *simp-all*
next
assume *dp*: $D > 0$ **and** *tB*: $t \in B$
show $(\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** *x*=*D*],*erule ballE*[**where** *x*=*t*])
using *dp tB* **by** *simp-all*
next
assume *dp*: $D > 0$ **thus** $(\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$ **by** *arith*
next
assume *dp*: $D > 0$ **thus** $\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t)$ **by** *arith*
next
assume *dp*: $D > 0$ **and** *tB*:$t \in B$
{fix *x* **assume** *nob*: $\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j$ **and** *g*: $x > t$ **and** *ng*: $\neg (x - D) > t$
hence $x - t \leq D$ **and** $1 \leq x - t$ **by** *simp+*
hence $\exists j \in \{1 .. D\}. x - t = j$ **by** *auto*
hence $\exists j \in \{1 .. D\}. x = t + j$ **by** (*simp add*: *ring-simps*)
with *nob tB* **have** *False* **by** *simp***}**
thus $\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t)$ **by** *blast*
next
assume *dp*: $D > 0$ **and** *tB*:$t - 1\in B$
{fix *x* **assume** *nob*: $\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j$ **and** *g*: $x \geq t$ **and** *ng*: $\neg (x - D) \geq t$
hence $x - (t - 1) \leq D$ **and** $1 \leq x - (t - 1)$ **by** *simp+*
hence $\exists j \in \{1 .. D\}. x - (t - 1) = j$ **by** *auto*
hence $\exists j \in \{1 .. D\}. x = (t - 1) + j$ **by** (*simp add*: *ring-simps*)
with *nob tB* **have** *False* **by** *simp***}**
thus $\forall x.(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t)$ **by** *blast*
next
assume *d*: *d dvd D*
{fix *x* **assume** *H*: *d dvd* $x + t$ **with** *d* **have** *d dvd* $(x - D) + t$
by (*clarsimp simp add*: *dvd-def*,*rule-tac* $x= ka - k$ **in** *exI*,*simp add*: *ring-simps*)**}**
thus $\forall (x{::}int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow (d\ dvd\ x+t) \longrightarrow (d\ dvd\ (x - D) + t)$ **by** *simp*
next
assume *d*: *d dvd D*
{fix *x* **assume** *H*: $\neg(d\ dvd\ x + t)$ **with** *d* **have** $\neg d\ dvd\ (x - D) + t$
by (*clarsimp simp add*: *dvd-def*,*erule-tac* $x= ka + k$ **in** *allE*,*simp add*: *ring-simps*)**}**
thus $\forall (x{::}int).(\forall j\in\{1 .. D\}. \forall b\in B. x \neq b + j) \longrightarrow (\neg d\ dvd\ x+t) \longrightarrow (\neg d\ dvd\ (x - D) + t)$ **by** *auto*
qed *blast*

lemma *aset*:
$[\![\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j) \longrightarrow P\ x \longrightarrow P(x + D)$;
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j) \longrightarrow Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$

$\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j) \longrightarrow (P x \wedge Q x) \longrightarrow (P(x + D) \wedge Q (x + D))$
$[\![\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow P x \longrightarrow P(x + D) ;$
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow Q x \longrightarrow Q(x + D)]\!] \Longrightarrow$
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (P x \vee Q x) \longrightarrow (P(x + D) \vee Q (x + D))$
$[\![D>0; t + 1\in A]\!] \Longrightarrow (\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x = t) \longrightarrow (x + D = t))$
$[\![D>0 ; t \in A]\!] \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x \neq t) \longrightarrow (x + D \neq t))$
$[\![D>0; t\in A]\!] \Longrightarrow(\forall (x::int). (\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x < t) \longrightarrow (x + D < t))$
$[\![D>0; t + 1 \in A]\!] \Longrightarrow (\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x \leq t) \longrightarrow (x + D \leq t))$
$D>0 \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x > t) \longrightarrow (x + D > t))$
$D>0 \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x \geq t) \longrightarrow (x + D \geq t))$
$d\ dvd\ D \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (d\ dvd\ x+t) \longrightarrow (d\ dvd\ (x + D) + t))$
$d\ dvd\ D \Longrightarrow(\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (\neg d\ dvd\ x+t) \longrightarrow (\neg\ d\ dvd\ (x + D) + t))$
$\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j) \longrightarrow F \longrightarrow F$
proof (*blast*, *blast*)
assume *dp*: $D > 0$ **and** *tA*: $t + 1 \in A$
show $(\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x = t) \longrightarrow (x + D = t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** $x=1$],*erule ballE*[**where** $x=t + 1$])
using *dp tA* **by** *simp-all*
next
assume *dp*: $D > 0$ **and** *tA*: $t \in A$
show $(\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x \neq t) \longrightarrow (x + D \neq t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** $x=D$],*erule ballE*[**where** $x=t$])
using *dp tA* **by** *simp-all*
next
assume *dp*: $D > 0$ **thus** $(\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x > t) \longrightarrow (x + D > t))$ **by** *arith*
next
assume *dp*: $D > 0$ **thus** $\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x \geq t) \longrightarrow (x + D \geq t)$ **by** *arith*
next
assume *dp*: $D > 0$ **and** *tA*:$t \in A$
{fix x **assume** *nob*: $\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j$ **and** *g*: $x < t$ **and** *ng*: $\neg (x + D) < t$
hence $t - x \leq D$ **and** $1 \leq t - x$ **by** *simp+*
hence $\exists j \in \{1 .. D\}. t - x = j$ **by** *auto*
hence $\exists j \in \{1 .. D\}. x = t - j$ **by** (*auto simp add*: *ring-simps*)
with *nob tA* **have** *False* **by** *simp***}**
thus $\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x < t) \longrightarrow (x + D < t)$ **by** *blast*

next
assume *dp*: $D > 0$ **and** *tA*:$t + 1 \in A$
{fix x **assume** *nob*: $\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j$ **and** *g*: $x \leq t$ **and** *ng*: $\neg (x + D) \leq t$
hence $(t + 1) - x \leq D$ **and** $1 \leq (t + 1) - x$ **by** (*simp-all add*: *ring-simps*)
hence $\exists j \in \{1 .. D\}. (t + 1) - x = j$ **by** *auto*
hence $\exists j \in \{1 .. D\}. x = (t + 1) - j$ **by** (*auto simp add*: *ring-simps*)
with *nob tA* **have** *False* **by** *simp***}**
thus $\forall x.(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (x \leq t) \longrightarrow (x + D \leq t)$ **by** *blast*
next
assume *d*: *d dvd D*
{fix x **assume** *H*: *d dvd* $x + t$ **with** d **have** *d dvd* $(x + D) + t$
by (*clarsimp simp add*: *dvd-def*,*rule-tac* $x= ka + k$ **in** *exI*,*simp add*: *ring-simps*)**}**
thus $\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (d\ dvd\ x+t) \longrightarrow (d\ dvd\ (x + D) + t)$ **by** *simp*
next
assume *d*: *d dvd D*
{fix x **assume** *H*: $\neg(d\ dvd\ x + t)$ **with** d **have** $\neg d\ dvd\ (x + D) + t$
by (*clarsimp simp add*: *dvd-def*,*erule-tac* $x= ka - k$ **in** *allE*,*simp add*: *ring-simps*)**}**
thus $\forall (x::int).(\forall j\in\{1 .. D\}. \forall b\in A. x \neq b - j)\longrightarrow (\neg d\ dvd\ x+t) \longrightarrow (\neg d\ dvd\ (x + D) + t)$ **by** *auto*
qed *blast*

40.3 Cooper's Theorem $-\infty$ and $+\infty$ Version

40.3.1 First some trivial facts about periodic sets or predicates

lemma *periodic-finite-ex*:
assumes *dpos*: $(0::int) < d$ **and** *modd*: *ALL x k*. $P\ x = P(x - k*d)$
shows $(EX\ x.\ P\ x) = (EX\ j : \{1..d\}.\ P\ j)$
(**is** *?LHS* = *?RHS*)
proof
assume *?LHS*
then obtain x **where** *P*: $P\ x$ **..**
have $x\ mod\ d = x - (x\ div\ d)*d$ **by**(*simp add*:*zmod-zdiv-equality mult-ac eq-diff-eq*)
hence *Pmod*: $P\ x = P(x\ mod\ d)$ **using** *modd* **by** *simp*
show *?RHS*
proof (*cases*)
assume $x\ mod\ d = 0$
hence $P\ 0$ **using** *P Pmod* **by** *simp*
moreover have $P\ 0 = P(0 - (-1)*d)$ **using** *modd* **by** *blast*
ultimately have $P\ d$ **by** *simp*
moreover have $d : \{1..d\}$ **using** *dpos* **by**(*simp add*:*atLeastAtMost-iff*)
ultimately show *?RHS* **..**
next
assume *not0*: $x\ mod\ d \neq 0$
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 −
from *dpos* **have** $0 \leq x \ mod \ d$ **by**(*rule pos-mod-sign*)
moreover from *dpos* **have** $x \ mod \ d < d$ **by**(*rule pos-mod-bound*)
ultimately show *?thesis* **using** *not0* **by**(*simp add*:*atLeastAtMost-iff*)
qed
ultimately show *?RHS* **..**
qed
qed *auto*

40.3.2 The $-\infty$ Version

lemma *decr-lemma*: $0 < (d::int) \Longrightarrow x - (abs(x-z)+1) * d < z$
by(*induct rule*: *int-gr-induct*,*simp-all add*:*int-distrib*)

lemma *incr-lemma*: $0 < (d::int) \Longrightarrow z < x + (abs(x-z)+1) * d$
by(*induct rule*: *int-gr-induct*, *simp-all add*:*int-distrib*)

theorem *int-induct*[*case-names base step1 step2*]:
assumes
base: $P(k::int)$ **and** *step1*: $\bigwedge i.\ [\![k \leq i;\ P\ i]\!] \Longrightarrow P(i+1)$ **and**
step2: $\bigwedge i.\ [\![k \geq i;\ P\ i]\!] \Longrightarrow P(i - 1)$
shows $P\ i$
proof −
have $i \leq k \vee i \geq k$ **by** *arith*
thus *?thesis* **using** *prems int-ge-induct*[**where** P=P **and** k=k **and** i=i] *int-le-induct*[**where** P=P **and** k=k **and** i=i] **by** *blast*
qed

lemma *decr-mult-lemma*:
assumes *dpos*: $(0::int) < d$ **and** *minus*: $\forall x.\ P\ x \longrightarrow P(x - d)$ **and** *knneg*: 0 $<= k$
shows $ALL\ x.\ P\ x \longrightarrow P(x - k*d)$
using *knneg*
proof (*induct rule*:*int-ge-induct*)
case *base* **thus** *?case* **by** *simp*
next
case (*step i*)
{fix x
have $P\ x \longrightarrow P\ (x - i * d)$ **using** *step.hyps* **by** *blast*
also have $\ldots \longrightarrow P(x - (i + 1) * d)$ **using** *minus*[*THEN spec*, *of* $x - i * d$]
by (*simp add*:*int-distrib OrderedGroup.diff-diff-eq*[*symmetric*])
ultimately have $P\ x \longrightarrow P(x - (i + 1) * d)$ **by** *blast***}**
thus *?case* **..**
qed

lemma *minusinfinity*:
assumes *dpos*: $0 < d$ **and**
P1eqP1: $ALL\ x\ k.\ P1\ x = P1(x - k*d)$ **and** *ePeqP1*: $EX\ z::int.\ ALL\ x.\ x <$ $z \longrightarrow (P\ x = P1\ x)$

shows $(EX\ x.\ P1\ x) \longrightarrow (EX\ x.\ P\ x)$
proof
assume *eP1*: $EX\ x.\ P1\ x$
then obtain x **where** *P1*: $P1\ x$ **..**
from *ePeqP1* **obtain** z **where** *P1eqP*: $ALL\ x.\ x < z \longrightarrow (P\ x = P1\ x)$ **..**
let $?w = x - (abs(x-z)+1) * d$
from *dpos* **have** *w*: $?w < z$ **by**(*rule decr-lemma*)
have $P1\ x = P1\ ?w$ **using** *P1eqP1* **by** *blast*
also have $\ldots = P(?w)$ **using** *w P1eqP* **by** *blast*
finally have $P\ ?w$ **using** *P1* **by** *blast*
thus $EX\ x.\ P\ x$ **..**
qed

lemma *cpmi*:
assumes *dp*: $0 < D$ **and** *p1*:$\exists z.\ \forall\ x< z.\ P\ x = P'\ x$
and *nb*:$\forall x.(\forall\ j\in \{1..D\}.\ \forall(b::int) \in B.\ x \neq b+j) \;{-}{-}{>}\; P\ (x) \;{-}{-}{>}\; P\ (x - D)$
and *pd*: $\forall\ x\ k.\ P'\ x = P'\ (x-k*D)$
shows $(\exists x.\ P\ x) = ((\exists\ j\in \{1..D\}\ .\ P'\ j) \mid (\exists\ j \in \{1..D\}.\exists\ b\in B.\ P\ (b+j)))$
(**is** $?L = (?R1 \lor ?R2)$)
proof−
{**assume** $?R2$ **hence** $?L$ **by** *blast*}
moreover
{**assume** H:$?R1$ **hence** $?L$ **using** *minusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] **by** *simp*}
moreover
{ **fix** x
assume *P*: $P\ x$ **and** *H*: $\neg\ ?R2$
{**fix** y **assume** $\neg\ (\exists j\in\{1..D\}.\ \exists b\in B.\ P\ (b + j))$ **and** *P*: $P\ y$
hence $\sim(EX\ (j::int) : \{1..D\}.\ EX\ (b::int) : B.\ y = b+j)$ **by** *auto*
with *nb P* **have** $P\ (y - D)$ **by** *auto* }
hence $ALL\ x.\sim(EX\ (j::int) : \{1..D\}.\ EX\ (b::int) : B.\ P(b+j)) \;{-}{-}{>}\; P\ (x) \;{-}{-}{>}\; P\ (x - D)$ **by** *blast*
with *H P* **have** *th*: $\forall x.\ P\ x \longrightarrow P\ (x - D)$ **by** *auto*
from *p1* **obtain** z **where** *z*: $ALL\ x.\ x < z \;{-}{-}{>}\; (P\ x = P'\ x)$ **by** *blast*
let $?y = x - (|x - z| + 1)*D$
have *zp*: $0 <= (|x - z| + 1)$ **by** *arith*
from *dp* **have** *yz*: $?y < z$ **using** *decr-lemma*[*OF dp*] **by** *simp*
from *z*[*rule-format*, *OF yz*] *decr-mult-lemma*[*OF dp th zp*, *rule-format*, *OF P*] **have** *th2*: $P'\ ?y$ **by** *auto*
with *periodic-finite-ex*[*OF dp pd*]
have $?R1$ **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

40.3.3 The $+\infty$ Version

lemma *plusinfinity*:
assumes *dpos*: $(0::int) < d$ **and**

P1eqP1: $\forall\, x\, k.\ P'\ x = P'(x - k{*}d)$ **and** *ePeqP1*: $\exists\ z.\ \forall\ x{>}z.\ P\ x = P'\ x$
shows $(\exists\ x.\ P'\ x) \longrightarrow (\exists\ x.\ P\ x)$
proof
assume *eP1*: $EX\ x.\ P'\ x$
then obtain x **where** *P1*: $P'\ x$ **..**
from *ePeqP1* **obtain** z **where** *P1eqP*: $\forall\, x{>}z.\ P\ x = P'\ x$ **..**
let $?w' = x + (abs(x{-}z){+}1) * d$
let $?w = x - (-(abs(x{-}z) + 1)){*}d$
have *ww'*[*simp*]: $?w = ?w'$ **by** (*simp add*: *ring-simps*)
from *dpos* **have** *w*: $?w > z$ **by**(*simp only*: *ww' incr-lemma*)
hence $P'\ x = P'\ ?w$ **using** *P1eqP1* **by** *blast*
also have $\ldots = P(?w)$ **using** *w P1eqP* **by** *blast*
finally have $P\ ?w$ **using** *P1* **by** *blast*
thus $EX\ x.\ P\ x$ **..**
qed

lemma *incr-mult-lemma*:
assumes *dpos*: $(0{::}int) < d$ **and** *plus*: $ALL\ x{::}int.\ P\ x \longrightarrow P(x + d)$ **and** *knneg*: $0 <= k$
shows $ALL\ x.\ P\ x \longrightarrow P(x + k{*}d)$
using *knneg*
proof (*induct rule*:*int-ge-induct*)
case *base* **thus** *?case* **by** *simp*
next
case (*step i*)
{fix x
have $P\ x \longrightarrow P\ (x + i * d)$ **using** *step.hyps* **by** *blast*
also have $\ldots \longrightarrow P(x + (i + 1) * d)$ **using** *plus*[*THEN spec*, *of* $x + i * d$]
by (*simp add*:*int-distrib zadd-ac*)
ultimately have $P\ x \longrightarrow P(x + (i + 1) * d)$ **by** *blast***}**
thus *?case* **..**
qed

lemma *cppi*:
assumes *dp*: $0 < D$ **and** *p1*:$\exists\, z.\ \forall\ x{>}\ z.\ P\ x = P'\ x$
and *nb*:$\forall\, x.(\forall\ j{\in}\ \{1..D\}.\ \forall\,(b{::}int) \in A.\ x \neq b - j)$ --> $P\ (x)$ --> $P\ (x + D)$
and *pd*: $\forall\ x\, k.\ P'\ x{=}\ P'\ (x{-}k{*}D)$
shows $(\exists\, x.\ P\ x) = ((\exists\ j{\in}\ \{1..D\}\ .\ P'\ j) \mid (\exists\ j \in \{1..D\}.\exists\ b{\in}\ A.\ P\ (b - j)))$
(**is** $?L = (?R1 \lor ?R2)$)
proof−
{assume *?R2* **hence** *?L* **by** *blast***}**
moreover
{assume *H*:*?R1* **hence** *?L* **using** *plusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] **by** *simp***}**
moreover
{ fix x
assume *P*: $P\ x$ **and** *H*: $\neg\ ?R2$
{fix y **assume** $\neg\ (\exists\, j{\in}\{1..D\}.\ \exists\, b{\in}A.\ P\ (b - j))$ **and** *P*: $P\ y$

hence ~$(EX\ (j::int) : \{1..D\}.\ EX\ (b::int) : A.\ y = b - j)$ **by** *auto*
with *nb P* **have** $P\ (y + D)$ **by** *auto* **}**
hence $ALL\ x.$~$(EX\ (j::int) : \{1..D\}.\ EX\ (b::int) : A.\ P(b-j)) --> P\ (x) --> P\ (x + D)$ **by** *blast*
with *H P* **have** *th*: $\forall x.\ P\ x \longrightarrow P\ (x + D)$ **by** *auto*
from *p1* **obtain** *z* **where** *z*: $ALL\ x.\ x > z --> (P\ x = P'\ x)$ **by** *blast*
let $?y = x + (|x - z| + 1)*D$
have *zp*: $0 <= (|x - z| + 1)$ **by** *arith*
from *dp* **have** *yz*: $?y > z$ **using** *incr-lemma*[*OF dp*] **by** *simp*
from *z*[*rule-format*, *OF yz*] *incr-mult-lemma*[*OF dp th zp*, *rule-format*, *OF P*] **have** *th2*: $P'\ ?y$ **by** *auto*
with *periodic-finite-ex*[*OF dp pd*]
have *?R1* **by** *blast*}
ultimately show *?thesis* **by** *blast*
qed

lemma *simp-from-to*: $\{i..j::int\} = (if\ j < i\ then\ \{\}\ else\ insert\ i\ \{i+1..j\})$
apply(*simp add*:*atLeastAtMost-def atLeast-def atMost-def*)
apply(*fastsimp*)
done

theorem *unity-coeff-ex*: $(\exists (x::'a::\{semiring\text{-}0,Divides.div\}).\ P\ (l * x)) \equiv (\exists x.\ l\ dvd\ (x + 0) \wedge P\ x)$
apply (*rule eq-reflection*[*symmetric*])
apply (*rule iffI*)
defer
apply (*erule exE*)
apply (*rule-tac* $x = l * x$ **in** *exI*)
apply (*simp add*: *dvd-def*)
apply (*rule-tac* $x{=}x$ **in** *exI*, *simp*)
apply (*erule exE*)
apply (*erule conjE*)
apply (*erule dvdE*)
apply (*rule-tac* $x = k$ **in** *exI*)
apply *simp*
done

lemma *zdvd-mono*: **assumes** *not0*: $(k::int) \neq 0$
shows $((m::int)\ dvd\ t) \equiv (k*m\ dvd\ k*t)$
using *not0* **by** (*simp add*: *dvd-def*)

lemma *uminus-dvd-conv*: $(d\ dvd\ (t::int)) \equiv (-d\ dvd\ t)$ $(d\ dvd\ (t::int)) \equiv (d\ dvd\ -t)$
by *simp-all*

Theorems for transforming predicates on nat to predicates on *int*

lemma *all-nat*: $(\forall x::nat.\ P\ x) = (\forall x::int.\ 0 <= x \longrightarrow P\ (nat\ x))$
by (*simp split add*: *split-nat*)

lemma *ex-nat*: $(\exists\, x{::}nat.\ P\ x) = (\exists\, x{::}int.\ 0 <= x \wedge P\ (nat\ x))$
apply (*auto split add*: *split-nat*)
apply (*rule-tac x=int x* **in** *exI*, *simp*)
apply (*rule-tac x = nat x* **in** *exI*,*erule-tac x = nat x* **in** *allE*, *simp*)
done

lemma *zdiff-int-split*: $P\ (int\ (x - y)) =$
$((y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0))$
by (*case-tac* $y \leq x$, *simp-all add*: *zdiff-int*)

lemma *number-of1*: $(0{::}int) <= number\text{-}of\ n \Longrightarrow (0{::}int) <= number\text{-}of\ (n\ BIT\ b)$ **by** *simp*
lemma *number-of2*: $(0{::}int) <= Numeral0$ **by** *simp*
lemma *Suc-plus1*: $Suc\ n = n + 1$ **by** *simp*

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*: $(0 <= x \Longrightarrow P = P') \Longrightarrow (0 <= (x{::}int) \longrightarrow P) = (0 <= x \longrightarrow P')$ **by** *simp*

theorem *conj-le-cong*: $(0 <= x \Longrightarrow P = P') \Longrightarrow (0 <= (x{::}int) \wedge P) = (0 <= x \wedge P')$
by (*simp cong*: *conj-cong*)
lemma *int-eq-number-of-eq*:
$(((number\text{-}of\ v){::}int) = (number\text{-}of\ w)) = iszero\ ((number\text{-}of\ (v + (uminus\ w))){::}int)$
by *simp*

lemma *mod-eq0-dvd-iff* [*presburger*]: $(m{::}nat)\ mod\ n = 0 \longleftrightarrow n\ dvd\ m$
unfolding *dvd-eq-mod-eq-0* [*symmetric*] **..**

lemma *zmod-eq0-zdvd-iff* [*presburger*]: $(m{::}int)\ mod\ n = 0 \longleftrightarrow n\ dvd\ m$
unfolding *zdvd-iff-zmod-eq-0* [*symmetric*] **..**
declare *mod-1* [*presburger*]
declare *mod-0* [*presburger*]
declare *zmod-1* [*presburger*]
declare *zmod-zero* [*presburger*]
declare *zmod-self* [*presburger*]
declare *mod-self* [*presburger*]
declare *DIVISION-BY-ZERO-MOD* [*presburger*]
declare *nat-mod-div-trivial* [*presburger*]
declare *div-mod-equality2* [*presburger*]
declare *div-mod-equality* [*presburger*]
declare *mod-div-equality2* [*presburger*]
declare *mod-div-equality* [*presburger*]
declare *mod-mult-self1* [*presburger*]
declare *mod-mult-self2* [*presburger*]
declare *zdiv-zmod-equality2* [*presburger*]
declare *zdiv-zmod-equality* [*presburger*]

declare *mod2-Suc-Suc*[*presburger*]
lemma [*presburger*]: (*a*::*int*) *div 0* = *0* **and** [*presburger*]: *a mod 0* = *a*
using *IntDiv.DIVISION-BY-ZERO* **by** *blast+*

use *Tools/Qelim/cooper.ML*
oracle *linzqe-oracle* (*term*) = *Coopereif.cooper-oracle*

use *Tools/Qelim/presburger.ML*

declaration ⟪ *fn - =>*
arith-tactic-add
(*mk-arith-tactic presburger* (*fn ctxt => fn i => fn st =>*
(*warning Trying Presburger arithmetic ...*;
Presburger.cooper-tac true [] [] *ctxt i st*)))
⟫

method-setup *presburger* = ⟪
let
fun keyword k = *Scan.lift* (*Args.$$$ k −− Args.colon*) >> *K* ()
fun simple-keyword k = *Scan.lift* (*Args.$$$ k*) >> *K* ()
val addN = *add*
val delN = *del*
val elimN = *elim*
val any-keyword = *keyword addN* || *keyword delN* || *simple-keyword elimN*
val thms = *Scan.repeat* (*Scan.unless any-keyword Attrib.multi-thm*) >> *flat*;
in
fn src => Method.syntax
((*Scan.optional* (*simple-keyword elimN* >> *K false*) *true*) −−
(*Scan.optional* (*keyword addN* |−− *thms*) []) −−
(*Scan.optional* (*keyword delN* |−− *thms*) [])) *src*
#> (*fn* (((*elim*, *add-ths*), *del-ths*),*ctxt*) =>
Method.SIMPLE-METHOD′ (*Presburger.cooper-tac elim add-ths del-ths ctxt*))
end
⟫ *Cooper′s algorithm for Presburger arithmetic*

lemma [*presburger*]: $m \ mod\ 2 = (1::nat) \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*
lemma [*presburger*]: $m \ mod\ 2 = Suc\ 0 \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*
lemma [*presburger*]: $m \ mod\ (Suc\ (Suc\ 0)) = (1::nat) \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*
lemma [*presburger*]: $m \ mod\ (Suc\ (Suc\ 0)) = Suc\ 0 \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*
lemma [*presburger*]: $m \ mod\ 2 = (1::int) \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*

lemma *zdvd-period*:
fixes *a d* :: *int*
assumes *advdd*: *a dvd d*
shows $a\ dvd\ (x + t) \longleftrightarrow a\ dvd\ ((x + c * d) + t)$
proof−
{

  fix *x k*
  from *inf-period*(*3*) [*OF advdd*, *rule-format*, **where** *x*=*x* **and** *k*=−*k*]
  have *a dvd* (*x* + *t*) ⟷ *a dvd* (*x* + *k* ∗ *d* + *t*) **by** *simp*
 }
 hence ∀ *x*.∀ *k*. ((*a*::*int*) *dvd* (*x* + *t*)) = (*a dvd* (*x*+*k*∗*d* + *t*)) **by** *simp*
 then show *?thesis* **by** *simp*
qed

40.4 Code generator setup

Presburger arithmetic is convenient to prove some of the following code lemmas on integer numerals:

lemma *eq-Pls-Pls*:
 Numeral.Pls = *Numeral.Pls* ⟷ *True* **by** *presburger*

lemma *eq-Pls-Min*:
 Numeral.Pls = *Numeral.Min* ⟷ *False*
 unfolding *Pls-def Numeral.Min-def* **by** *presburger*

lemma *eq-Pls-Bit0*:
 Numeral.Pls = *Numeral.Bit k bit.B0* ⟷ *Numeral.Pls* = *k*
 unfolding *Pls-def Bit-def bit.cases* **by** *presburger*

lemma *eq-Pls-Bit1*:
 Numeral.Pls = *Numeral.Bit k bit.B1* ⟷ *False*
 unfolding *Pls-def Bit-def bit.cases* **by** *presburger*

lemma *eq-Min-Pls*:
 Numeral.Min = *Numeral.Pls* ⟷ *False*
 unfolding *Pls-def Numeral.Min-def* **by** *presburger*

lemma *eq-Min-Min*:
 Numeral.Min = *Numeral.Min* ⟷ *True* **by** *presburger*

lemma *eq-Min-Bit0*:
 Numeral.Min = *Numeral.Bit k bit.B0* ⟷ *False*
 unfolding *Numeral.Min-def Bit-def bit.cases* **by** *presburger*

lemma *eq-Min-Bit1*:
 Numeral.Min = *Numeral.Bit k bit.B1* ⟷ *Numeral.Min* = *k*
 unfolding *Numeral.Min-def Bit-def bit.cases* **by** *presburger*

lemma *eq-Bit0-Pls*:
 Numeral.Bit k bit.B0 = *Numeral.Pls* ⟷ *Numeral.Pls* = *k*
 unfolding *Pls-def Bit-def bit.cases* **by** *presburger*

lemma *eq-Bit1-Pls*:
 Numeral.Bit k bit.B1 = *Numeral.Pls* ⟷ *False*
 unfolding *Pls-def Bit-def bit.cases* **by** *presburger*

lemma *eq-Bit0-Min*:
$Numeral.Bit\ k\ bit.B0 = Numeral.Min \longleftrightarrow False$
unfolding *Numeral.Min-def Bit-def bit.cases* **by** *presburger*

lemma *eq-Bit1-Min*:
$(Numeral.Bit\ k\ bit.B1) = Numeral.Min \longleftrightarrow Numeral.Min = k$
unfolding *Numeral.Min-def Bit-def bit.cases* **by** *presburger*

lemma *eq-Bit-Bit*:
$Numeral.Bit\ k1\ v1 = Numeral.Bit\ k2\ v2 \longleftrightarrow$
$v1 = v2 \wedge k1 = k2$
unfolding *Bit-def*
apply (*cases v1*)
apply (*cases v2*)
apply *auto*
apply *presburger*
apply (*cases v2*)
apply *auto*
apply *presburger*
apply (*cases v2*)
apply *auto*
done

lemma *eq-number-of*:
$(number\text{-}of\ k :: int) = number\text{-}of\ l \longleftrightarrow k = l$
unfolding *number-of-is-id* **..**

lemma *less-eq-Pls-Pls*:
$Numeral.Pls \le Numeral.Pls \longleftrightarrow True$ **by** *rule+*

lemma *less-eq-Pls-Min*:
$Numeral.Pls \le Numeral.Min \longleftrightarrow False$
unfolding *Pls-def Numeral.Min-def* **by** *presburger*

lemma *less-eq-Pls-Bit*:
$Numeral.Pls \le Numeral.Bit\ k\ v \longleftrightarrow Numeral.Pls \le k$
unfolding *Pls-def Bit-def* **by** (*cases v*) *auto*

lemma *less-eq-Min-Pls*:
$Numeral.Min \le Numeral.Pls \longleftrightarrow True$
unfolding *Pls-def Numeral.Min-def* **by** *presburger*

lemma *less-eq-Min-Min*:
$Numeral.Min \le Numeral.Min \longleftrightarrow True$ **by** *rule+*

lemma *less-eq-Min-Bit0*:
$Numeral.Min \le Numeral.Bit\ k\ bit.B0 \longleftrightarrow Numeral.Min < k$

unfolding *Numeral.Min-def Bit-def* **by** *auto*

lemma *less-eq-Min-Bit1*:
Numeral.Min $\leq$ *Numeral.Bit k bit.B1* $\longleftrightarrow$ *Numeral.Min* $\leq$ *k*
unfolding *Numeral.Min-def Bit-def* **by** *auto*

lemma *less-eq-Bit0-Pls*:
Numeral.Bit k bit.B0 $\leq$ *Numeral.Pls* $\longleftrightarrow$ *k* $\leq$ *Numeral.Pls*
unfolding *Pls-def Bit-def* **by** *simp*

lemma *less-eq-Bit1-Pls*:
Numeral.Bit k bit.B1 $\leq$ *Numeral.Pls* $\longleftrightarrow$ *k* $<$ *Numeral.Pls*
unfolding *Pls-def Bit-def* **by** *auto*

lemma *less-eq-Bit-Min*:
Numeral.Bit k v $\leq$ *Numeral.Min* $\longleftrightarrow$ *k* $\leq$ *Numeral.Min*
unfolding *Numeral.Min-def Bit-def* **by** (*cases v*) *auto*

lemma *less-eq-Bit0-Bit*:
Numeral.Bit k1 bit.B0 $\leq$ *Numeral.Bit k2 v* $\longleftrightarrow$ *k1* $\leq$ *k2*
unfolding *Bit-def bit.cases* **by** (*cases v*) *auto*

lemma *less-eq-Bit-Bit1*:
Numeral.Bit k1 v $\leq$ *Numeral.Bit k2 bit.B1* $\longleftrightarrow$ *k1* $\leq$ *k2*
unfolding *Bit-def bit.cases* **by** (*cases v*) *auto*

lemma *less-eq-Bit1-Bit0*:
Numeral.Bit k1 bit.B1 $\leq$ *Numeral.Bit k2 bit.B0* $\longleftrightarrow$ *k1* $<$ *k2*
unfolding *Bit-def* **by** (*auto split*: *bit.split*)

lemma *less-eq-number-of*:
(*number-of k* :: *int*) $\leq$ *number-of l* $\longleftrightarrow$ *k* $\leq$ *l*
unfolding *number-of-is-id* **..**

lemma *less-Pls-Pls*:
Numeral.Pls $<$ *Numeral.Pls* $\longleftrightarrow$ *False* **by** *simp*

lemma *less-Pls-Min*:
Numeral.Pls $<$ *Numeral.Min* $\longleftrightarrow$ *False*
unfolding *Pls-def Numeral.Min-def* **by** *presburger*

lemma *less-Pls-Bit0*:
Numeral.Pls $<$ *Numeral.Bit k bit.B0* $\longleftrightarrow$ *Numeral.Pls* $<$ *k*
unfolding *Pls-def Bit-def* **by** *auto*

lemma *less-Pls-Bit1*:
Numeral.Pls $<$ *Numeral.Bit k bit.B1* $\longleftrightarrow$ *Numeral.Pls* $\leq$ *k*
unfolding *Pls-def Bit-def* **by** *auto*

lemma *less-Min-Pls*:
Numeral.Min < *Numeral.Pls* ⟷ *True*
unfolding *Pls-def Numeral.Min-def* **by** *presburger*

lemma *less-Min-Min*:
Numeral.Min < *Numeral.Min* ⟷ *False* **by** *simp*

lemma *less-Min-Bit*:
Numeral.Min < *Numeral.Bit k v* ⟷ *Numeral.Min* < *k*
unfolding *Numeral.Min-def Bit-def* **by** (*auto split*: *bit.split*)

lemma *less-Bit-Pls*:
Numeral.Bit k v < *Numeral.Pls* ⟷ *k* < *Numeral.Pls*
unfolding *Pls-def Bit-def* **by** (*auto split*: *bit.split*)

lemma *less-Bit0-Min*:
Numeral.Bit k bit.B0 < *Numeral.Min* ⟷ *k* ≤ *Numeral.Min*
unfolding *Numeral.Min-def Bit-def* **by** *auto*

lemma *less-Bit1-Min*:
Numeral.Bit k bit.B1 < *Numeral.Min* ⟷ *k* < *Numeral.Min*
unfolding *Numeral.Min-def Bit-def* **by** *auto*

lemma *less-Bit-Bit0*:
Numeral.Bit k1 v < *Numeral.Bit k2 bit.B0* ⟷ *k1* < *k2*
unfolding *Bit-def* **by** (*auto split*: *bit.split*)

lemma *less-Bit1-Bit*:
Numeral.Bit k1 bit.B1 < *Numeral.Bit k2 v* ⟷ *k1* < *k2*
unfolding *Bit-def* **by** (*auto split*: *bit.split*)

lemma *less-Bit0-Bit1*:
Numeral.Bit k1 bit.B0 < *Numeral.Bit k2 bit.B1* ⟷ *k1* ≤ *k2*
unfolding *Bit-def bit.cases* **by** *arith*

lemma *less-number-of*:
(*number-of k* :: *int*) < *number-of l* ⟷ *k* < *l*
unfolding *number-of-is-id* **..**

lemmas *pred-succ-numeral-code* [*code func*] =
arith-simps(*5*−*12*)

lemmas *plus-numeral-code* [*code func*] =
arith-simps(*13*−*17*)
arith-simps(*26*−*27*)
arith-extra-simps(*1*) [**where** *'a* = *int*]

lemmas *minus-numeral-code* [*code func*] =

arith-simps(*18*−*21*)
arith-extra-simps(*2*) [**where** $'a$ = *int*]
arith-extra-simps(*5*) [**where** $'a$ = *int*]

lemmas *times-numeral-code* [*code func*] =
arith-simps(*22*−*25*)
arith-extra-simps(*4*) [**where** $'a$ = *int*]

lemmas *eq-numeral-code* [*code func*] =
eq-Pls-Pls eq-Pls-Min eq-Pls-Bit0 eq-Pls-Bit1
eq-Min-Pls eq-Min-Min eq-Min-Bit0 eq-Min-Bit1
eq-Bit0-Pls eq-Bit1-Pls eq-Bit0-Min eq-Bit1-Min eq-Bit-Bit
eq-number-of

lemmas *less-eq-numeral-code* [*code func*] = *less-eq-Pls-Pls less-eq-Pls-Min less-eq-Pls-Bit*
less-eq-Min-Pls less-eq-Min-Min less-eq-Min-Bit0 less-eq-Min-Bit1
less-eq-Bit0-Pls less-eq-Bit1-Pls less-eq-Bit-Min less-eq-Bit0-Bit less-eq-Bit-Bit1
less-eq-Bit1-Bit0
less-eq-number-of

lemmas *less-numeral-code* [*code func*] = *less-Pls-Pls less-Pls-Min less-Pls-Bit0*
less-Pls-Bit1 less-Min-Pls less-Min-Min less-Min-Bit less-Bit-Pls
less-Bit0-Min less-Bit1-Min less-Bit-Bit0 less-Bit1-Bit less-Bit0-Bit1
less-number-of

context *ring-1*
begin

lemma *of-int-num* [*code func*]:
of-int k = (*if* $k = 0$ *then* 0 *else if* $k < 0$ *then*
− *of-int* (− k) *else let*
(l, m) = *divAlg* (k, 2);
l' = *of-int* l
in if $m = 0$ *then* $l' + l'$ *else* $l' + l' + 1$)
proof −
have *aux1*: k *mod* (2::*int*) ≠ (0::*int*) ⟹
of-int k = *of-int* (k *div* $2 * 2 + 1$)
proof −
assume k *mod* $2 \neq 0$
then have k *mod* $2 = 1$ **by** *arith*
moreover have *of-int* k = *of-int* (k *div* $2 * 2 + k$ *mod* 2) **by** *simp*
ultimately show *?thesis* **by** *auto*
qed
have *aux2*: $\bigwedge x$. *of-int* $2 * x = x + x$
proof −
fix x
have *int2*: (2::*int*) = $1 + 1$ **by** *arith*
show *of-int* $2 * x = x + x$
unfolding *int2 of-int-add left-distrib* **by** *simp*

  qed
  have *aux3*: $\bigwedge x.\ x * \textit{of-int}\ 2 = x + x$
  proof −
    fix *x*
    have *int2*: $(2::int) = 1 + 1$ **by** *arith*
    show $x * \textit{of-int}\ 2 = x + x$
    unfolding *int2 of-int-add right-distrib* **by** *simp*
  qed
  from *aux1* **show** *?thesis* **by** (*auto simp add*: *divAlg-mod-div Let-def aux2 aux3*)
qed

end

end

41 Relation-Power: Powers of Relations and Functions

theory *Relation-Power*
imports *Power*
begin

instance
  set :: (*type*) *power* **..**
    — only type $('a \times 'a)$ *set* should be in class *power*!

primrec (**unchecked** *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 (**unchecked** *funpow*)
  $f\hat{\ }0 = id$
  $f\hat{\ }(Suc\ n) = f\ o\ (f\hat{\ }n)$

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.

Circumvent this problem for code generation:

```
definition
  funpow :: nat ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
where
  funpow-def: funpow n f = f ^ n

lemmas [code inline] = funpow-def [symmetric]

lemma [code func]:
  funpow 0 f = id
  funpow (Suc n) f = f o funpow n f
  unfolding funpow-def by simp-all

lemma funpow-add: f ^ (m+n) = f^m o f^n
  by (induct m) simp-all

lemma funpow-swap1: f((f^n) x) = (f^n)(f x)
proof −
  have f((f^n) x) = (f^(n+1)) x by simp
  also have ... = (f^n o f^1) x by (simp only: funpow-add)
  also have ... = (f^n)(f x) by simp
  finally show ?thesis .
qed

lemma rel-pow-1 [simp]:
  fixes R :: ('a*'a)set
  shows R^1 = R
  by 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)
  by auto

lemma rel-pow-Suc-I2:
    (x, y) : R ⟹ (y, z) : R^n ⟹ (x,z) : R^(Suc n)
  apply (induct n arbitrary: z)
   apply simp
  apply fastsimp
  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 (cases n) auto

lemma rel-pow-Suc-D2:
  (x, z) : R^(Suc n) ⟹ (∃ y. (x,y) : R & (y,z) : R^n)
  apply (induct n arbitrary: x z)
   apply (blast intro: rel-pow-0-I elim: rel-pow-0-E rel-pow-Suc-E)
  apply (blast intro: rel-pow-Suc-I elim: rel-pow-0-E rel-pow-Suc-E)
  done

lemma rel-pow-Suc-D2':
  ∀ x y z. (x,y) : R^n & (y,z) : R --> (∃ w. (x,w) : R & (w,z) : R^n)
  by (induct n) (simp-all, blast)

lemma rel-pow-E2:
  [| (x,z) : R^n;  [| n=0; x = z |] ==> P;
      !!y m. [| n = Suc m; (x,y) : R; (y,z) : R^m |] ==> P
   |] ==> P
  apply (case-tac n, simp)
  apply (cut-tac n=nat and R=R in rel-pow-Suc-D2', simp, blast)
  done

lemma rtrancl-imp-UN-rel-pow: !!p. p:R^* ==> p : (UN n. R^n)
  apply (simp only: split-tupled-all)
  apply (erule rtrancl-induct)
   apply (blast intro: rel-pow-0-I rel-pow-Suc-I)+
  done

lemma rel-pow-imp-rtrancl: !!p. p:R^n ==> p:R^*
  apply (simp only: split-tupled-all)
  apply (induct n)
   apply (blast intro: rtrancl-refl elim: rel-pow-0-E)
  apply (blast elim: rel-pow-Suc-E intro: rtrancl-into-rtrancl)
  done

lemma rtrancl-is-UN-rel-pow: R^* = (UN n. R^n)
  by (blast intro: rtrancl-imp-UN-rel-pow rel-pow-imp-rtrancl)

lemma trancl-power:
  x ∈ r^+ = (∃ n > 0. x ∈ r^n)
  apply (cases x)
  apply simp
  apply (rule iffI)
   apply (drule tranclD2)
   apply (clarsimp simp: rtrancl-is-UN-rel-pow)
   apply (rule-tac x=Suc x in exI)
   apply (clarsimp simp: rel-comp-def)
```

```
  apply fastsimp
 apply clarsimp
 apply (case-tac n, simp)
 apply clarsimp
 apply (drule rel-pow-imp-rtrancl)
 apply fastsimp
 done
```

```
lemma single-valued-rel-pow:
  !!r::('a * 'a)set. single-valued r ==> single-valued (r^n)
 apply (rule single-valuedI)
 apply (induct n)
  apply simp
 apply (fast dest: single-valuedD elim: rel-pow-Suc-E)
 done
```

```
ML
⟪
val funpow-add = thm funpow-add;
val rel-pow-1 = thm rel-pow-1;
val rel-pow-0-I = thm rel-pow-0-I;
val rel-pow-Suc-I = thm rel-pow-Suc-I;
val rel-pow-Suc-I2 = thm rel-pow-Suc-I2;
val rel-pow-0-E = thm rel-pow-0-E;
val rel-pow-Suc-E = thm rel-pow-Suc-E;
val rel-pow-E = thm rel-pow-E;
val rel-pow-Suc-D2 = thm rel-pow-Suc-D2;
val rel-pow-Suc-D2 = thm rel-pow-Suc-D2;
val rel-pow-E2 = thm rel-pow-E2;
val rtrancl-imp-UN-rel-pow = thm rtrancl-imp-UN-rel-pow;
val rel-pow-imp-rtrancl = thm rel-pow-imp-rtrancl;
val rtrancl-is-UN-rel-pow = thm rtrancl-is-UN-rel-pow;
val single-valued-rel-pow = thm single-valued-rel-pow;
⟫
```

end

42 Refute: Refute

```
theory Refute
imports Datatype
uses Tools/prop-logic.ML
     Tools/sat-solver.ML
     Tools/refute.ML
     Tools/refute-isar.ML
begin
```

setup *Refute.setup*

```
(* ------------------------------------------------------------------------- *)
(* REFUTE                                                                    *)
(*                                                                           *)
(* We use a SAT solver to search for a (finite) model that refutes a given   *)
(* HOL formula.                                                              *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* NOTE                                                                      *)
(*                                                                           *)
(* I strongly recommend that you install a stand-alone SAT solver if you     *)
(* want to use 'refute'.  For details see 'HOL/Tools/sat_solver.ML'.  If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME'  *)
(* in 'etc/settings'.                                                        *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* USAGE                                                                     *)
(*                                                                           *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples.  The supported    *)
(* parameters are explained below.                                           *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* CURRENT LIMITATIONS                                                       *)
(*                                                                           *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and           *)
(* inductively defined sets.  Constants are unfolded automatically, and sort *)
(* axioms are added as well.  Other, user-asserted axioms however are        *)
(* ignored.  Inductive datatypes and recursive functions are supported, but  *)
(* may lead to spurious countermodels.                                       *)
(*                                                                           *)
(* The (space) complexity of the algorithm is non-elementary.                *)
(*                                                                           *)
(* Schematic type variables are not supported.                               *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* PARAMETERS                                                                *)
(*                                                                           *)
(* The following global parameters are currently supported (and required):  *)
(*                                                                           *)
(* Name          Type    Description                                         *)
(*                                                                           *)
(* "minsize"     int     Only search for models with size at least           *)
(*                       'minsize'.                                          *)
(* "maxsize"     int     If >0, only search for models with size at most     *)
```

```
(*                        'maxsize'.                                    *)
(* "maxvars"     int     If >0, use at most 'maxvars' boolean variables   *)
(*                        when transforming the term into a propositional *)
(*                        formula.                                        *)
(* "maxtime"     int     If >0, terminate after at most 'maxtime' seconds. *)
(*                        This value is ignored under some ML compilers.  *)
(* "satsolver"   string  Name of the SAT solver to be used.              *)
(*                                                                        *)
(* See 'HOL/SAT.thy' for default values.                                  *)
(*                                                                        *)
(* The size of particular types can be specified in the form type=size    *)
(* (where 'type' is a string, and 'size' is an int).  Examples:           *)
(* "'a"=1                                                                 *)
(* "List.list"=2                                                          *)
(* ---------------------------------------------------------------------- *)

(* ---------------------------------------------------------------------- *)
(* FILES                                                                  *)
(*                                                                        *)
(* HOL/Tools/prop_logic.ML    Propositional logic                         *)
(* HOL/Tools/sat_solver.ML    SAT solvers                                 *)
(* HOL/Tools/refute.ML        Translation HOL -> propositional logic and  *)
(*                            Boolean assignment -> HOL model             *)
(* HOL/Tools/refute_isar.ML   Adds 'refute'/'refute_params' to Isabelle's *)
(*                            syntax                                      *)
(* HOL/Refute.thy             This file: loads the ML files, basic setup, *)
(*                            documentation                               *)
(* HOL/SAT.thy                Sets default parameters                     *)
(* HOL/ex/RefuteExamples.thy  Examples                                    *)
(* ---------------------------------------------------------------------- *)
```

end

43 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

44 Recdef: TFL: recursive function definitions

theory *Recdef*
imports *Wellfounded-Relations FunDef*
uses
(*Tools*/*TFL*/*casesplit.ML*)
(*Tools*/*TFL*/*utils.ML*)
(*Tools*/*TFL*/*usyntax.ML*)
(*Tools*/*TFL*/*dcterm.ML*)
(*Tools*/*TFL*/*thms.ML*)
(*Tools*/*TFL*/*rules.ML*)
(*Tools*/*TFL*/*thry.ML*)
(*Tools*/*TFL*/*tfl.ML*)
(*Tools*/*TFL*/*post.ML*)
(*Tools*/*recdef-package.ML*)
begin

lemma *tfl-eq-True*: (*x* = *True*) −−> *x*
by *blast*

lemma *tfl-rev-eq-mp*: (*x* = *y*) −−> *y* −−> *x*
by *blast*

lemma *tfl-simp-thm*: (*x* −−> *y*) −−> (*x* = *x′*) −−> (*x′* −−> *y*)
by *blast*

lemma *tfl-P-imp-P-iff-True*: *P* ==> *P* = *True*
by *blast*

lemma *tfl-imp-trans*: $(A \dashrightarrow B) \Longrightarrow (B \dashrightarrow C) \Longrightarrow (A \dashrightarrow C)$
by *blast*

lemma *tfl-disj-assoc*: $(a \lor b) \lor c == a \lor (b \lor c)$
by *simp*

lemma *tfl-disjE*: $P \lor Q ==> P --> R ==> Q --> R ==> R$
by *blast*

lemma *tfl-exE*: $\exists x.\ P\ x ==> \forall x.\ P\ x --> Q ==> Q$
by *blast*

use *Tools/TFL/casesplit.ML*
use *Tools/TFL/utils.ML*
use *Tools/TFL/usyntax.ML*
use *Tools/TFL/dcterm.ML*
use *Tools/TFL/thms.ML*
use *Tools/TFL/rules.ML*
use *Tools/TFL/thry.ML*
use *Tools/TFL/tfl.ML*
use *Tools/TFL/post.ML*
use *Tools/recdef-package.ML*
setup *RecdefPackage.setup*

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def
less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong let-cong image-cong INT-cong UN-cong bex-cong ball-cong imp-cong

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-pred-nat
wf-same-fst
wf-empty

end

45 Extraction: Program extraction for HOL

theory *Extraction*
imports *Datatype*
uses *Tools/rewrite-hol-proof.ML*
begin

45.1 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*] =
  meta-spec atomize-eq atomize-all atomize-imp atomize-conj
  allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
  notE′ impE′ impE iffE imp-cong simp-thms eq-True eq-False
  induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq

induct-forall-def induct-implies-def induct-equal-def induct-conj-def induct-atomize induct-rulify induct-rulify-fallback True-implies-equals TrueE

datatype *sumbool* = *Left* | *Right*

45.2 Type of extracted program

extract-type

$typeof\ (Trueprop\ P) \equiv typeof\ P$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('Q))$

$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE(Null))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('P \Rightarrow 'Q))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow typeof\ (\forall x.\ P\ x) \equiv Type\ (TYPE(Null))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow typeof\ (\forall x::'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow typeof\ (\exists x::'a.\ P\ x) \equiv Type\ (TYPE('a))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow typeof\ (\exists x::'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

45.3 Realizability

realizability

$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \vee Q)) \equiv$
$\quad (case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \vee Q)) \equiv$
$\quad (case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \vee Q)) \equiv$
$\quad (case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad realizes\ t\ (\neg\ P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$\quad realizes\ t\ (\neg\ P) \equiv (\forall\, x{::}'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P{::}bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\ typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

45.4 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: $case\ x\ of\ None \Rightarrow P \mid Some\ q \Rightarrow Q\ q$
  and $r1$: $P \Longrightarrow R\ f$ **and** $r2$: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
  shows $R\ (case\ x\ of\ None \Rightarrow f \mid Some\ q \Rightarrow g\ q)$
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: $case\ x\ of\ Left \Rightarrow P \mid Right \Rightarrow Q$
  and $r1$: $P \Longrightarrow R\ f$ **and** $r2$: $Q \Longrightarrow R\ g$
  shows $R\ (case\ x\ of\ Left \Rightarrow f \mid Right \Rightarrow g)$
proof (*cases x*)

case *Left*
with r **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Right*
with r **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *conjI-realizer*:
$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (fst\ (p,\ q)) \wedge Q\ (snd\ (p,\ q))$
by *simp*

theorem *exI-realizer*:
$P\ y\ x \Longrightarrow P\ (snd\ (x,\ y))\ (fst\ (x,\ y))$ **by** *simp*

theorem *exE-realizer*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q\ (f\ x\ y)) \Longrightarrow Q\ (let\ (x,\ y) = p\ in\ f\ x\ y)$
by (*cases p*) (*simp add*: *Let-def*)

theorem *exE-realizer'*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q$ **by** (*cases p*) *simp*

realizers
$impI\ (P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ P\ Q\ pq\ (h{:}\ \text{-}).\ allI \cdot \text{-} \bullet (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \bullet (h \cdot x))$

$impI\ (P)$: *Null*
$\Lambda\ P\ Q\ (h{:}\ \text{-}).\ allI \cdot \text{-} \bullet (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \bullet (h \cdot x))$

$impI\ (Q)$: $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ impI \cdot \text{-} \cdot \text{-}$

impI: *Null impI*

$mp\ (P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ P\ Q\ pq\ (h{:}\ \text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \bullet (spec \cdot \text{-} \cdot p \bullet h)$

$mp\ (P)$: *Null*
$\Lambda\ P\ Q\ (h{:}\ \text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \bullet (spec \cdot \text{-} \cdot p \bullet h)$

$mp\ (Q)$: $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ mp \cdot \text{-} \cdot \text{-}$

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*

$\mathit{exI}\ (P)\colon \lambda x\ p.\ (x,\ p)\ \Lambda\ P\ x\ p.\ \mathit{exI\text{-}realizer} \cdot P \cdot p \cdot x$

$\mathit{exI}\colon \lambda x.\ x\ \Lambda\ P\ x\ (h\colon \text{-}).\ h$

$\mathit{exE}\ (P,\ Q)\colon \lambda p\ pq.\ \mathit{let}\ (x,\ y) = p\ \mathit{in}\ pq\ x\ y$
$\Lambda\ P\ Q\ p\ (h\colon \text{-})\ pq.\ \mathit{exE\text{-}realizer} \cdot P \cdot p \cdot Q \cdot pq \bullet h$

$\mathit{exE}\ (P)\colon \mathit{Null}$
$\Lambda\ P\ Q\ p.\ \mathit{exE\text{-}realizer}' \cdot \text{-} \cdot \text{-} \cdot \text{-}$

$\mathit{exE}\ (Q)\colon \lambda x\ pq.\ pq\ x$
$\Lambda\ P\ Q\ x\ (h1\colon \text{-})\ pq\ (h2\colon \text{-}).\ h2 \cdot x \bullet h1$

$\mathit{exE}\colon \mathit{Null}$
$\Lambda\ P\ Q\ x\ (h1\colon \text{-})\ (h2\colon \text{-}).\ h2 \cdot x \bullet h1$

$\mathit{conjI}\ (P,\ Q)\colon \mathit{Pair}$
$\Lambda\ P\ Q\ p\ (h\colon \text{-})\ q.\ \mathit{conjI\text{-}realizer} \cdot P \cdot p \cdot Q \cdot q \bullet h$

$\mathit{conjI}\ (P)\colon \lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjI} \cdot \text{-} \cdot \text{-}$

$\mathit{conjI}\ (Q)\colon \lambda q.\ q$
$\Lambda\ P\ Q\ (h\colon \text{-})\ q.\ \mathit{conjI} \cdot \text{-} \cdot \text{-} \bullet h$

$\mathit{conjI}\colon \mathit{Null}\ \mathit{conjI}$

$\mathit{conjunct1}\ (P,\ Q)\colon \mathit{fst}$
$\Lambda\ P\ Q\ pq.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-}$

$\mathit{conjunct1}\ (P)\colon \lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-}$

$\mathit{conjunct1}\ (Q)\colon \mathit{Null}$
$\Lambda\ P\ Q\ q.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-}$

$\mathit{conjunct1}\colon \mathit{Null}\ \mathit{conjunct1}$

$\mathit{conjunct2}\ (P,\ Q)\colon \mathit{snd}$
$\Lambda\ P\ Q\ pq.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-}$

$\mathit{conjunct2}\ (P)\colon \mathit{Null}$
$\Lambda\ P\ Q\ p.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-}$

$\mathit{conjunct2}\ (Q)\colon \lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-}$

$\mathit{conjunct2}\colon \mathit{Null}\ \mathit{conjunct2}$

disjI1 $(P,\ Q)$: *Inl*
$\Lambda\ P\ Q\ p.\ \mathit{iffD2} \cdot \cdot \cdot \cdot \bullet (\mathit{sum.cases\text{-}1} \cdot P \cdot \cdot \cdot p)$

disjI1 (P): *Some*
$\Lambda\ P\ Q\ p.\ \mathit{iffD2} \cdot \cdot \cdot \cdot \bullet (\mathit{option.cases\text{-}2} \cdot \cdot \cdot P \cdot p)$

disjI1 (Q): *None*
$\Lambda\ P\ Q.\ \mathit{iffD2} \cdot \cdot \cdot \cdot \bullet (\mathit{option.cases\text{-}1} \cdot \cdot \cdot \text{-})$

disjI1: *Left*
$\Lambda\ P\ Q.\ \mathit{iffD2} \cdot \cdot \cdot \cdot \bullet (\mathit{sumbool.cases\text{-}1} \cdot \cdot \cdot \text{-})$

disjI2 $(P,\ Q)$: *Inr*
$\Lambda\ Q\ P\ q.\ \mathit{iffD2} \cdot \cdot \cdot \cdot \bullet (\mathit{sum.cases\text{-}2} \cdot \cdot \cdot Q \cdot q)$

disjI2 (P): *None*
$\Lambda\ Q\ P.\ \mathit{iffD2} \cdot \cdot \cdot \cdot \bullet (\mathit{option.cases\text{-}1} \cdot \cdot \cdot \text{-})$

disjI2 (Q): *Some*
$\Lambda\ Q\ P\ q.\ \mathit{iffD2} \cdot \cdot \cdot \cdot \bullet (\mathit{option.cases\text{-}2} \cdot \cdot \cdot Q \cdot q)$

disjI2: *Right*
$\Lambda\ Q\ P.\ \mathit{iffD2} \cdot \cdot \cdot \cdot \bullet (\mathit{sumbool.cases\text{-}2} \cdot \cdot \cdot \text{-})$

disjE $(P,\ Q,\ R)$: $\lambda pq\ pr\ qr.$
$(\mathit{case}\ pq\ \mathit{of}\ \mathit{Inl}\ p \Rightarrow pr\ p \mid \mathit{Inr}\ q \Rightarrow qr\ q)$
$\Lambda\ P\ Q\ R\ pq\ (h1\text{: -})\ pr\ (h2\text{: -})\ qr.$
$\mathit{disjE\text{-}realizer} \cdot \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \bullet h1 \bullet h2$

disjE $(Q,\ R)$: $\lambda pq\ pr\ qr.$
$(\mathit{case}\ pq\ \mathit{of}\ \mathit{None} \Rightarrow pr \mid \mathit{Some}\ q \Rightarrow qr\ q)$
$\Lambda\ P\ Q\ R\ pq\ (h1\text{: -})\ pr\ (h2\text{: -})\ qr.$
$\mathit{disjE\text{-}realizer2} \cdot \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \bullet h1 \bullet h2$

disjE $(P,\ R)$: $\lambda pq\ pr\ qr.$
$(\mathit{case}\ pq\ \mathit{of}\ \mathit{None} \Rightarrow qr \mid \mathit{Some}\ p \Rightarrow pr\ p)$
$\Lambda\ P\ Q\ R\ pq\ (h1\text{: -})\ pr\ (h2\text{: -})\ qr\ (h3\text{: -}).$
$\mathit{disjE\text{-}realizer2} \cdot \cdot \cdot \cdot \cdot pq \cdot R \cdot qr \cdot pr \bullet h1 \bullet h3 \bullet h2$

disjE (R): $\lambda pq\ pr\ qr.$
$(\mathit{case}\ pq\ \mathit{of}\ \mathit{Left} \Rightarrow pr \mid \mathit{Right} \Rightarrow qr)$
$\Lambda\ P\ Q\ R\ pq\ (h1\text{: -})\ pr\ (h2\text{: -})\ qr.$
$\mathit{disjE\text{-}realizer3} \cdot \cdot \cdot \cdot \cdot pq \cdot R \cdot pr \cdot qr \bullet h1 \bullet h2$

disjE $(P,\ Q)$: *Null*
$\Lambda\ P\ Q\ R\ pq.\ \mathit{disjE\text{-}realizer} \cdot \cdot \cdot \cdot \cdot pq \cdot (\lambda x.\ R) \cdot \cdot \cdot \text{-}$

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*
Λ *Q P pq* (*h*: -) *p*.
mp · - · - • (*spec* · - · *p* • (*conjunct1* · - · - • *h*))

iffD1 (*P*): λ*p. p*
Λ *Q P p* (*h*: -). *mp* · - · - • (*conjunct1* · - · - • *h*)

iffD1 (*Q*): *Null*
Λ *Q P q1* (*h*: -) *q2*.
mp · - · - • (*spec* · - · *q2* • (*conjunct1* · - · - • *h*))

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*

Λ *P Q pq* (*h*: -) *q*.
  mp · - - - • (*spec* · - · *q* • (*conjunct2* · - - - • *h*))

iffD2 (*P*): $\lambda p.\ p$
Λ *P Q p* (*h*: -). *mp* · - - - • (*conjunct2* · - - - • *h*)

iffD2 (*Q*): *Null*
Λ *P Q q1* (*h*: -) *q2*.
  mp · - - - • (*spec* · - · *q2* • (*conjunct2* · - - - • *h*))

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
Λ *P Q pq* (*h1* : -) *qp* (*h2* : -). *conjI-realizer* ·
  ($\lambda pq.\ \forall x.\ P\ x \longrightarrow Q\ (pq\ x)$) · *pq* ·
  ($\lambda qp.\ \forall x.\ Q\ x \longrightarrow P\ (qp\ x)$) · *qp* •
  (*allI* · - • (Λ *x*. *impI* · - - - • (*h1* · *x*))) •
  (*allI* · - • (Λ *x*. *impI* · - - - • (*h2* · *x*)))

iffI (*P*): $\lambda p.\ p$
Λ *P Q* (*h1* : -) *p* (*h2* : -). *conjI* · - - - •
  (*allI* · - • (Λ *x*. *impI* · - - - • (*h1* · *x*))) •
  (*impI* · - - - • *h2*)

iffI (*Q*): $\lambda q.\ q$
Λ *P Q q* (*h1* : -) (*h2* : -). *conjI* · - - - •
  (*impI* · - - - • *h1*) •
  (*allI* · - • (Λ *x*. *impI* · - - - • (*h2* · *x*)))

iffI: *Null iffI*

end

46 ATP-Linkup: The Isabelle-ATP Linkup

theory *ATP-Linkup*
imports *Divides Record Hilbert-Choice Presburger Relation-Power SAT Recdef Extraction*

uses
Tools/polyhash.ML
Tools/res-clause.ML
(*Tools/res-hol-clause.ML*)
(*Tools/res-axioms.ML*)
(*Tools/res-reconstruct.ML*)
(*Tools/watcher.ML*)

(*Tools/res-atp.ML*)
(*Tools/res-atp-provers.ML*)
(*Tools/res-atp-methods.ML*)
~~/*src/Tools/Metis/metis.ML*
(*Tools/metis-tools.ML*)
begin

definition *COMBI* :: *'a* => *'a*
where *COMBI P* == *P*

definition *COMBK* :: *'a* => *'b* => *'a*
where *COMBK P Q* == *P*

definition *COMBB* :: (*'b* => *'c*) => (*'a* => *'b*) => *'a* => *'c*
where *COMBB P Q R* == *P* (*Q R*)

definition *COMBC* :: (*'a* => *'b* => *'c*) => *'b* => *'a* => *'c*
where *COMBC P Q R* == *P R Q*

definition *COMBS* :: (*'a* => *'b* => *'c*) => (*'a* => *'b*) => *'a* => *'c*
where *COMBS P Q R* == *P R* (*Q R*)

definition *fequal* :: *'a* => *'a* => *bool*
where *fequal X Y* == (*X*=*Y*)

lemma *fequal-imp-equal*: *fequal X Y* ==> *X*=*Y*
by (*simp add*: *fequal-def*)

lemma *equal-imp-fequal*: *X*=*Y* ==> *fequal X Y*
by (*simp add*: *fequal-def*)

These two represent the equivalence between Boolean equality and iff. They can't be converted to clauses automatically, as the iff would be expanded...

lemma *iff-positive*: *P* | *Q* | *P*=*Q*
by *blast*

lemma *iff-negative*: ~*P* | ~*Q* | *P*=*Q*
by *blast*

Theorems for translation to combinators

lemma *abs-S*: (%*x*. (*f x*) (*g x*)) == *COMBS f g*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBS-def*)
done

lemma *abs-I*: (%*x*. *x*) == *COMBI*
apply (*rule eq-reflection*)
apply (*rule ext*)

apply (*simp add*: *COMBI-def*)
done

lemma *abs-K*: (%*x*. *y*) == *COMBK y*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBK-def*)
done

lemma *abs-B*: (%*x*. *a* (*g x*)) == *COMBB a g*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBB-def*)
done

lemma *abs-C*: (%*x*. (*f x*) *b*) == *COMBC f b*
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBC-def*)
done

use *Tools/res-axioms.ML* — requires the combinators declared above
use *Tools/res-hol-clause.ML*
use *Tools/res-reconstruct.ML*
use *Tools/watcher.ML*
use *Tools/res-atp.ML*

setup *ResAxioms.meson-method-setup*

46.1 Setup for Vampire, E prover and SPASS

use *Tools/res-atp-provers.ML*

oracle *vampire-oracle* (*string* ∗ *int*) = ⟪ *ResAtpProvers.vampire-o* ⟫
oracle *eprover-oracle* (*string* ∗ *int*) = ⟪ *ResAtpProvers.eprover-o* ⟫
oracle *spass-oracle* (*string* ∗ *int*) = ⟪ *ResAtpProvers.spass-o* ⟫

use *Tools/res-atp-methods.ML*
setup *ResAtpMethods.setup* — Oracle ATP methods: still useful?
setup *ResAxioms.setup* — Sledgehammer

46.2 The Metis prover

use *Tools/metis-tools.ML*
setup *MetisTools.setup*

setup ⟪
  Theory.at-end ResAxioms.clause-cache-endtheory
⟫

end

47 PreList: A Basis for Building the Theory of Lists

theory *PreList*
imports *ATP-Linkup*
uses *Tools/function-package/lexicographic-order.ML*
Tools/function-package/fundef-datatype.ML
begin

This is defined separately to serve as a basis for theory ToyList in the documentation.

setup *LexicographicOrder.setup*
setup *FundefDatatype.setup*

end

48 List: The datatype of finite lists

theory *List*
imports *PreList*
uses *Tools/string-syntax.ML*
begin

datatype *'a list* =
Nil ([])
| *Cons 'a 'a list* (**infixr** # *65*)

48.1 Basic list processing functions

consts
filter:: (*'a* => *bool*) => *'a list* => *'a list*
concat:: *'a list list* => *'a list*
foldl :: (*'b* => *'a* => *'b*) => *'b* => *'a list* => *'b*
foldr :: (*'a* => *'b* => *'b*) => *'a list* => *'b* => *'b*
hd:: *'a list* => *'a*
tl:: *'a list* => *'a list*
last:: *'a list* => *'a*
butlast :: *'a list* => *'a list*
set :: *'a list* => *'a set*
map :: (*'a*=>*'b*) => (*'a list* => *'b list*)
listsum :: *'a list* => *'a*::*monoid-add*
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*
distinct:: *'a list => bool*
replicate :: *nat => 'a => 'a list*
splice :: *'a list* ⇒ *'a list* ⇒ *'a list*

nonterminals *lupdbinds lupdbind*

syntax
— list Enumeration
@*list* :: *args => 'a list* ([(-)])

— Special syntax for filter
@*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list* ((*1*[-<−-./ -]))

— list update
-*lupdbind*:: [*'a*, *'a*] => *lupdbind* ((*2*- :=/ -))
:: *lupdbind => lupdbinds* (-)
-*lupdbinds* :: [*lupdbind*, *lupdbinds*] => *lupdbinds* (-,/ -)
-*LUpdate* :: [*'a*, *lupdbinds*] => *'a* (-/[(-)] [*900*,*0*] *900*)

translations
[*x*, *xs*] == *x*#[*xs*]
[*x*] == *x*#[]
[*x*<−*xs* . *P*]== *filter* (%*x*. *P*) *xs*

-*LUpdate xs* (-*lupdbinds b bs*)== -*LUpdate* (-*LUpdate xs b*) *bs*
xs[*i*:=*x*] == *list-update xs i x*

syntax (*xsymbols*)
@*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list*((*1*[-←- ./ -]))
syntax (*HTML* **output**)
@*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list*((*1*[-←- ./ -]))

Function *size* is overloaded for all datatypes. Users may refer to the list version as *length*.

abbreviation
length :: *'a list => nat* **where**
length == *size*

primrec
$hd(x\#xs) = x$

primrec
$tl([]) = []$
$tl(x\#xs) = xs$

primrec
$last(x\#xs) = (\mathit{if}\ xs{=}[]\ \mathit{then}\ x\ \mathit{else}\ last\ xs)$

primrec
$butlast\ [] = []$
$butlast(x\#xs) = (\mathit{if}\ xs{=}[]\ \mathit{then}\ []\ \mathit{else}\ x\#butlast\ xs)$

primrec
$set\ [] = \{\}$
$set\ (x\#xs) = insert\ x\ (set\ xs)$

primrec
$map\ f\ [] = []$
$map\ f\ (x\#xs) = f(x)\#map\ f\ xs$

setup $\langle\!\langle$ *snd o Sign.declare-const* [] (∗*authentic syntax*∗)
(*append*, @{*typ* *'a list ⇒ 'a list ⇒ 'a list*}, *InfixrName* (@, *65*)) $\rangle\!\rangle$
primrec
append-Nil:$[]@ys = ys$
append-Cons: $(x\#xs)@ys = x\#(xs@ys)$

primrec
$rev([]) = []$
$rev(x\#xs) = rev(xs)\ @\ [x]$

primrec
$filter\ P\ [] = []$
$filter\ P\ (x\#xs) = (\mathit{if}\ P\ x\ \mathit{then}\ x\#filter\ P\ xs\ \mathit{else}\ filter\ P\ xs)$

primrec
foldl-Nil:$foldl\ f\ a\ [] = a$
foldl-Cons: $foldl\ f\ a\ (x\#xs) = foldl\ f\ (f\ a\ x)\ xs$

primrec
$foldr\ f\ []\ a = a$
$foldr\ f\ (x\#xs)\ a = f\ x\ (foldr\ f\ xs\ a)$

primrec
$concat([]) = []$
$concat(x\#xs) = x\ @\ concat(xs)$

primrec
listsum [] = *0*
listsum (*x* # *xs*) = *x* + *listsum xs*

primrec
drop-Nil:*drop n* [] = []
drop-Cons: *drop n* (*x*#*xs*) = (*case n of 0* => *x*#*xs* | *Suc*(*m*) => *drop m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
take-Nil:*take n* [] = []
take-Cons: *take n* (*x*#*xs*) = (*case n of 0* => [] | *Suc*(*m*) => *x* # *take m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
nth-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
$replicate\text{-}0$: $replicate\ 0\ x = []$
$replicate\text{-}Suc$: $replicate\ (Suc\ n)\ x = x\ \#\ replicate\ n\ x$

definition
$rotate1 :: {'a}\ list \Rightarrow {'a}\ list$ **where**
$rotate1\ xs = (case\ xs\ of\ [] \Rightarrow []\ |\ x\#xs \Rightarrow xs\ @\ [x])$

definition
$rotate :: nat \Rightarrow {'a}\ list \Rightarrow {'a}\ list$ **where**
$rotate\ n = rotate1\ \hat{}\ n$

definition
$list\text{-}all2 :: ({'a} => {'b} => bool) => {'a}\ list => {'b}\ list => bool$ **where**
$list\text{-}all2\ P\ xs\ ys =$
$(length\ xs = length\ ys \wedge (\forall (x, y) \in set\ (zip\ xs\ ys).\ P\ x\ y))$

definition
$sublist :: {'a}\ list => nat\ set => {'a}\ list$ **where**
$sublist\ xs\ A = map\ fst\ (filter\ (\lambda p.\ snd\ p \in A)\ (zip\ xs\ [0..{<}size\ xs]))$

primrec
$splice\ []\ ys = ys$
$splice\ (x\#xs)\ ys = (if\ ys{=}[]\ then\ x\#xs\ else\ x\ \#\ hd\ ys\ \#\ splice\ xs\ (tl\ ys))$
— Warning: simpset does not contain the second eqn but a derived one.

The following simple sort functions are intended for proofs, not for efficient implementations.

context *linorder*
begin

fun $sorted :: {'a}\ list \Rightarrow bool$ **where**
$sorted\ [] \longleftrightarrow True\ |$
$sorted\ [x] \longleftrightarrow True\ |$
$sorted\ (x\#y\#zs) \longleftrightarrow x <= y \wedge sorted\ (y\#zs)$

fun $insort :: {'a} \Rightarrow {'a}\ list \Rightarrow {'a}\ list$ **where**
$insort\ x\ [] = [x]\ |$
$insort\ x\ (y\#ys) = (if\ x <= y\ then\ (x\#y\#ys)\ else\ y\#(insort\ x\ ys))$

fun $sort :: {'a}\ list \Rightarrow {'a}\ list$ **where**
$sort\ [] = []\ |$
$sort\ (x\#xs) = insort\ x\ (sort\ xs)$

end

48.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: $[(x,y).\ x \leftarrow xs,\ y \leftarrow ys,\ x \neq y]$, the list of all pairs of distinct elements from xs and ys. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle's set comprehension): $[e.\ x \leftarrow xs, \ldots]$ rather than `[e| x <- xs, ...]`.

The qualifiers after the dot are

generators $p \leftarrow xs$, where p is a pattern and xs an expression of list type, or

guards b, where b is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of $[e.\ x \leftarrow xs]$ is optmized to *map* $(\lambda x.\ e)\ xs$.

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

nonterminals *lc-qual lc-quals*

syntax
-listcompr :: *'a* ⇒ *lc-qual* ⇒ *lc-quals* ⇒ *'a list* ([- . --)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- <− -)
-lc-test :: *bool* ⇒ *lc-qual* (-)

-lc-end :: *lc-quals* (])
-lc-quals :: *lc-qual* ⇒ *lc-quals* ⇒ *lc-quals* (, --)
-lc-abs :: *'a* => *'b list* => *'b list*

syntax (*xsymbols*)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- ← -)
syntax (*HTML* **output**)
-lc-gen :: *'a* ⇒ *'a list* ⇒ *lc-qual* (- ← -)

parse-translation (**advanced**) ⟪
let
  val NilC = *Syntax.const* @{*const-name Nil*};
  val ConsC = *Syntax.const* @{*const-name Cons*};

```
  val mapC = Syntax.const @{const-name map};
  val concatC = Syntax.const @{const-name concat};
  val IfC = Syntax.const @{const-name If};
  fun singl x = ConsC $ x $ NilC;

   fun pat-tr ctxt p e opti = (* %x. case x of p => e | - => [] *)
    let
      val x = Free (Name.variant (add-term-free-names (p$e, [])) x, dummyT);
      val e = if opti then singl e else e;
      val case1 = Syntax.const -case1 $ p $ e;
      val case2 = Syntax.const -case1 $ Syntax.const Term.dummy-patternN
                                    $ NilC;
      val cs = Syntax.const -case2 $ case1 $ case2
      val ft = DatatypeCase.case-tr false DatatypePackage.datatype-of-constr
                 ctxt [x, cs]
    in lambda x ft end;

  fun abs-tr ctxt (p as Free(s,T)) e opti =
        let val thy = ProofContext.theory-of ctxt;
            val s' = Sign.intern-const thy s
        in if Sign.declared-const thy s'
           then (pat-tr ctxt p e opti, false)
           else (lambda p e, true)
        end
    | abs-tr ctxt p e opti = (pat-tr ctxt p e opti, false);

  fun lc-tr ctxt [e, Const(-lc-test,-)$b, qs] =
        let val res = case qs of Const(-lc-end,-) => singl e
                      | Const(-lc-quals,-)$q$qs => lc-tr ctxt [e,q,qs];
        in IfC $ b $ res $ NilC end
    | lc-tr ctxt [e, Const(-lc-gen,-) $ p $ es, Const(-lc-end,-)] =
        (case abs-tr ctxt p e true of
           (f,true) => mapC $ f $ es
         | (f, false) => concatC $ (mapC $ f $ es))
    | lc-tr ctxt [e, Const(-lc-gen,-) $ p $ es, Const(-lc-quals,-)$q$qs] =
        let val e' = lc-tr ctxt [e,q,qs];
        in concatC $ (mapC $ (fst(abs-tr ctxt p e' false)) $ es) end

in [(-listcompr, lc-tr)] end
⟫
```

48.1.2 [] and *op*

lemma *not-Cons-self* [*simp*]:
  $xs \neq x \# xs$
by (*induct xs*) *auto*

lemmas *not-Cons-self2* [*simp*] = *not-Cons-self* [*symmetric*]

lemma *neq-Nil-conv*: $(xs \neq []) = (\exists\, y\ ys.\ xs = y\ \#\ ys)$
by (*induct xs*) *auto*

lemma *length-induct*:
$(\bigwedge xs.\ \forall\, ys.\ length\ ys < length\ xs \longrightarrow P\ ys \Longrightarrow P\ xs) \Longrightarrow P\ xs$
by (*rule measure-induct* [*of length*]) *iprover*

48.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: $length\ (xs\ @\ ys) = length\ xs + length\ ys$
by (*induct xs*) *auto*

lemma *length-map* [*simp*]: $length\ (map\ f\ xs) = length\ xs$
by (*induct xs*) *auto*

lemma *length-rev* [*simp*]: $length\ (rev\ xs) = length\ xs$
by (*induct xs*) *auto*

lemma *length-tl* [*simp*]: $length\ (tl\ xs) = length\ xs - 1$
by (*cases xs*) *auto*

lemma *length-0-conv* [*iff*]: $(length\ xs = 0) = (xs = [])$
by (*induct xs*) *auto*

lemma *length-greater-0-conv* [*iff*]: $(0 < length\ xs) = (xs \neq [])$
by (*induct xs*) *auto*

lemma *length-pos-if-in-set*: $x : set\ xs \Longrightarrow length\ xs > 0$
by *auto*

lemma *length-Suc-conv*:
$(length\ xs = Suc\ n) = (\exists\, y\ ys.\ xs = y\ \#\ ys \wedge length\ ys = n)$
by (*induct xs*) *auto*

lemma *Suc-length-conv*:
$(Suc\ n = length\ xs) = (\exists\, y\ ys.\ xs = y\ \#\ ys \wedge length\ ys = n)$
apply (*induct xs*, *simp*, *simp*)
apply *blast*
done

lemma *impossible-Cons*: $length\ xs <= length\ ys ==> xs = x\ \#\ ys = False$
by (*induct xs*) *auto*

lemma *list-induct2* [*consumes 1*]:
$[\![\ length\ xs = length\ ys;$
$P\ []\ [];$
$\bigwedge x\ xs\ y\ ys.\ [\![\ length\ xs = length\ ys;\ P\ xs\ ys\ ]\!] \Longrightarrow P\ (x\#xs)\ (y\#ys)\ ]\!]$
$\Longrightarrow P\ xs\ ys$

```
apply(induct xs arbitrary: ys)
 apply simp
apply(case-tac ys)
 apply simp
apply simp
done

lemma list-induct2':
  ⟦ P [] [];
  ⋀x xs. P (x#xs) [];
  ⋀y ys. P [] (y#ys);
   ⋀x xs y ys. P xs ys  ⟹ P (x#xs) (y#ys) ⟧
 ⟹ P xs ys
by (induct xs arbitrary: ys) (case-tac x, auto)+

lemma neq-if-length-neq: length xs ≠ length ys ⟹ (xs = ys) == False
by (rule Eq-FalseI) auto

simproc-setup list-neq ((xs::'a list) = ys) = ⟨⟨
(*
Reduces xs=ys to False if xs and ys cannot be of the same length.
This is the case if the atomic sublists of one are a submultiset
of those of the other list and there are fewer Cons's in one than the other.
*)

let

fun len (Const(List.list.Nil,-)) acc = acc
  | len (Const(List.list.Cons,-) $ - $ xs) (ts,n) = len xs (ts,n+1)
  | len (Const(List.append,-) $ xs $ ys) acc = len xs (len ys acc)
  | len (Const(List.rev,-) $ xs) acc = len xs acc
  | len (Const(List.map,-) $ - $ xs) acc = len xs acc
  | len t (ts,n) = (t::ts,n);

fun list-neq - ss ct =
  let
    val (Const(-,eqT) $ lhs $ rhs) = Thm.term-of ct;
    val (ls,m) = len lhs ([],0) and (rs,n) = len rhs ([],0);
    fun prove-neq() =
      let
        val Type(-,listT::-) = eqT;
        val size = HOLogic.size-const listT;
        val eq-len = HOLogic.mk-eq (size $ lhs, size $ rhs);
        val neq-len = HOLogic.mk-Trueprop (HOLogic.Not $ eq-len);
        val thm = Goal.prove (Simplifier.the-context ss) [] [] neq-len
          (K (simp-tac (Simplifier.inherit-context ss @{simpset}) 1));
      in SOME (thm RS @{thm neq-if-length-neq}) end
  in
    if m < n andalso submultiset (op aconv) (ls,rs) orelse
```

```
      n < m andalso submultiset (op aconv) (rs,ls)
    then prove-neq() else NONE
  end;
in list-neq end;
⟫
```

48.1.4 @ – append

lemma *append-assoc* [*simp*]: (*xs* @ *ys*) @ *zs* = *xs* @ (*ys* @ *zs*)
by (*induct xs*) *auto*

lemma *append-Nil2* [*simp*]: *xs* @ [] = *xs*
by (*induct xs*) *auto*

interpretation *semigroup-append*: *semigroup-add* [*op* @]
by *unfold-locales simp*
interpretation *monoid-append*: *monoid-add* [[] *op* @]
by *unfold-locales* (*simp*+)

lemma *append-is-Nil-conv* [*iff*]: (*xs* @ *ys* = []) = (*xs* = [] ∧ *ys* = [])
by (*induct xs*) *auto*

lemma *Nil-is-append-conv* [*iff*]: ([] = *xs* @ *ys*) = (*xs* = [] ∧ *ys* = [])
by (*induct xs*) *auto*

lemma *append-self-conv* [*iff*]: (*xs* @ *ys* = *xs*) = (*ys* = [])
by (*induct xs*) *auto*

lemma *self-append-conv* [*iff*]: (*xs* = *xs* @ *ys*) = (*ys* = [])
by (*induct xs*) *auto*

lemma *append-eq-append-conv* [*simp*, *noatp*]:
length xs = *length ys* ∨ *length us* = *length vs*
==> (*xs*@*us* = *ys*@*vs*) = (*xs*=*ys* ∧ *us*=*vs*)
apply (*induct xs arbitrary*: *ys*)
apply (*case-tac ys*, *simp*, *force*)
apply (*case-tac ys*, *force*, *simp*)
done

lemma *append-eq-append-conv2*: (*xs* @ *ys* = *zs* @ *ts*) =
(*EX us*. *xs* = *zs* @ *us* & *us* @ *ys* = *ts* | *xs* @ *us* = *zs* & *ys* = *us*@ *ts*)
apply (*induct xs arbitrary*: *ys zs ts*)
apply *fastsimp*
apply(*case-tac zs*)
apply *simp*
apply *fastsimp*
done

lemma *same-append-eq* [*iff*]: (*xs* @ *ys* = *xs* @ *zs*) = (*ys* = *zs*)

by *simp*

lemma *append1-eq-conv* [*iff*]: $(xs \ @ \ [x] = ys \ @ \ [y]) = (xs = ys \wedge x = y)$
by *simp*

lemma *append-same-eq* [*iff*]: $(ys \ @ \ xs = zs \ @ \ xs) = (ys = zs)$
by *simp*

lemma *append-self-conv2* [*iff*]: $(xs \ @ \ ys = ys) = (xs = [])$
using *append-same-eq* [*of* - - []] **by** *auto*

lemma *self-append-conv2* [*iff*]: $(ys = xs \ @ \ ys) = (xs = [])$
using *append-same-eq* [*of* []] **by** *auto*

lemma *hd-Cons-tl* [*simp*,*noatp*]: $xs \neq [] ==> hd \ xs \ \# \ tl \ xs = xs$
by (*induct xs*) *auto*

lemma *hd-append*: $hd \ (xs \ @ \ ys) = (if \ xs = [] \ then \ hd \ ys \ else \ hd \ xs)$
by (*induct xs*) *auto*

lemma *hd-append2* [*simp*]: $xs \neq [] ==> hd \ (xs \ @ \ ys) = hd \ xs$
by (*simp add*: *hd-append split*: *list.split*)

lemma *tl-append*: $tl \ (xs \ @ \ ys) = (case \ xs \ of \ [] => tl \ ys \mid z\#zs => zs \ @ \ ys)$
by (*simp split*: *list.split*)

lemma *tl-append2* [*simp*]: $xs \neq [] ==> tl \ (xs \ @ \ ys) = tl \ xs \ @ \ ys$
by (*simp add*: *tl-append split*: *list.split*)

lemma *Cons-eq-append-conv*: $x\#xs = ys@zs =$
$(ys = [] \ \& \ x\#xs = zs \mid (EX \ ys'. \ x\#ys' = ys \ \& \ xs = ys'@zs))$
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*: $xs = ys ==> xs = [] \ @ \ ys$
by *simp*

lemma *Cons-eq-appendI*:
$[| \ x \ \# \ xs1 = ys; \ xs = xs1 \ @ \ zs \ |] ==> x \ \# \ xs = ys \ @ \ zs$
by (*drule sym*) *simp*

lemma *append-eq-appendI*:
$[| \ xs \ @ \ xs1 = zs; \ ys = xs1 \ @ \ us \ |] ==> xs \ @ \ ys = zs \ @ \ us$
by (*drule sym*) *simp*

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

ML-setup ⟪
```
local

fun last (cons as Const(List.list.Cons,-) $ - $ xs) =
  (case xs of Const(List.list.Nil,-) => cons | - => last xs)
  | last (Const(List.append,-) $ - $ 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.append,-) $ xs) $ ys) = app $ butlast ys
  | butlast xs = Const(List.list.Nil,fastype-of xs);

val rearr-ss = HOL-basic-ss addsimps [@{thm append-assoc},
  @{thm append-Nil}, @{thm append-Cons}];

fun list-eq ss (F as (eq as Const(-,eqT)) $ lhs $ rhs) =
  let
    val lastl = last lhs and lastr = last rhs;
    fun rearr conv =
      let
        val lhs1 = butlast lhs and rhs1 = butlast rhs;
        val Type(-,listT::-) = eqT
        val appT = [listT,listT] ---> listT
        val app = Const(List.append,appT)
        val F2 = eq $ (app$lhs1$lastl) $ (app$rhs1$lastr)
        val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (F,F2));
        val thm = Goal.prove (Simplifier.the-context ss) [] [] eq
          (K (simp-tac (Simplifier.inherit-context ss rearr-ss) 1));
      in SOME ((conv RS (thm RS trans)) RS eq-reflection) end;

  in
    if list1 lastl andalso list1 lastr then rearr @{thm append1-eq-conv}
    else if lastl aconv lastr then rearr @{thm append-same-eq}
    else NONE
  end;

in

val list-eq-simproc =
  Simplifier.simproc @{theory} list-eq [(xs::'a list) = ys] (K list-eq);

end;
```

Addsimprocs [*list-eq-simproc*];
⟫

48.1.5 *map*

lemma *map-ext*: (!!*x*. *x* : *set xs* −−> *f x* = *g x*) ==> *map f xs* = *map g xs*
by (*induct xs*) *simp-all*

lemma *map-ident* [*simp*]: *map* (λ*x*. *x*) = (λ*xs*. *xs*)
by (*rule ext*, *induct-tac xs*) *auto*

lemma *map-append* [*simp*]: *map f* (*xs* @ *ys*) = *map f xs* @ *map f ys*
by (*induct xs*) *auto*

lemma *map-compose*: *map* (*f o g*) *xs* = *map f* (*map g xs*)
by (*induct xs*) (*auto simp add*: *o-def*)

lemma *rev-map*: *rev* (*map f xs*) = *map f* (*rev xs*)
by (*induct xs*) *auto*

lemma *map-eq-conv*[*simp*]: (*map f xs* = *map g xs*) = (!*x* : *set xs*. *f x* = *g x*)
by (*induct xs*) *auto*

lemma *map-cong* [*fundef-cong*, *recdef-cong*]:
xs = *ys* ==> (!!*x*. *x* : *set ys* ==> *f x* = *g x*) ==> *map f xs* = *map g ys*
— a congruence rule for *map*
by *simp*

lemma *map-is-Nil-conv* [*iff*]: (*map f xs* = []) = (*xs* = [])
by (*cases xs*) *auto*

lemma *Nil-is-map-conv* [*iff*]: ([] = *map f xs*) = (*xs* = [])
by (*cases xs*) *auto*

lemma *map-eq-Cons-conv*:
(*map f xs* = *y*#*ys*) = (∃ *z zs*. *xs* = *z*#*zs* ∧ *f z* = *y* ∧ *map f zs* = *ys*)
by (*cases xs*) *auto*

lemma *Cons-eq-map-conv*:
(*x*#*xs* = *map f ys*) = (∃ *z zs*. *ys* = *z*#*zs* ∧ *x* = *f z* ∧ *xs* = *map f zs*)
by (*cases ys*) *auto*

lemmas *map-eq-Cons-D* = *map-eq-Cons-conv* [*THEN iffD1*]
lemmas *Cons-eq-map-D* = *Cons-eq-map-conv* [*THEN iffD1*]
declare *map-eq-Cons-D* [*dest*!] *Cons-eq-map-D* [*dest*!]

lemma *ex-map-conv*:
(*EX xs*. *ys* = *map f xs*) = (*ALL y* : *set ys*. *EX x*. *y* = *f x*)

by(*induct ys*, *auto simp add*: *Cons-eq-map-conv*)

lemma *map-eq-imp-length-eq*:
 map f xs = *map f ys* ==> *length xs* = *length ys*
apply (*induct ys arbitrary*: *xs*)
 apply *simp*
apply (*metis Suc-length-conv length-map*)
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*) $\Longrightarrow$ (*map f xs* = *map f ys*) = (*xs* = *ys*)
by(*blast dest*:*map-inj-on*)

lemma *map-injective*:
 map f xs = *map f ys* ==> *inj f* ==> *xs* = *ys*
by (*induct ys arbitrary*: *xs*) (*auto dest*!:*injD*)

lemma *inj-map-eq-map*[*simp*]: *inj f* $\Longrightarrow$ (*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*)

48.1.6 *rev*

lemma *rev-append* [*simp*]: $rev\ (xs\ @\ ys) = rev\ ys\ @\ rev\ xs$
by (*induct xs*) *auto*

lemma *rev-rev-ident* [*simp*]: $rev\ (rev\ xs) = xs$
by (*induct xs*) *auto*

lemma *rev-swap*: $(rev\ xs = ys) = (xs = rev\ ys)$
by *auto*

lemma *rev-is-Nil-conv* [*iff*]: $(rev\ xs = []) = (xs = [])$
by (*induct xs*) *auto*

lemma *Nil-is-rev-conv* [*iff*]: $([] = rev\ xs) = (xs = [])$
by (*induct xs*) *auto*

lemma *rev-singleton-conv* [*simp*]: $(rev\ xs = [x]) = (xs = [x])$
by (*cases xs*) *auto*

lemma *singleton-rev-conv* [*simp*]: $([x] = rev\ xs) = (xs = [x])$
by (*cases xs*) *auto*

lemma *rev-is-rev-conv* [*iff*]: $(rev\ xs = rev\ ys) = (xs = ys)$
apply (*induct xs arbitrary*: *ys*, *force*)
apply (*case-tac ys*, *simp*, *force*)
done

lemma *inj-on-rev*[*iff*]: *inj-on rev A*
by(*simp add*:*inj-on-def*)

lemma *rev-induct* [*case-names Nil snoc*]:
$[|\ P\ [];\ !!x\ xs.\ P\ xs ==> P\ (xs\ @\ [x])\ |] ==> P\ xs$
apply(*simplesubst rev-rev-ident*[*symmetric*])

apply(*rule-tac list* = *rev xs* **in** *list.induct*, *simp-all*)
done

lemma *rev-exhaust* [*case-names Nil snoc*]:
 (xs = [] ==> P) ==>(!!*ys y*. xs = ys @ [y] ==> P) ==> P
by (*induct xs rule*: *rev-induct*) *auto*

lemmas *rev-cases* = *rev-exhaust*

lemma *rev-eq-Cons-iff* [*iff*]: (*rev xs* = $y\#ys$) = (xs = *rev ys* @ [y])
by(*rule rev-cases*[*of xs*]) *auto*

48.1.7 *set*

lemma *finite-set* [*iff*]: *finite* (*set xs*)
by (*induct xs*) *auto*

lemma *set-append* [*simp*]: *set* (xs @ ys) = (*set xs* $\cup$ *set ys*)
by (*induct xs*) *auto*

lemma *hd-in-set*[*simp*]: $xs \neq []$ $\Longrightarrow$ *hd xs* : *set xs*
by(*cases xs*) *auto*

lemma *set-subset-Cons*: *set xs* $\subseteq$ *set* (x # xs)
by *auto*

lemma *set-ConsD*: $y \in$ *set* (x # xs) $\Longrightarrow$ $y{=}x \vee y \in$ *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* $\wedge$ P x}
by (*induct xs*) *auto*

lemma *set-upt* [*simp*]: *set*[i..<j] = {k. $i \le k \wedge k < j$}
apply (*induct j*, *simp-all*)
apply (*erule ssubst*, *auto*)
done

```
lemma in-set-conv-decomp: (x : set xs) = (∃ ys zs. xs = ys @ x # zs)
proof (induct xs)
  case Nil show ?case by simp
next
  case (Cons a xs)
  show ?case
  proof
    assume x ∈ set (a # xs)
    with Cons show ∃ ys zs. a # xs = ys @ x # zs
      by (auto intro: Cons-eq-appendI)
  next
    assume ∃ ys zs. a # xs = ys @ x # zs
    then obtain ys zs where eq: a # xs = ys @ x # zs by blast
    show x ∈ set (a # xs)
      by (cases ys) (auto simp add: eq)
  qed
qed

lemma split-list: x : set xs ⟹ ∃ ys zs. xs = ys @ x # zs
  by (rule in-set-conv-decomp [THEN iffD1])

lemma in-set-conv-decomp-first:
  (x : set xs) = (∃ ys zs. xs = ys @ x # zs ∧ x ∉ set ys)
proof (induct xs)
  case Nil show ?case by simp
next
  case (Cons a xs)
  show ?case
  proof cases
    assume x = a thus ?case using Cons by fastsimp
  next
    assume x ≠ a
    show ?case
    proof
      assume x ∈ set (a # xs)
      with Cons and ‹x ≠ a› show ∃ ys zs. a # xs = ys @ x # zs ∧ x ∉ set ys
        by (fastsimp intro!: Cons-eq-appendI)
    next
      assume ∃ ys zs. a # xs = ys @ x # zs ∧ x ∉ set ys
      then obtain ys zs where eq: a # xs = ys @ x # zs by blast
      show x ∈ set (a # xs) by (cases ys) (auto simp add: eq)
    qed
  qed
qed

lemma split-list-first: x : set xs ⟹ ∃ ys zs. xs = ys @ x # zs ∧ x ∉ set ys
  by (rule in-set-conv-decomp-first [THEN iffD1])
```

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*)

48.1.8 *filter*

lemma *filter-append* [*simp*]: *filter P* (*xs @ ys*) = *filter P xs @ filter P ys*
by (*induct xs*) *auto*

lemma *rev-filter*: *rev* (*filter P xs*) = *filter P* (*rev xs*)
by (*induct xs*) *simp-all*

lemma *filter-filter* [*simp*]: *filter P* (*filter Q xs*) = *filter* (λx. *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 *sum-length-filter-compl*:
length(*filter P xs*) + *length*(*filter* (%*x*. ~*P x*) *xs*) = *length xs*
by(*induct xs*) *simp-all*

lemma *filter-True* [*simp*]: $\forall$ *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* = []) = ($\forall$ *x*$\in$*set xs*. $\neg$ *P x*)
by (*induct xs*) *simp-all*

lemma *filter-id-conv*: (*filter P xs = xs*) = ($\forall$ *x*$\in$*set xs*. *P x*)
apply (*induct xs*)
 apply *auto*
apply(*cut-tac P=P* **and** *xs=xs* **in** *length-filter-le*)
apply *simp*
done

lemma *filter-map*:
filter P (*map f xs*) = *map f* (*filter* (*P o f*) *xs*)
by (*induct xs*) *simp-all*

lemma *length-filter-map*[*simp*]:
length (*filter P* (*map f xs*)) = *length*(*filter* (*P o f*) *xs*)
by (*simp add*:*filter-map*)

```
lemma filter-is-subset [simp]: set (filter P xs) ≤ set xs
by auto

lemma length-filter-less:
  ⟦ x : set xs; ~ P x ⟧ ⟹ length(filter P xs) < length xs
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs) thus ?case
    apply (auto split:split-if-asm)
    using length-filter-le[of P xs] apply arith
  done
qed

lemma length-filter-conv-card:
 length(filter p xs) = card{i. i < length xs & p(xs!i)}
proof (induct xs)
  case Nil thus ?case by simp
next
  case (Cons x xs)
  let ?S = {i. i < length xs & p(xs!i)}
  have fin: finite ?S by(fast intro: bounded-nat-set-is-finite)
  show ?case (is ?l = card ?S')
  proof (cases)
    assume p x
    hence eq: ?S' = insert 0 (Suc ` ?S)
      by(auto simp: image-def split:nat.split dest:gr0-implies-Suc)
    have length (filter p (x # xs)) = Suc(card ?S)
      using Cons ‹p x› by simp
    also have ... = Suc(card(Suc ` ?S)) using fin
      by (simp add: card-image inj-Suc)
    also have ... = card ?S' using eq fin
      by (simp add:card-insert-if) (simp add:image-def)
    finally show ?thesis .
  next
    assume ¬ p x
    hence eq: ?S' = Suc ` ?S
      by(auto simp add: image-def split:nat.split elim:lessE)
    have length (filter p (x # xs)) = card ?S
      using Cons ‹¬ p x› by simp
    also have ... = card(Suc ` ?S) using fin
      by (simp add: card-image inj-Suc)
    also have ... = card ?S' using eq fin
      by (simp add:card-insert-if)
    finally show ?thesis .
  qed
qed
```

lemma *Cons-eq-filterD*:
$x\#xs = filter\ P\ ys \Longrightarrow$
$\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
(**is** - $\Longrightarrow \exists\, us\ vs.\ ?P\ ys\ us\ vs$)
proof(*induct ys*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons y ys*)
  show *?case* (**is** $\exists\, x.\ ?Q\ x$)
  proof *cases*
    assume *Py*: $P\ y$
    show *?thesis*
    proof *cases*
      assume $x = y$
      with *Py Cons.prems* **have** *?Q* [] **by** *simp*
      then show *?thesis* **..**
    next
      assume $x \neq y$
      with *Py Cons.prems* **show** *?thesis* **by** *simp*
    qed
  next
    assume $\neg\ P\ y$
    with *Cons* **obtain** *us vs* **where** $?P\ (y\#ys)\ (y\#us)\ vs$ **by** *fastsimp*
    then have $?Q\ (y\#us)$ **by** *simp*
    then show *?thesis* **..**
  qed
qed

lemma *filter-eq-ConsD*:
$filter\ P\ ys = x\#xs \Longrightarrow$
$\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs$
by(*rule Cons-eq-filterD*) *simp*

lemma *filter-eq-Cons-iff*:
$(filter\ P\ ys = x\#xs) =$
$(\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
by(*auto dest*:*filter-eq-ConsD*)

lemma *Cons-eq-filter-iff*:
$(x\#xs = filter\ P\ ys) =$
$(\exists\, us\ vs.\ ys = us\ @\ x\ \#\ vs \wedge (\forall\, u{\in}set\ us.\ \neg\ P\ u) \wedge P\ x \wedge xs = filter\ P\ vs)$
by(*auto dest*:*Cons-eq-filterD*)

lemma *filter-cong*[*fundef-cong*, *recdef-cong*]:
$xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ ys \Longrightarrow P\ x = Q\ x) \Longrightarrow filter\ P\ xs = filter\ Q\ ys$
apply *simp*
apply(*erule thin-rl*)
by (*induct ys*) *simp-all*

48.1.9 *concat*

lemma *concat-append* [*simp*]: *concat* (*xs* @ *ys*) = *concat xs* @ *concat ys*
by (*induct xs*) *auto*

lemma *concat-eq-Nil-conv* [*simp*]: (*concat xss* = []) = (∀ *xs* ∈ *set xss*. *xs* = [])
by (*induct xss*) *auto*

lemma *Nil-eq-concat-conv* [*simp*]: ([] = *concat xss*) = (∀ *xs* ∈ *set xss*. *xs* = [])
by (*induct xss*) *auto*

lemma *set-concat* [*simp*]: *set* (*concat xs*) = (*UN x*:*set xs*. *set x*)
by (*induct xs*) *auto*

lemma *concat-map-singleton*[*simp*]: *concat*(*map* (%*x*. [*f x*]) *xs*) = *map f xs*
by (*induct xs*) *auto*

lemma *map-concat*: *map f* (*concat xs*) = *concat* (*map* (*map f*) *xs*)
by (*induct xs*) *auto*

lemma *filter-concat*: *filter p* (*concat xs*) = *concat* (*map* (*filter p*) *xs*)
by (*induct xs*) *auto*

lemma *rev-concat*: *rev* (*concat xs*) = *concat* (*map rev* (*rev xs*))
by (*induct xs*) *auto*

48.1.10 *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*:
  (*xs* @ *ys*)!*n* = (*if n* < *length xs then xs*!*n else ys*!(*n* − *length xs*))
apply (*induct xs arbitrary*: *n*, *simp*)
apply (*case-tac n*, *auto*)
done

lemma *nth-append-length* [*simp*]: (*xs* @ *x* # *ys*) ! *length xs* = *x*
by (*induct xs*) *auto*

lemma *nth-append-length-plus*[*simp*]: (*xs* @ *ys*) ! (*length xs* + *n*) = *ys* ! *n*
by (*induct xs*) *auto*

lemma *nth-map* [*simp*]: *n* < *length xs* ==> (*map f xs*)!*n* = *f*(*xs*!*n*)
apply (*induct xs arbitrary*: *n*, *simp*)

apply (*case-tac n*, *auto*)
done

lemma *hd-conv-nth*: $xs \neq [] \Longrightarrow hd\ xs = xs!0$
by(*cases xs*) *simp-all*

lemma *list-eq-iff-nth-eq*:
$(xs = ys) = (length\ xs = length\ ys \wedge (ALL\ i{<}length\ xs.\ xs!i = ys!i))$
apply(*induct xs arbitrary*: *ys*)
apply *force*
apply(*case-tac ys*)
apply *simp*
apply(*simp add*:*nth-Cons split*:*nat.split*)**apply** *blast*
done

lemma *set-conv-nth*: $set\ xs = \{xs!i \mid i.\ i < length\ xs\}$
apply (*induct xs*, *simp*, *simp*)
apply *safe*
apply (*metis nat-case-0 nth.simps zero-less-Suc*)
apply (*metis less-Suc-eq-0-disj nth-Cons-Suc*)
apply (*case-tac i*, *simp*)
apply (*metis diff-Suc-Suc nat-case-Suc nth.simps zero-less-diff*)
done

lemma *in-set-conv-nth*: $(x \in set\ xs) = (\exists\, i < length\ xs.\ xs!i = x)$
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*)

lemma *rev-nth*:
$n < size\ xs \Longrightarrow rev\ xs\ !\ n = xs\ !\ (length\ xs - Suc\ n)$
proof (*induct xs arbitrary*: *n*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
hence *n*: $n < Suc\ (length\ xs)$ **by** *simp*

moreover
{ **assume** $n < length\ xs$
with n **obtain** n' **where** $length\ xs - n = Suc\ n'$
by $(cases\ length\ xs - n,\ auto)$
moreover
then have $length\ xs - Suc\ n = n'$ **by** *simp*
ultimately
have $xs\ !\ (length\ xs - Suc\ n) = (x\ \#\ xs)\ !\ (length\ xs - n)$ **by** *simp*
}
ultimately
show *?case* **by** (*clarsimp simp add*: *Cons nth-append*)
qed

48.1.11 *list-update*

lemma *length-list-update* [*simp*]: $length(xs[i{:=}x]) = length\ xs$
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *nth-list-update*:
$i < length\ xs{==}{>}\ (xs[i{:=}x])!j = (if\ i = j\ then\ x\ else\ xs!j)$
by (*induct xs arbitrary*: *i j*) (*auto simp add*: *nth-Cons split*: *nat.split*)

lemma *nth-list-update-eq* [*simp*]: $i < length\ xs\ {==}{>}\ (xs[i{:=}x])!i = x$
by (*simp add*: *nth-list-update*)

lemma *nth-list-update-neq* [*simp*]: $i \neq j\ {==}{>}\ xs[i{:=}x]!j = xs!j$
by (*induct xs arbitrary*: *i j*) (*auto simp add*: *nth-Cons split*: *nat.split*)

lemma *list-update-overwrite* [*simp*]:
$i < size\ xs\ {==}{>}\ xs[i{:=}x,\ i{:=}y] = xs[i{:=}y]$
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *list-update-id*[*simp*]: $xs[i := xs!i] = xs$
by (*induct xs arbitrary*: *i*) (*simp-all split*:*nat.splits*)

lemma *list-update-beyond*[*simp*]: $length\ xs \le i \Longrightarrow xs[i{:=}x] = xs$
apply (*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
done

lemma *list-update-same-conv*:
$i < length\ xs\ {==}{>}\ (xs[i := x] = xs) = (xs!i = x)$
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *list-update-append1*:
$i < size\ xs \Longrightarrow (xs\ @\ ys)[i{:=}x] = xs[i{:=}x]\ @\ ys$
apply (*induct xs arbitrary*: *i*, *simp*)

apply(*simp split*:*nat.split*)
done

lemma *list-update-append*:
$(xs\ @\ ys)\ [n{:=}\ x] =$
$(\textit{if}\ n < \textit{length}\ xs\ \textit{then}\ xs[n{:=}\ x]\ @\ ys\ \textit{else}\ xs\ @\ (ys\ [n{-}\textit{length}\ xs{:=}\ x]))$
by (*induct xs arbitrary*: *n*) (*auto split*:*nat.splits*)

lemma *list-update-length* [*simp*]:
$(xs\ @\ x\ \#\ ys)[\textit{length}\ xs := y] = (xs\ @\ y\ \#\ ys)$
by (*induct xs*, *auto*)

lemma *update-zip*:
length xs = length ys ==>
$(\textit{zip}\ xs\ ys)[i{:=}xy] = \textit{zip}\ (xs[i{:=}\textit{fst}\ xy])\ (ys[i{:=}\textit{snd}\ xy])$
by (*induct ys arbitrary*: *i xy xs*) (*auto*, *case-tac xs*, *auto split*: *nat.split*)

lemma *set-update-subset-insert*: $\textit{set}(xs[i{:=}x]) <= \textit{insert}\ x\ (\textit{set}\ xs)$
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *set-update-subsetI*: $[|\ \textit{set}\ xs <= A;\ x{:}A\ |] ==> \textit{set}(xs[i := x]) <= A$
by (*blast dest*!: *set-update-subset-insert* [*THEN subsetD*])

lemma *set-update-memI*: $n < \textit{length}\ xs \Longrightarrow x \in \textit{set}\ (xs[n := x])$
by (*induct xs arbitrary*: *n*) (*auto split*:*nat.splits*)

lemma *list-update-overwrite*:
$xs\ [i := x,\ i := y] = xs\ [i := y]$
apply (*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
done

lemma *list-update-swap*:
$i \neq i' \Longrightarrow xs\ [i := x,\ i' := x'] = xs\ [i' := x',\ i := x]$
apply (*induct xs arbitrary*: $i\ i'$)
apply *simp*
apply (*case-tac i*, *case-tac* i')
apply *auto*
apply (*case-tac* i')
apply *auto*
done

48.1.12 *last* and *butlast*

lemma *last-snoc* [*simp*]: $\textit{last}\ (xs\ @\ [x]) = x$
by (*induct xs*) *auto*

lemma *butlast-snoc* [*simp*]: *butlast* (*xs* @ [*x*]) = *xs*
by (*induct xs*) *auto*

lemma *last-ConsL*: $xs = [] \Longrightarrow last(x\#xs) = x$
by(*simp add*:*last.simps*)

lemma *last-ConsR*: $xs \neq [] \Longrightarrow last(x\#xs) = last\ xs$
by(*simp add*:*last.simps*)

lemma *last-append*: *last*(*xs* @ *ys*) = (*if ys* = [] *then last xs else last ys*)
by (*induct xs*) (*auto*)

lemma *last-appendL*[*simp*]: $ys = [] \Longrightarrow last(xs\ @\ ys) = last\ xs$
by(*simp add*:*last-append*)

lemma *last-appendR*[*simp*]: $ys \neq [] \Longrightarrow last(xs\ @\ ys) = last\ ys$
by(*simp add*:*last-append*)

lemma *hd-rev*: $xs \neq [] \Longrightarrow hd(rev\ xs) = last\ xs$
by(*rule rev-exhaust*[*of xs*]) *simp-all*

lemma *last-rev*: $xs \neq [] \Longrightarrow last(rev\ xs) = hd\ xs$
by(*cases xs*) *simp-all*

lemma *last-in-set*[*simp*]: $as \neq [] \Longrightarrow last\ as \in set\ as$
by (*induct as*) *auto*

lemma *length-butlast* [*simp*]: *length* (*butlast xs*) = *length xs* − *1*
by (*induct xs rule*: *rev-induct*) *auto*

lemma *butlast-append*:
butlast (*xs* @ *ys*) = (*if ys* = [] *then butlast xs else xs* @ *butlast ys*)
by (*induct xs arbitrary*: *ys*) *auto*

lemma *append-butlast-last-id* [*simp*]:
xs ≠ [] ==> *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*) | *x* : *set* (*butlast ys*) ==> *x* : *set* (*butlast* (*xs* @ *ys*))
by (*auto dest*: *in-set-butlastD simp add*: *butlast-append*)

lemma *last-drop*[*simp*]: $n < length\ xs \Longrightarrow last\ (drop\ n\ xs) = last\ xs$
apply (*induct xs arbitrary*: *n*)
apply *simp*
apply (*auto split*:*nat.split*)

done

lemma *last-conv-nth*: *xs*≠[] ⟹ *last xs* = *xs*!(*length xs* − *1*)
by(*induct xs*)(*auto simp*:*neq-Nil-conv*)

48.1.13 *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*: *drop n* (*tl xs*) = *tl*(*drop n xs*)
by(*induct xs arbitrary*: *n*, *simp-all add*:*drop-Cons drop-Suc split*:*nat.split*)

lemma *nth-via-drop*: *drop n xs* = *y*#*ys* ⟹ *xs*!*n* = *y*
apply (*induct xs arbitrary*: *n*, *simp*)
apply(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)
done

lemma *take-Suc-conv-app-nth*:
  i < *length xs* ⟹ *take* (*Suc i*) *xs* = *take i xs* @ [*xs*!*i*]
apply (*induct xs arbitrary*: *i*, *simp*)
apply (*case-tac i*, *auto*)
done

lemma *drop-Suc-conv-tl*:
  i < *length xs* ⟹ (*xs*!*i*) # (*drop* (*Suc i*) *xs*) = *drop i xs*
apply (*induct xs arbitrary*: *i*, *simp*)
apply (*case-tac i*, *auto*)
done

lemma *length-take* [*simp*]: *length* (*take n xs*) = *min* (*length xs*) *n*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *length-drop* [*simp*]: *length* (*drop n xs*) = (*length xs* − *n*)
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-all* [*simp*]: *length xs* <= *n* ==> *take n xs* = *xs*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-all* [*simp*]: *length xs* <= *n* ==> *drop n xs* = []
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-append* [*simp*]:
  take n (*xs* @ *ys*) = (*take n xs* @ *take* (*n* − *length xs*) *ys*)
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-append* [*simp*]:
  drop n (*xs* @ *ys*) = *drop n xs* @ *drop* (*n* − *length xs*) *ys*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-take* [*simp*]: *take n* (*take m xs*) = *take* (*min n m*) *xs*
apply (*induct m arbitrary*: *xs n*, *auto*)
apply (*case-tac xs*, *auto*)
apply (*case-tac n*, *auto*)
done

lemma *drop-drop* [*simp*]: *drop n* (*drop m xs*) = *drop* (*n* + *m*) *xs*
apply (*induct m arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *take-drop*: *take n* (*drop m xs*) = *drop m* (*take* (*n* + *m*) *xs*)
apply (*induct m arbitrary*: *xs n*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *drop-take*: *drop n* (*take m xs*) = *take* (*m*−*n*) (*drop n xs*)
apply(*induct xs arbitrary*: *m n*)
 apply *simp*
apply(*simp add*: *take-Cons drop-Cons split*:*nat.split*)
done

lemma *append-take-drop-id* [*simp*]: *take n xs* @ *drop n xs* = *xs*
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *take-eq-Nil*[*simp*]: (*take n xs* = []) = (*n* = *0* ∨ *xs* = [])
apply(*induct xs arbitrary*: *n*)
 apply *simp*
apply(*simp add*:*take-Cons split*:*nat.split*)

done

lemma *drop-eq-Nil*[*simp*]: (*drop n xs* = []) = (*length xs* <= *n*)
apply(*induct xs arbitrary*: *n*)
apply *simp*
apply(*simp add*:*drop-Cons split*:*nat.split*)
done

lemma *take-map*: *take n* (*map f xs*) = *map f* (*take n xs*)
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *drop-map*: *drop n* (*map f xs*) = *map f* (*drop n xs*)
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *rev-take*: *rev* (*take i xs*) = *drop* (*length xs* − *i*) (*rev xs*)
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *auto*)
done

lemma *rev-drop*: *rev* (*drop i xs*) = *take* (*length xs* − *i*) (*rev xs*)
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *auto*)
done

lemma *nth-take* [*simp*]: *i* < *n* ==> (*take n xs*)!*i* = *xs*!*i*
apply (*induct xs arbitrary*: *i n*, *auto*)
apply (*case-tac n*, *blast*)
apply (*case-tac i*, *auto*)
done

lemma *nth-drop* [*simp*]:
  n + *i* <= *length xs* ==> (*drop n xs*)!*i* = *xs*!(*n* + *i*)
apply (*induct n arbitrary*: *xs i*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *hd-drop-conv-nth*: ⟦ *xs* ≠ []; *n* < *length xs* ⟧ ⟹ *hd*(*drop n xs*) = *xs*!*n*
by(*simp add*: *hd-conv-nth*)

lemma *set-take-subset*: *set*(*take n xs*) ⊆ *set xs*
by(*induct xs arbitrary*: *n*)(*auto simp*:*take-Cons split*:*nat.split*)

lemma *set-drop-subset*: *set*(*drop n xs*) ⊆ *set xs*
by(*induct xs arbitrary*: *n*)(*auto simp*:*drop-Cons split*:*nat.split*)

lemma *in-set-takeD*: $x : set(take\ n\ xs) \Longrightarrow x : set\ xs$
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*:
$(xs\ @\ ys = zs) = (xs = take\ (length\ xs)\ zs \wedge ys = drop\ (length\ xs)\ zs)$
apply (*induct xs arbitrary*: *zs*, *simp*, *clarsimp*)
apply (*case-tac zs*, *auto*)
done

lemma *take-add*:
$i{+}j \leq length(xs) \Longrightarrow take\ (i{+}j)\ xs = take\ i\ xs\ @\ take\ j\ (drop\ i\ xs)$
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *simp-all*)
done

lemma *append-eq-append-conv-if*:
$(xs_1\ @\ xs_2 = ys_1\ @\ ys_2) =$
$(if\ size\ xs_1 \leq size\ ys_1$
$then\ xs_1 = take\ (size\ xs_1)\ ys_1 \wedge xs_2 = drop\ (size\ xs_1)\ ys_1\ @\ ys_2$
$else\ take\ (size\ ys_1)\ xs_1 = ys_1 \wedge drop\ (size\ ys_1)\ xs_1\ @\ xs_2 = ys_2)$
apply(*induct* xs_1 *arbitrary*: ys_1)
apply *simp*
apply(*case-tac* ys_1)
apply *simp-all*
done

lemma *take-hd-drop*:
$n < length\ xs \Longrightarrow take\ n\ xs\ @\ [hd\ (drop\ n\ xs)] = take\ (n{+}1)\ xs$
apply(*induct xs arbitrary*: *n*)
apply *simp*
apply(*simp add*:*drop-Cons split*:*nat.split*)
done

lemma *id-take-nth-drop*:
$i < length\ xs \Longrightarrow xs = take\ i\ xs\ @\ xs!i\ \#\ drop\ (Suc\ i)\ xs$
proof –
assume *si*: $i < length\ xs$
hence $xs = take\ (Suc\ i)\ xs\ @\ drop\ (Suc\ i)\ xs$ **by** *auto*
moreover
from *si* **have** $take\ (Suc\ i)\ xs = take\ i\ xs\ @\ [xs!i]$
apply (*rule-tac take-Suc-conv-app-nth*) **by** *arith*
ultimately show *?thesis* **by** *auto*
qed

lemma *upd-conv-take-nth-drop*:
$i < length\ xs \Longrightarrow xs[i{:=}a] = take\ i\ xs\ @\ a\ \#\ drop\ (Suc\ i)\ xs$

proof –
  assume *i*: $i < length\ xs$
  have $xs[i{:=}a] = (take\ i\ xs\ @\ xs!i\ \#\ drop\ (Suc\ i)\ xs)[i{:=}a]$
    by(*rule arg-cong*[*OF id-take-nth-drop*[*OF i*]])
  also have $\ldots = take\ i\ xs\ @\ a\ \#\ drop\ (Suc\ i)\ xs$
    using *i* **by** (*simp add*: *list-update-append*)
  finally show *?thesis* **.**
qed

lemma *nth-drop′*:
  $i < length\ xs \Longrightarrow xs\ !\ i\ \#\ drop\ (Suc\ i)\ xs = drop\ i\ xs$
apply (*induct i arbitrary*: *xs*)
apply (*simp add*: *neq-Nil-conv*)
apply (*erule exE*)+
apply *simp*
apply (*case-tac xs*)
apply *simp-all*
done

48.1.14 *takeWhile* and *dropWhile*

lemma *takeWhile-dropWhile-id* [*simp*]: $takeWhile\ P\ xs\ @\ dropWhile\ P\ xs = xs$
by (*induct xs*) *auto*

lemma *takeWhile-append1* [*simp*]:
$[|\ x{:}set\ xs;\ \tilde{}P(x)|] ==> takeWhile\ P\ (xs\ @\ ys) = takeWhile\ P\ xs$
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*: $\neg\ P\ x ==> takeWhile\ P\ (xs\ @\ (x\#l)) = takeWhile\ P\ xs$
by (*induct xs*) *auto*

lemma *dropWhile-append1* [*simp*]:
$[|\ x : set\ xs;\ \tilde{}P(x)|] ==> dropWhile\ P\ (xs\ @\ ys) = (dropWhile\ P\ xs)@ys$
by (*induct xs*) *auto*

lemma *dropWhile-append2* [*simp*]:
$(!!x.\ x{:}set\ xs ==> P(x)) ==> dropWhile\ P\ (xs\ @\ ys) = dropWhile\ P\ ys$
by (*induct xs*) *auto*

lemma *set-takeWhileD*: $x : set\ (takeWhile\ P\ xs) ==> x : set\ xs \wedge P\ x$
by (*induct xs*) (*auto split*: *split-if-asm*)

lemma *takeWhile-eq-all-conv*[*simp*]:
 $(takeWhile\ P\ xs = xs) = (\forall\, x \in set\ xs.\ P\ x)$
by(*induct xs*, *auto*)

lemma *dropWhile-eq-Nil-conv*[*simp*]:
$(dropWhile\ P\ xs = []) = (\forall x \in set\ xs.\ P\ x)$
by(*induct xs*, *auto*)

lemma *dropWhile-eq-Cons-conv*:
$(dropWhile\ P\ xs = y\#ys) = (xs = takeWhile\ P\ xs\ @\ y\ \#\ ys\ \&\ \neg\ P\ y)$
by(*induct xs*, *auto*)

The following two lemmmas could be generalized to an arbitrary property.

lemma *takeWhile-neq-rev*: $[\![distinct\ xs;\ x \in set\ xs]\!] \Longrightarrow$
$takeWhile\ (\lambda y.\ y \neq x)\ (rev\ xs) = rev\ (tl\ (dropWhile\ (\lambda y.\ y \neq x)\ xs))$
by(*induct xs*) (*auto simp*: *takeWhile-tail*[**where** *l*=[]])

lemma *dropWhile-neq-rev*: $[\![distinct\ xs;\ x \in set\ xs]\!] \Longrightarrow$
$dropWhile\ (\lambda y.\ y \neq x)\ (rev\ xs) = x\ \#\ rev\ (takeWhile\ (\lambda y.\ y \neq x)\ xs)$
apply(*induct xs*)
apply *simp*
apply *auto*
apply(*subst dropWhile-append2*)
apply *auto*
done

lemma *takeWhile-not-last*:
$[\![\ xs \neq [];\ distinct\ xs]\!] \Longrightarrow takeWhile\ (\lambda y.\ y \neq last\ xs)\ xs = butlast\ xs$
apply(*induct xs*)
apply *simp*
apply(*case-tac xs*)
apply(*auto*)
done

lemma *takeWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| *l* = *k*; !!*x*. *x* : *set l* ==> *P x* = *Q x* |]
==> *takeWhile P l* = *takeWhile Q k*
by (*induct k arbitrary*: *l*) (*simp-all*)

lemma *dropWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| *l* = *k*; !!*x*. *x* : *set l* ==> *P x* = *Q x* |]
==> *dropWhile P l* = *dropWhile Q k*
by (*induct k arbitrary*: *l*, *simp-all*)

48.1.15 *zip*

lemma *zip-Nil* [*simp*]: *zip* [] *ys* = []
by (*induct ys*) *auto*

lemma *zip-Cons-Cons* [*simp*]: $zip\ (x\ \#\ xs)\ (y\ \#\ ys) = (x,\ y)\ \#\ zip\ xs\ ys$
by *simp*

declare *zip-Cons* [*simp del*]

lemma *zip-Cons1*:
zip (*x*#*xs*) *ys* = (*case ys of* [] ⇒ [] | *y*#*ys* ⇒ (*x*,*y*)#*zip xs ys*)
by(*auto split*:*list.split*)

lemma *length-zip* [*simp*]:
length (*zip xs ys*) = *min* (*length xs*) (*length ys*)
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *zip-append1*:
zip (*xs* @ *ys*) *zs* =
zip xs (*take* (*length xs*) *zs*) @ *zip ys* (*drop* (*length xs*) *zs*)
by (*induct xs zs rule*:*list-induct2'*) *auto*

lemma *zip-append2*:
zip xs (*ys* @ *zs*) =
zip (*take* (*length ys*) *xs*) *ys* @ *zip* (*drop* (*length ys*) *xs*) *zs*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *zip-append* [*simp*]:
[| *length xs* = *length us*; *length ys* = *length vs* |] ==>
zip (*xs*@*ys*) (*us*@*vs*) = *zip xs us* @ *zip ys vs*
by (*simp add*: *zip-append1*)

lemma *zip-rev*:
length xs = *length ys* ==> *zip* (*rev xs*) (*rev ys*) = *rev* (*zip xs ys*)
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *map-zip-map*:
map f (*zip* (*map g xs*) *ys*) = *map* (%(*x*,*y*). *f*(*g x*, *y*)) (*zip xs ys*)
apply(*induct xs arbitrary*:*ys*) **apply** *simp*
apply(*case-tac ys*)
apply *simp-all*
done

lemma *map-zip-map2*:
map f (*zip xs* (*map g ys*)) = *map* (%(*x*,*y*). *f*(*x*, *g y*)) (*zip xs ys*)
apply(*induct xs arbitrary*:*ys*) **apply** *simp*
apply(*case-tac ys*)
apply *simp-all*
done

lemma *nth-zip* [*simp*]:
[| *i* < *length xs*; *i* < *length ys*|] ==> (*zip xs ys*)!*i* = (*xs*!*i*, *ys*!*i*)
apply (*induct ys arbitrary*: *i xs*, *simp*)
apply (*case-tac xs*)
apply (*simp-all add*: *nth.simps split*: *nat.split*)
done

lemma *set-zip*:
set (*zip xs ys*) = {(*xs*!*i*, *ys*!*i*) | *i*. *i* < *min* (*length xs*) (*length ys*)}
by (*simp add*: *set-conv-nth cong*: *rev-conj-cong*)

lemma *zip-update*:
length xs = *length ys* ==> *zip* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*]) = (*zip xs ys*)[*i*:=(*x*,*y*)]
by (*rule sym*, *simp add*: *update-zip*)

lemma *zip-replicate* [*simp*]:
zip (*replicate i x*) (*replicate j y*) = *replicate* (*min i j*) (*x*,*y*)
apply (*induct i arbitrary*: *j*, *auto*)
apply (*case-tac j*, *auto*)
done

lemma *take-zip*:
take n (*zip xs ys*) = *zip* (*take n xs*) (*take n ys*)
apply (*induct n arbitrary*: *xs ys*)
apply *simp*
apply (*case-tac xs*, *simp*)
apply (*case-tac ys*, *simp-all*)
done

lemma *drop-zip*:
drop n (*zip xs ys*) = *zip* (*drop n xs*) (*drop n ys*)
apply (*induct n arbitrary*: *xs ys*)
apply *simp*
apply (*case-tac xs*, *simp*)
apply (*case-tac ys*, *simp-all*)
done

lemma *set-zip-leftD*:
$(x,y) \in$ *set* (*zip xs ys*) $\Longrightarrow$ $x \in$ *set xs*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *set-zip-rightD*:
$(x,y) \in$ *set* (*zip xs ys*) $\Longrightarrow$ $y \in$ *set ys*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *in-set-zipE*:
(x,y) : *set*(*zip xs ys*) $\Longrightarrow$ ($[\![$ x : *set xs*; y : *set ys* $]\!]$ $\Longrightarrow$ R) $\Longrightarrow$ R
by(*blast dest*: *set-zip-leftD set-zip-rightD*)

48.1.16 *list-all2*

lemma *list-all2-lengthD* [*intro?*]:
list-all2 P xs ys ==> *length xs* = *length ys*
by (*simp add*: *list-all2-def*)

lemma *list-all2-Nil* [*iff*, *code*]: *list-all2 P* [] *ys* = (*ys* = [])
by (*simp add*: *list-all2-def*)

lemma *list-all2-Nil2* [*iff*, *code*]: *list-all2 P xs* [] = (*xs* = [])
by (*simp add*: *list-all2-def*)

lemma *list-all2-Cons* [*iff*, *code*]:
list-all2 P (*x* # *xs*) (*y* # *ys*) = (*P x y* ∧ *list-all2 P xs ys*)
by (*auto simp add*: *list-all2-def*)

lemma *list-all2-Cons1*:
list-all2 P (*x* # *xs*) *ys* = (∃ *z zs*. *ys* = *z* # *zs* ∧ *P x z* ∧ *list-all2 P xs zs*)
by (*cases ys*) *auto*

lemma *list-all2-Cons2*:
list-all2 P xs (*y* # *ys*) = (∃ *z zs*. *xs* = *z* # *zs* ∧ *P z y* ∧ *list-all2 P zs ys*)
by (*cases xs*) *auto*

lemma *list-all2-rev* [*iff*]:
list-all2 P (*rev xs*) (*rev ys*) = *list-all2 P xs ys*
by (*simp add*: *list-all2-def zip-rev cong*: *conj-cong*)

lemma *list-all2-rev1*:
list-all2 P (*rev xs*) *ys* = *list-all2 P xs* (*rev ys*)
by (*subst list-all2-rev* [*symmetric*]) *simp*

lemma *list-all2-append1*:
list-all2 P (*xs* @ *ys*) *zs* =
(*EX us vs*. *zs* = *us* @ *vs* ∧ *length us* = *length xs* ∧ *length vs* = *length ys* ∧
list-all2 P xs us ∧ *list-all2 P ys vs*)
apply (*simp add*: *list-all2-def zip-append1*)
apply (*rule iffI*)
apply (*rule-tac x* = *take* (*length xs*) *zs* **in** *exI*)
apply (*rule-tac x* = *drop* (*length xs*) *zs* **in** *exI*)
apply (*force split*: *nat-diff-split simp add*: *min-def*, *clarify*)
apply (*simp add*: *ball-Un*)
done

lemma *list-all2-append2*:
list-all2 P xs (*ys* @ *zs*) =
(*EX us vs*. *xs* = *us* @ *vs* ∧ *length us* = *length ys* ∧ *length vs* = *length zs* ∧
list-all2 P us ys ∧ *list-all2 P vs zs*)
apply (*simp add*: *list-all2-def zip-append2*)
apply (*rule iffI*)
apply (*rule-tac x* = *take* (*length ys*) *xs* **in** *exI*)
apply (*rule-tac x* = *drop* (*length ys*) *xs* **in** *exI*)
apply (*force split*: *nat-diff-split simp add*: *min-def*, *clarify*)
apply (*simp add*: *ball-Un*)
done

lemma *list-all2-append*:
  length xs = *length ys* $\Longrightarrow$
  list-all2 P (*xs*@*us*) (*ys*@*vs*) = (*list-all2 P xs ys* $\wedge$ *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* $]\!] \Longrightarrow$ *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* $\wedge$ ($\forall$ *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* $\Longrightarrow$ ($\bigwedge$*n*. *n* < *length a* $\Longrightarrow$ *P* (*a*!*n*) (*b*!*n*)) $\Longrightarrow$ *list-all2 P a b*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2I*:
  $\forall$ *x* $\in$ *set* (*zip a b*). *split P x* $\Longrightarrow$ *length a* = *length b* $\Longrightarrow$ *list-all2 P a b*
by (*simp add*: *list-all2-def*)

lemma *list-all2-nthD*:
  $[\![$ *list-all2 P xs ys*; *p* < *size xs* $]\!] \Longrightarrow$ *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*$]\!] \Longrightarrow$ *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 (λx y. P x (f y)) as bs
by (*auto simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-refl* [*intro?*]:
$(\bigwedge x.\ P\ x\ x) \Longrightarrow$ *list-all2 P xs xs*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-update-cong*:
$[\![\ i{<}size\ xs;\ list\text{-}all2\ P\ xs\ ys;\ P\ x\ y\ ]\!] \Longrightarrow list\text{-}all2\ P\ (xs[i{:=}x])\ (ys[i{:=}y])$
by (*simp add*: *list-all2-conv-all-nth nth-list-update*)

lemma *list-all2-update-cong2*:
$[\![list\text{-}all2\ P\ xs\ ys;\ P\ x\ y;\ i < length\ ys]\!] \Longrightarrow list\text{-}all2\ P\ (xs[i{:=}x])\ (ys[i{:=}y])$
by (*simp add*: *list-all2-lengthD list-all2-update-cong*)

lemma *list-all2-takeI* [*simp*,*intro?*]:
$list\text{-}all2\ P\ xs\ ys \Longrightarrow list\text{-}all2\ P\ (take\ n\ xs)\ (take\ n\ ys)$
apply (*induct xs arbitrary*: *n ys*)
 apply *simp*
apply (*clarsimp simp add*: *list-all2-Cons1*)
apply (*case-tac n*)
apply *auto*
done

lemma *list-all2-dropI* [*simp*,*intro?*]:
$list\text{-}all2\ P\ as\ bs \Longrightarrow list\text{-}all2\ P\ (drop\ n\ as)\ (drop\ n\ bs)$
apply (*induct as arbitrary*: *n bs*, *simp*)
apply (*clarsimp simp add*: *list-all2-Cons1*)
apply (*case-tac n*, *simp*, *simp*)
done

lemma *list-all2-mono* [*intro?*]:
$list\text{-}all2\ P\ xs\ ys \Longrightarrow (\bigwedge xs\ ys.\ P\ xs\ ys \Longrightarrow Q\ xs\ ys) \Longrightarrow list\text{-}all2\ Q\ xs\ ys$
apply (*induct xs arbitrary*: *ys*, *simp*)
apply (*case-tac ys*, *auto*)
done

lemma *list-all2-eq*:
$xs = ys \longleftrightarrow list\text{-}all2\ (op =)\ xs\ ys$
by (*induct xs ys rule*: *list-induct2'*) *auto*

48.1.17 *foldl* and *foldr*

lemma *foldl-append* [*simp*]:
foldl f a (xs @ ys) = foldl f (foldl f a xs) ys
by (*induct xs arbitrary*: *a*) *auto*

lemma *foldr-append*[*simp*]: *foldr f (xs @ ys) a = foldr f xs (foldr f ys a)*

by (*induct xs*) *auto*

lemma *foldr-map*: *foldr g* (*map f xs*) *a* = *foldr* (*g o f*) *xs a*
by(*induct xs*) *simp-all*

For efficient code generation: avoid intermediate list.

lemma *foldl-map*[*code unfold*]:
foldl g a (*map f xs*) = *foldl* (%*a x*. *g a* (*f x*)) *a xs*
by(*induct xs arbitrary*:*a*) *simp-all*

lemma *foldl-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f a x* = *g a x* |]
==> *foldl f a l* = *foldl g b k*
by (*induct k arbitrary*: *a b l*) *simp-all*

lemma *foldr-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f x a* = *g x a* |]
==> *foldr f l a* = *foldr g k b*
by (*induct k arbitrary*: *a b l*) *simp-all*

lemma (**in** *semigroup-add*) *foldl-assoc*:
shows *foldl op+* (*x+y*) *zs* = *x* + (*foldl op+ y zs*)
by (*induct zs arbitrary*: *y*) (*simp-all add*:*add-assoc*)

lemma (**in** *monoid-add*) *foldl-absorb0*:
shows *x* + (*foldl op+ 0 zs*) = *foldl op+ x zs*
by (*induct zs*) (*simp-all add*:*foldl-assoc*)

The "First Duality Theorem" in Bird & Wadler:

lemma *foldl-foldr1-lemma*:
foldl op + a xs = *a* + *foldr op + xs* (*0*::'*a*::*monoid-add*)
by (*induct xs arbitrary*: *a*) (*auto simp*:*add-assoc*)

corollary *foldl-foldr1*:
foldl op + 0 xs = *foldr op + xs* (*0*::'*a*::*monoid-add*)
by (*simp add*:*foldl-foldr1-lemma*)

The "Third Duality Theorem" in Bird & Wadler:

lemma *foldr-foldl*: *foldr f xs a* = *foldl* (%*x y*. *f y x*) *a* (*rev xs*)
by (*induct xs*) *auto*

lemma *foldl-foldr*: *foldl f a xs* = *foldr* (%*x y*. *f y x*) (*rev xs*) *a*
by (*simp add*: *foldr-foldl* [*of* %*x y*. *f y x rev xs*])

lemma (**in** *ab-semigroup-add*) *foldr-conv-foldl*: *foldr op + xs a* = *foldl op + a xs*
by (*induct xs*, *auto simp add*: *foldl-assoc add-commute*)

Note: $n \leq$ *foldl* (*op* +) *n ns* looks simpler, but is more difficult to use because it requires an additional transitivity step.

lemma *start-le-sum*: $(m::nat) <= n ==> m <= foldl\ (op\ +)\ n\ ns$
by (*induct ns arbitrary*: *n*) *auto*

lemma *elem-le-sum*: $(n::nat) : set\ ns ==> n <= foldl\ (op\ +)\ 0\ ns$
by (*force intro*: *start-le-sum simp add*: *in-set-conv-decomp*)

lemma *sum-eq-0-conv* [*iff*]:
$(foldl\ (op\ +)\ (m::nat)\ ns = 0) = (m = 0 \wedge (\forall n \in set\ ns.\ n = 0))$
by (*induct ns arbitrary*: *m*) *auto*

lemma *foldr-invariant*:
$[\![Q\ x\ ;\ \forall\ x\in\ set\ xs.\ P\ x;\ \forall\ x\ y.\ P\ x \wedge Q\ y \longrightarrow Q\ (f\ x\ y)\ ]\!] \Longrightarrow Q\ (foldr\ f\ xs\ x)$
by (*induct xs*, *simp-all*)

lemma *foldl-invariant*:
$[\![Q\ x\ ;\ \forall\ x\in\ set\ xs.\ P\ x;\ \forall\ x\ y.\ P\ x \wedge Q\ y \longrightarrow Q\ (f\ y\ x)\ ]\!] \Longrightarrow Q\ (foldl\ f\ x\ xs)$
by (*induct xs arbitrary*: *x*, *simp-all*)

foldl and *concat*

lemma *concat-conv-foldl*: *concat xss* = *foldl op@* [] *xss*
by (*induct xss*) (*simp-all add*:*monoid-append.foldl-absorb0*)

lemma *foldl-conv-concat*:
foldl (*op* @) *xs xxs* = *xs* @ (*concat xxs*)
by(*simp add*:*concat-conv-foldl monoid-append.foldl-absorb0*)

48.1.18 List summation: *listsum* and $\sum$

lemma *listsum-append*[*simp*]: *listsum* (*xs* @ *ys*) = *listsum xs* + *listsum ys*
by (*induct xs*) (*simp-all add*:*add-assoc*)

lemma *listsum-rev*[*simp*]:
fixes *xs* :: *'a*::*comm-monoid-add list*
shows *listsum* (*rev xs*) = *listsum xs*
by (*induct xs*) (*simp-all add*:*add-ac*)

lemma *listsum-foldr*:
listsum xs = *foldr* (*op* +) *xs 0*
by(*induct xs*) *auto*

For efficient code generation — *listsum* is not tail recursive but *foldl* is.

lemma *listsum*[*code unfold*]: *listsum xs* = *foldl* (*op* +) *0 xs*
by(*simp add*:*listsum-foldr foldl-foldr1*)

Some syntactic sugar for summing a function over a list:

syntax
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3SUM* -<--. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*$\sum$ -←-. -) [*0*, *51*, *10*] *10*)

syntax (*HTML* **output**)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*$\sum$-←-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM x<−xs. b == *CONST listsum* (*map* (%*x*. *b*) *xs*)
$\sum$*x*←*xs*. *b* == *CONST listsum* (*map* (%*x*. *b*) *xs*)

lemma *listsum-0* [*simp*]: ($\sum$*x*←*xs*. *0*) = *0*
by (*induct xs*) *simp-all*

For non-Abelian groups *xs* needs to be reversed on one side:

lemma *uminus-listsum-map*:
− *listsum* (*map f xs*) = (*listsum* (*map* (*uminus o f*) *xs*) :: *'a*::*ab-group-add*)
by(*induct xs*) *simp-all*

48.1.19 *upt*

lemma *upt-rec*[*code*]: [*i*..<*j*] = (*if i<j then i*#[*Suc i*..<*j*] *else* [])
— simp does not terminate!
by (*induct j*) *auto*

lemma *upt-conv-Nil* [*simp*]: *j* <= *i* ==> [*i*..<*j*] = []
by (*subst upt-rec*) *simp*

lemma *upt-eq-Nil-conv*[*simp*]: ([*i*..<*j*] = []) = (*j* = *0* ∨ *j* <= *i*)
by(*induct j*)*simp-all*

lemma *upt-eq-Cons-conv*:
([*i*..<*j*] = *x*#*xs*) = (*i* < *j* & *i* = *x* & [*i*+*1*..<*j*] = *xs*)
apply(*induct j arbitrary*: *x xs*)
apply *simp*
apply(*clarsimp simp add*: *append-eq-Cons-conv*)
apply *arith*
done

lemma *upt-Suc-append*: *i* <= *j* ==> [*i*..<(*Suc j*)] = [*i*..<*j*]@[*j*]
— Only needed if *upt-Suc* is deleted from the simpset.
by *simp*

lemma *upt-conv-Cons*: *i* < *j* ==> [*i*..<*j*] = *i* # [*Suc i*..<*j*]
by (*metis upt-rec*)

lemma *upt-add-eq-append*: *i*<=*j* ==> [*i*..<*j*+*k*] = [*i*..<*j*]@[*j*..<*j*+*k*]
— LOOPS as a simprule, since *j* <= *j*.
by (*induct k*) *auto*

lemma *length-upt* [*simp*]: *length* [*i*..<*j*] = *j* − *i*
by (*induct j*) (*auto simp add*: *Suc-diff-le*)

lemma *nth-upt* [*simp*]: *i* + *k* < *j* ==> [*i*..<*j*] ! *k* = *i* + *k*
apply (*induct j*)
apply (*auto simp add*: *less-Suc-eq nth-append split*: *nat-diff-split*)
done

lemma *hd-upt*[*simp*]: *i* < *j* ⟹ *hd*[*i*..<*j*] = *i*
by(*simp add*:*upt-conv-Cons*)

lemma *last-upt*[*simp*]: *i* < *j* ⟹ *last*[*i*..<*j*] = *j* − *1*
apply(*cases j*)
 apply *simp*
by(*simp add*:*upt-Suc-append*)

lemma *take-upt* [*simp*]: *i*+*m* <= *n* ==> *take m* [*i*..<*n*] = [*i*..<*i*+*m*]
apply (*induct m arbitrary*: *i*, *simp*)
apply (*subst upt-rec*)
apply (*rule sym*)
apply (*subst upt-rec*)
apply (*simp del*: *upt.simps*)
done

lemma *drop-upt*[*simp*]: *drop m* [*i*..<*j*] = [*i*+*m*..<*j*]
apply(*induct j*)
apply *auto*
done

lemma *map-Suc-upt*: *map Suc* [*m*..<*n*] = [*Suc m*..<*Suc n*]
by (*induct n*) *auto*

lemma *nth-map-upt*: *i* < *n*−*m* ==> (*map f* [*m*..<*n*]) ! *i* = *f*(*m*+*i*)
apply (*induct n m arbitrary*: *i rule*: *diff-induct*)
prefer *3* **apply** (*subst map-Suc-upt*[*symmetric*])
apply (*auto simp add*: *less-diff-conv nth-upt*)
done

lemma *nth-take-lemma*:
  k <= *length xs* ==> *k* <= *length ys* ==>
    (!!*i*. *i* < *k* −−> *xs*!*i* = *ys*!*i*) ==> *take k xs* = *take k ys*
apply (*atomize*, *induct k arbitrary*: *xs ys*)
apply (*simp-all add*: *less-Suc-eq-0-disj all-conj-distrib*, *clarify*)

Both lists must be non-empty

apply (*case-tac xs*, *simp*)
apply (*case-tac ys*, *clarify*)
 apply (*simp* (*no-asm-use*))
apply *clarify*

prenexing's needed, not miniscoping

apply (*simp* (*no-asm-use*) *add*: *all-simps* [*symmetric*] *del*: *all-simps*)

apply *blast*
done

lemma *nth-equalityI*:
[| *length xs* = *length ys*; *ALL i* < *length xs*. *xs*!*i* = *ys*!*i* |] ==> *xs* = *ys*
apply (*frule nth-take-lemma* [*OF le-refl eq-imp-le*])
apply (*simp-all add*: *take-all*)
done

lemma *map-nth*:
map (λi. *xs* ! *i*) [*0*..<*length xs*] = *xs*
by (*rule nth-equalityI*, *auto*)

lemma *list-all2-antisym*:
⟦ ($\bigwedge x\ y$. ⟦*P x y*; *Q y x*⟧ ⟹ *x* = *y*); *list-all2 P xs ys*; *list-all2 Q ys xs* ⟧
⟹ *xs* = *ys*
apply (*simp add*: *list-all2-conv-all-nth*)
apply (*rule nth-equalityI*, *blast*, *simp*)
done

lemma *take-equalityI*: (∀ *i*. *take i xs* = *take i ys*) ==> *xs* = *ys*
— The famous take-lemma.
apply (*drule-tac x* = *max* (*length xs*) (*length ys*) **in** *spec*)
apply (*simp add*: *le-max-iff-disj take-all*)
done

lemma *take-Cons'*:
take n (*x* # *xs*) = (*if n* = *0 then* [] *else x* # *take* (*n* − *1*) *xs*)
by (*cases n*) *simp-all*

lemma *drop-Cons'*:
drop n (*x* # *xs*) = (*if n* = *0 then x* # *xs else drop* (*n* − *1*) *xs*)
by (*cases n*) *simp-all*

lemma *nth-Cons'*: (*x* # *xs*)!*n* = (*if n* = *0 then x else xs*!(*n* − *1*))
by (*cases n*) *simp-all*

lemmas *take-Cons-number-of* = *take-Cons'*[*of number-of v*,*standard*]
lemmas *drop-Cons-number-of* = *drop-Cons'*[*of number-of v*,*standard*]
lemmas *nth-Cons-number-of* = *nth-Cons'*[*of - - number-of v*,*standard*]

declare *take-Cons-number-of* [*simp*]
drop-Cons-number-of [*simp*]
nth-Cons-number-of [*simp*]

48.1.20 *distinct* and *remdups*

lemma *distinct-append* [*simp*]:
distinct (*xs* @ *ys*) = (*distinct xs* ∧ *distinct ys* ∧ *set xs* ∩ *set ys* = {})
by (*induct xs*) *auto*

lemma *distinct-rev*[*simp*]: *distinct*(*rev xs*) = *distinct xs*
by(*induct xs*) *auto*

lemma *set-remdups* [*simp*]: *set* (*remdups xs*) = *set xs*
by (*induct xs*) (*auto simp add*: *insert-absorb*)

lemma *distinct-remdups* [*iff*]: *distinct* (*remdups xs*)
by (*induct xs*) *auto*

lemma *distinct-remdups-id*: *distinct xs* ==> *remdups xs* = *xs*
by (*induct xs*, *auto*)

lemma *remdups-id-iff-distinct*[*simp*]: (*remdups xs* = *xs*) = *distinct xs*
by(*metis distinct-remdups distinct-remdups-id*)

lemma *finite-distinct-list*: *finite A* ⟹ *EX xs*. *set xs* = *A* & *distinct xs*
by (*metis distinct-remdups finite-list set-remdups*)

lemma *remdups-eq-nil-iff* [*simp*]: (*remdups x* = []) = (*x* = [])
by (*induct x*, *auto*)

lemma *remdups-eq-nil-right-iff* [*simp*]: ([] = *remdups x*) = (*x* = [])
by (*induct x*, *auto*)

lemma *length-remdups-leq*[*iff*]: *length*(*remdups xs*) <= *length xs*
by (*induct xs*) *auto*

lemma *length-remdups-eq*[*iff*]:
 (*length* (*remdups xs*) = *length xs*) = (*remdups xs* = *xs*)
apply(*induct xs*)
 apply *auto*
apply(*subgoal-tac length* (*remdups xs*) <= *length xs*)
 apply *arith*
apply(*rule length-remdups-leq*)
done

lemma *distinct-map*:
 distinct(*map f xs*) = (*distinct xs* & *inj-on f* (*set xs*))
by (*induct xs*) *auto*

lemma *distinct-filter* [*simp*]: *distinct xs* ==> *distinct* (*filter P xs*)
by (*induct xs*) *auto*

lemma *distinct-upt*[*simp*]: *distinct*[*i*..<*j*]
by (*induct j*) *auto*

lemma *distinct-take*[*simp*]: *distinct xs* $\Longrightarrow$ *distinct* (*take i xs*)
apply(*induct xs arbitrary*: *i*)
 apply *simp*
apply (*case-tac i*)
 apply *simp-all*
apply(*blast dest*:*in-set-takeD*)
done

lemma *distinct-drop*[*simp*]: *distinct xs* $\Longrightarrow$ *distinct* (*drop i xs*)
apply(*induct xs arbitrary*: *i*)
 apply *simp*
apply (*case-tac i*)
 apply *simp-all*
done

lemma *distinct-list-update*:
assumes *d*: *distinct xs* **and** *a*: $a \notin$ *set xs* $- \{xs!i\}$
shows *distinct* (*xs*[*i*:=*a*])
proof (*cases* $i <$ *length xs*)
  case *True*
  with *a* **have** $a \notin$ *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 = ($\forall\, i <$ *size xs*. $\forall\, j <$ *size xs*. $i \neq j$ --> $xs!i \neq xs!j$)
apply (*induct xs*, *simp*, *simp*)
apply (*rule iffI*, *clarsimp*)
 apply (*case-tac i*)
apply (*case-tac j*, *simp*)
apply (*simp add*: *set-conv-nth*)
 apply (*case-tac j*)
apply (*clarsimp simp add*: *set-conv-nth*, *simp*)
apply (*rule conjI*)

 apply (*clarsimp simp add*: *set-conv-nth*)
 apply (*erule-tac* $x = 0$ **in** *allE*, *simp*)
 apply (*erule-tac* $x =$ *Suc i* **in** *allE*, *simp*, *clarsimp*)

apply (*erule-tac* $x = Suc\ i$ **in** *allE*, *simp*)
apply (*erule-tac* $x = Suc\ j$ **in** *allE*, *simp*)
done

lemma *nth-eq-iff-index-eq*:
$[\![$ *distinct xs*; $i <$ *length xs*; $j <$ *length xs* $]\!] \Longrightarrow (xs!i = xs!j) = (i = j)$
by(*auto simp*: *distinct-conv-nth*)

lemma *distinct-card*: *distinct xs* ==> *card* (*set xs*) = *size xs*
by (*induct xs*) *auto*

lemma *card-distinct*: *card* (*set xs*) = *size xs* ==> *distinct xs*
proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons x xs*)
  show *?case*
  proof (*cases* $x \in$ *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*) $\leq$ *length xs* **by** (*rule card-length*)
    ultimately have *False* **by** *simp*
    thus *?thesis* **..**
  qed
qed

lemma *not-distinct-decomp*: ~ *distinct ws* ==> *EX xs ys zs y*. *ws* = *xs*@[*y*]@*ys*@[*y*]@*zs*
apply (*induct n* == *length ws arbitrary*:*ws*) **apply** *simp*
apply(*case-tac ws*) **apply** *simp*
apply (*simp split*:*split-if-asm*)
apply (*metis Cons-eq-appendI eq-Nil-appendI split-list*)
done

lemma *length-remdups-concat*:
length(*remdups*(*concat xss*)) = *card*($\bigcup$*xs* $\in$ *set xss*. *set xs*)
by(*simp add*: *set-concat distinct-card*[*symmetric*])

48.1.21 *remove1*

lemma *remove1-append*:
  remove1 x (*xs* @ *ys*) =
  (*if* $x \in$ *set xs then remove1 x xs* @ *ys else xs* @ *remove1 x ys*)
by (*induct xs*) *auto*

lemma *in-set-remove1*[*simp*]:

$a \neq b \Longrightarrow a : set(remove1\ b\ xs) = (a : set\ xs)$
apply (*induct xs*)
apply *auto*
done

lemma *set-remove1-subset*: *set*(*remove1 x xs*) <= *set xs*
apply(*induct xs*)
apply *simp*
apply *simp*
apply *blast*
done

lemma *set-remove1-eq* [*simp*]: *distinct xs* ==> *set*(*remove1 x xs*) = *set xs* − {*x*}
apply(*induct xs*)
apply *simp*
apply *simp*
apply *blast*
done

lemma *length-remove1*:
length(*remove1 x xs*) = (*if x* : *set xs then length xs* − *1 else length xs*)
apply (*induct xs*)
apply (*auto dest*!:*length-pos-if-in-set*)
done

lemma *remove1-filter-not*[*simp*]:
$\neg\ P\ x \Longrightarrow remove1\ x\ (filter\ P\ xs) = filter\ P\ xs$
by(*induct xs*) *auto*

lemma *notin-set-remove1*[*simp*]: *x* ~: *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*

48.1.22 *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* < *n* ==> (*replicate n x*)!*i* = *x*
apply (*induct n arbitrary*: *i*, *simp*)
apply (*simp add*: *nth-Cons split*: *nat.split*)
done

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate*[*simp*]: *take i* (*replicate k x*) = *replicate* (*min i k*) *x*
apply (*case-tac k* ≤ *i*)
apply (*simp add*: *min-def*)
apply (*drule not-leE*)
apply (*simp add*: *min-def*)
apply (*subgoal-tac replicate k x* = *replicate i x* @ *replicate* (*k* − *i*) *x*)
apply *simp*
apply (*simp add*: *replicate-add* [*symmetric*])
done

lemma *drop-replicate*[*simp*]: *drop i* (*replicate k x*) = *replicate* (*k*−*i*) *x*
apply (*induct k arbitrary*: *i*)
apply *simp*
apply *clarsimp*
apply (*case-tac i*)
apply *simp*
apply *clarsimp*
done

lemma *set-replicate-Suc*: *set* (*replicate* (*Suc* *n*) *x*) = {*x*}
by (*induct* *n*) *auto*

lemma *set-replicate* [*simp*]: *n* ≠ *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*)

lemma *replicate-append-same*:
  replicate *i* *x* @ [*x*] = *x* # *replicate* *i* *x*
  by (*induct* *i*) *simp-all*

lemma *map-replicate-trivial*:
  map (λ*i*. *x*) [*0*..<*i*] = *replicate* *i* *x*
  by (*induct* *i*) (*simp-all* *add*: *replicate-append-same*)

48.1.23 *rotate1* and *rotate*

lemma *rotate-simps*[*simp*]: *rotate1* [] = [] ∧ *rotate1* (*x*#*xs*) = *xs* @ [*x*]
by(*simp* *add*:*rotate1-def*)

lemma *rotate0*[*simp*]: *rotate* *0* = *id*
by(*simp* *add*:*rotate-def*)

lemma *rotate-Suc*[*simp*]: *rotate* (*Suc* *n*) *xs* = *rotate1*(*rotate* *n* *xs*)
by(*simp* *add*:*rotate-def*)

lemma *rotate-add*:
  rotate (*m*+*n*) = *rotate* *m* *o* *rotate* *n*
by(*simp* *add*:*rotate-def* *funpow-add*)

lemma *rotate-rotate*: *rotate* *m* (*rotate* *n* *xs*) = *rotate* (*m*+*n*) *xs*
by(*simp* *add*:*rotate-add*)

lemma *rotate1-rotate-swap*: *rotate1* (*rotate* *n* *xs*) = *rotate* *n* (*rotate1* *xs*)
by(*simp* *add*:*rotate-def* *funpow-swap1*)

lemma *rotate1-length01*[*simp*]: *length* *xs* <= *1* ⟹ *rotate1* *xs* = *xs*
by(*cases* *xs*) *simp-all*

lemma *rotate-length01*[*simp*]: *length* *xs* <= *1* ⟹ *rotate* *n* *xs* = *xs*
apply(*induct* *n*)
 apply *simp*
apply (*simp* *add*:*rotate-def*)

done

lemma *rotate1-hd-tl*: $xs \neq [] \Longrightarrow rotate1\ xs = tl\ xs\ @\ [hd\ xs]$
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*]: $n\ mod\ length\ xs = 0 \Longrightarrow rotate\ n\ xs = xs$
by(*simp add*:*rotate-drop-take*)

lemma *length-rotate1*[*simp*]: $length(rotate1\ xs) = length\ xs$
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *length-rotate*[*simp*]: $length(rotate\ n\ xs) = length\ xs$
by (*induct n arbitrary*: *xs*) (*simp-all add*:*rotate-def*)

lemma *distinct1-rotate*[*simp*]: $distinct(rotate1\ xs) = distinct\ xs$
by(*simp add*:*rotate1-def split*:*list.split*) *blast*

lemma *distinct-rotate*[*simp*]: $distinct(rotate\ n\ xs) = distinct\ xs$
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate-map*: $rotate\ n\ (map\ f\ xs) = map\ f\ (rotate\ n\ xs)$
by(*simp add*:*rotate-drop-take take-map drop-map*)

lemma *set-rotate1*[*simp*]: $set(rotate1\ xs) = set\ xs$
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *set-rotate*[*simp*]: $set(rotate\ n\ xs) = set\ xs$
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate1-is-Nil-conv*[*simp*]: $(rotate1\ xs = []) = (xs = [])$
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *rotate-is-Nil-conv*[*simp*]: (*rotate n xs* = []) = (*xs* = [])
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate-rev*:
rotate n (*rev xs*) = *rev*(*rotate* (*length xs* − (*n mod length xs*)) *xs*)
apply(*simp add*:*rotate-drop-take rev-drop rev-take*)
apply(*cases length xs* = *0*)
apply *simp*
apply(*cases n mod length xs* = *0*)
apply *simp*
apply(*simp add*:*rotate-drop-take rev-drop rev-take*)
done

lemma *hd-rotate-conv-nth*: *xs* ≠ [] ⟹ *hd*(*rotate n xs*) = *xs*!(*n mod length xs*)
apply(*simp add*:*rotate-drop-take hd-append hd-drop-conv-nth hd-conv-nth*)
apply(*subgoal-tac length xs* ≠ *0*)
prefer *2* **apply** *simp*
using *mod-less-divisor*[*of length xs n*] **by** *arith*

48.1.24 *sublist* — a generalization of *nth* to sets

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

lemma *sublist-nil* [*simp*]: *sublist* [] *A* = []
by (*auto simp add*: *sublist-def*)

lemma *length-sublist*:
length(*sublist xs I*) = *card*{*i*. *i* < *length xs* ∧ *i* : *I*}
by(*simp add*: *sublist-def length-filter-conv-card cong*:*conj-cong*)

lemma *sublist-shift-lemma-Suc*:
map fst (*filter* (%*p*. *P*(*Suc*(*snd p*))) (*zip xs is*)) =
map fst (*filter* (%*p*. *P*(*snd p*)) (*zip xs* (*map Suc is*)))
apply(*induct xs arbitrary*: *is*)
apply *simp*
apply (*case-tac is*)
apply *simp*
apply *simp*
done

lemma *sublist-shift-lemma*:
map fst [*p*<−*zip xs* [*i*..<*i* + *length xs*] . *snd p* : *A*] =
map fst [*p*<−*zip xs* [*0*..<*length xs*] . *snd p* + *i* : *A*]
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *add-commute*)

lemma *sublist-append*:
sublist (*l* @ *l′*) *A* = *sublist l A* @ *sublist l′* {*j*. *j* + *length l* : *A*}
apply (*unfold sublist-def*)

apply (*induct* l' *rule*: *rev-induct*, *simp*)
apply (*simp add*: *upt-add-eq-append*[*of 0*] *zip-append sublist-shift-lemma*)
apply (*simp add*: *add-commute*)
done

lemma *sublist-Cons*:
sublist (x # l) A = (*if* 0:A *then* [x] *else* []) @ *sublist* l {j. *Suc* j : A}
apply (*induct* l *rule*: *rev-induct*)
apply (*simp add*: *sublist-def*)
apply (*simp del*: *append-Cons add*: *append-Cons*[*symmetric*] *sublist-append*)
done

lemma *set-sublist*: *set*(*sublist xs* I) = {*xs*!i|i. i<*size xs* $\wedge$ $i \in I$}
apply(*induct xs arbitrary*: I)
apply(*auto simp*: *sublist-Cons nth-Cons split*:*nat.split dest*!: *gr0-implies-Suc*)
done

lemma *set-sublist-subset*: *set*(*sublist xs* I) $\subseteq$ *set xs*
by(*auto simp add*:*set-sublist*)

lemma *notin-set-sublistI*[*simp*]: $x \notin$ *set xs* $\Longrightarrow$ $x \notin$ *set*(*sublist xs* I)
by(*auto simp add*:*set-sublist*)

lemma *in-set-sublistD*: $x \in$ *set*(*sublist xs* I) $\Longrightarrow$ $x \in$ *set xs*
by(*auto simp add*:*set-sublist*)

lemma *sublist-singleton* [*simp*]: *sublist* [x] A = (*if* 0 : A *then* [x] *else* [])
by (*simp add*: *sublist-Cons*)

lemma *distinct-sublistI*[*simp*]: *distinct xs* $\Longrightarrow$ *distinct*(*sublist xs* I)
apply(*induct xs arbitrary*: I)
apply *simp*
apply(*auto simp add*:*sublist-Cons*)
done

lemma *sublist-upt-eq-take* [*simp*]: *sublist* l {..<n} = *take* n l
apply (*induct* l *rule*: *rev-induct*, *simp*)
apply (*simp split*: *nat-diff-split add*: *sublist-append*)
done

lemma *filter-in-sublist*:
distinct xs $\Longrightarrow$ *filter* (%x. $x \in$ *set*(*sublist xs s*)) *xs* = *sublist xs s*
proof (*induct xs arbitrary*: *s*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a xs*)
moreover hence !x. x: *set xs* $\longrightarrow$ $x \neq a$ **by** *auto*

ultimately show *?case* **by**(*simp add*: *sublist-Cons cong*:*filter-cong*)
qed

48.1.25 *splice*

lemma *splice-Nil2* [*simp*, *code*]:
splice xs [] = *xs*
by (*cases xs*) *simp-all*

lemma *splice-Cons-Cons* [*simp*, *code*]:
splice (*x*#*xs*) (*y*#*ys*) = *x* # *y* # *splice xs ys*
by *simp*

declare *splice.simps*(*2*) [*simp del*, *code del*]

lemma *length-splice*[*simp*]: *length*(*splice xs ys*) = *length xs* + *length ys*
apply(*induct xs arbitrary*: *ys*) **apply** *simp*
apply(*case-tac ys*)
apply *auto*
done

48.2 Sorting

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not yet available. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

context *linorder*
begin

lemma *sorted-Cons*: *sorted* (*x*#*xs*) = (*sorted xs* & (*ALL y*:*set xs*. *x* <= *y*))
apply(*induct xs arbitrary*: *x*) **apply** *simp*
by *simp* (*blast intro*: *order-trans*)

lemma *sorted-append*:
sorted (*xs*@*ys*) = (*sorted xs* & *sorted ys* & ($\forall x \in$ *set xs*. $\forall y \in$ *set ys*. $x \le y$))
by (*induct xs*) (*auto simp add*:*sorted-Cons*)

lemma *set-insort*: *set*(*insort x xs*) = *insert x* (*set xs*)
by (*induct xs*) *auto*

lemma *set-sort*[*simp*]: *set*(*sort xs*) = *set xs*
by (*induct xs*) (*simp-all add*:*set-insort*)

lemma *distinct-insort*: *distinct* (*insort x xs*) = ($x \notin$ *set xs* $\wedge$ *distinct xs*)
by(*induct xs*)(*auto simp*:*set-insort*)

lemma *distinct-sort*[*simp*]: *distinct* (*sort xs*) = *distinct xs*
by(*induct xs*)(*simp-all add*:*distinct-insort set-sort*)

lemma *sorted-insort*: *sorted* (*insort x xs*) = *sorted xs*
apply (*induct xs*)
apply(*auto simp*:*sorted-Cons set-insort*)
done

theorem *sorted-sort*[*simp*]: *sorted* (*sort xs*)
by (*induct xs*) (*auto simp*:*sorted-insort*)

lemma *sorted-distinct-set-unique*:
assumes *sorted xs distinct xs sorted ys distinct ys set xs* = *set ys*
shows *xs* = *ys*
proof −
  from *assms* **have** *1*: *length xs* = *length ys* **by** (*metis distinct-card*)
  from *assms* **show** *?thesis*
  proof(*induct rule*:*list-induct2*[*OF 1*])
    case *1* **show** *?case* **by** *simp*
  next
    case *2* **thus** *?case* **by** (*simp add*:*sorted-Cons*)
      (*metis Diff-insert-absorb antisym insertE insert-iff*)
  qed
qed

lemma *finite-sorted-distinct-unique*:
shows *finite A* ⟹ *EX*! *xs*. *set xs* = *A* & *sorted xs* & *distinct xs*
apply(*drule finite-distinct-list*)
apply *clarify*
apply(*rule-tac a*=*sort xs* **in** *ex1I*)
apply (*auto simp*: *sorted-distinct-set-unique*)
done

end

lemma *sorted-upt*[*simp*]: *sorted*[*i*..<*j*]
by (*induct j*) (*simp-all add*:*sorted-append*)

48.2.1 *sorted-list-of-set*

This function maps (finite) linearly ordered sets to sorted lists. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via *set*).

context *linorder*
begin

definition
sorted-list-of-set :: *'a set* ⇒ *'a list* **where**
sorted-list-of-set A == *THE xs*. *set xs* = *A* & *sorted xs* & *distinct xs*

lemma *sorted-list-of-set*[*simp*]: *finite A* $\Longrightarrow$
  set(*sorted-list-of-set A*) = *A* &
  sorted(*sorted-list-of-set A*) & *distinct*(*sorted-list-of-set A*)
apply(*simp add*:*sorted-list-of-set-def*)
apply(*rule the1I2*)
 apply(*simp-all add*: *finite-sorted-distinct-unique*)
done

lemma *sorted-list-of-empty*[*simp*]: *sorted-list-of-set* {} = []
unfolding *sorted-list-of-set-def*
apply(*subst the-equality*[*of* - []])
apply *simp-all*
done

end

48.2.2 *upto*: the generic interval-list

class *finite-intvl-succ* = *linorder* +
fixes *successor* :: $'a \Rightarrow 'a$
assumes *finite-intvl*: *finite*{*a..b*}
and *successor-incr*: $a <$ *successor a*
and *ord-discrete*: $\neg(\exists\, x.\ a < x$ & $x <$ *successor a*)

context *finite-intvl-succ*
begin

definition
 upto :: $'a \Rightarrow 'a \Rightarrow 'a$ *list* ((*1*[-../-])) **where**
upto i j == *sorted-list-of-set* {*i..j*}

lemma *upto*[*simp*]: *set*[*a..b*] = {*a..b*} & *sorted*[*a..b*] & *distinct*[*a..b*]
by(*simp add*:*upto-def finite-intvl*)

lemma *insert-intvl*: $i \le j \Longrightarrow$ *insert i* {*successor i..j*} = {*i..j*}
apply(*insert successor-incr*[*of i*])
apply(*auto simp*: *atLeastAtMost-def atLeast-def atMost-def*)
apply (*metis ord-discrete less-le not-le*)
done

lemma *sorted-list-of-set-rec*: $i \le j \Longrightarrow$
  sorted-list-of-set {*i..j*} = *i* # *sorted-list-of-set* {*successor i..j*}
apply(*simp add*:*sorted-list-of-set-def upto-def*)
apply (*rule the1-equality*[*OF finite-sorted-distinct-unique*])
 apply (*simp add*:*finite-intvl*)
apply(*rule the1I2*[*OF finite-sorted-distinct-unique*])
 apply (*simp add*:*finite-intvl*)
apply (*simp add*: *sorted-Cons insert-intvl Ball-def*)

apply (*metis successor-incr leD less-imp-le order-trans*)
done

lemma *upto-rec*[*code*]: [*i..j*] = (*if* $i \leq j$ *then* *i* # [*successor i..j*] *else* [])
by(*simp add*: *upto-def sorted-list-of-set-rec*)

end

The integers are an instance of the above class:

instance *int*:: *finite-intvl-succ*
successor-int-def: *successor* == (%*i*. *i*+*1*)
by *intro-classes* (*simp-all add*: *successor-int-def*)

Now [*i..j*] is defined for integers.

hide (**open**) *const successor*

48.2.3 *lists*: the list-forming operator over sets

inductive-set
lists :: *'a set* => *'a list set*
for *A* :: *'a set*
where
Nil [*intro*!]: []: *lists A*
| *Cons* [*intro*!,*noatp*]: [| *a*: *A*;*l*: *lists A*|] ==> *a*#*l* : *lists A*

inductive-cases *listsE* [*elim*!,*noatp*]: *x*#*l* : *lists A*
inductive-cases *listspE* [*elim*!,*noatp*]: *listsp A* (*x* # *l*)

lemma *listsp-mono* [*mono*]: $A \leq B$ ==> *listsp A* $\leq$ *listsp B*
by (*clarify*, *erule listsp.induct*, *blast*+)

lemmas *lists-mono* = *listsp-mono* [*to-set*]

lemma *listsp-infI*:
assumes *l*: *listsp A l* **shows** *listsp B l* ==> *listsp* (*inf A B*) *l* **using** *l*
by *induct blast*+

lemmas *lists-IntI* = *listsp-infI* [*to-set*]

lemma *listsp-inf-eq* [*simp*]: *listsp* (*inf A B*) = *inf* (*listsp A*) (*listsp B*)
proof (*rule mono-inf* [**where** *f*=*listsp*, *THEN order-antisym*])
show *mono listsp* **by** (*simp add*: *mono-def listsp-mono*)
show *inf* (*listsp A*) (*listsp B*) $\leq$ *listsp* (*inf A B*) **by** (*blast intro*: *listsp-infI*)
qed

lemmas *listsp-conj-eq* [*simp*] = *listsp-inf-eq* [*simplified inf-fun-eq inf-bool-eq*]

lemmas *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set*]

lemma *append-in-listsp-conv* [*iff*]:
  (*listsp A* (*xs* @ *ys*)) = (*listsp A xs* ∧ *listsp A ys*)
by (*induct xs*) *auto*

lemmas *append-in-lists-conv* [*iff*] = *append-in-listsp-conv* [*to-set*]

lemma *in-listsp-conv-set*: (*listsp A xs*) = (∀ *x* ∈ *set xs*. *A x*)
— eliminate *listsp* in favour of *set*
by (*induct xs*) *auto*

lemmas *in-lists-conv-set* = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest*!,*noatp*]: *listsp A xs* ==> ∀ *x*∈*set xs*. *A x*
by (*rule in-listsp-conv-set* [*THEN iffD1*])

lemmas *in-listsD* [*dest*!,*noatp*] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro*!,*noatp*]: ∀ *x*∈*set xs*. *A x* ==> *listsp A xs*
by (*rule in-listsp-conv-set* [*THEN iffD2*])

lemmas *in-listsI* [*intro*!,*noatp*] = *in-listspI* [*to-set*]

lemma *lists-UNIV* [*simp*]: *lists UNIV* = *UNIV*
by *auto*

48.2.4 Inductive definition for membership

inductive *ListMem* :: ′*a* ⇒ ′*a list* ⇒ *bool*
where
  elem: *ListMem x* (*x* # *xs*)
 | *insert*: *ListMem x xs* ⟹ *ListMem x* (*y* # *xs*)

lemma *ListMem-iff*: (*ListMem x xs*) = (*x* ∈ *set xs*)
apply (*rule iffI*)
 apply (*induct set*: *ListMem*)
  apply *auto*
apply (*induct xs*)
 apply (*auto intro*: *ListMem.intros*)
done

48.2.5 Lists as Cartesian products

set-Cons A Xs: the set of lists with head drawn from *A* and tail drawn from *Xs*.

constdefs
 set-Cons :: ′*a set* ⇒ ′*a list set* ⇒ ′*a list set*
 set-Cons A XS == {*z*. ∃ *x xs*. *z* = *x*#*xs* & *x* ∈ *A* & *xs* ∈ *XS*}

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*)

48.3 Relations on Lists

48.3.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

consts *lexn* :: (*'a* ∗ *'a*)*set* => *nat* => (*'a list* ∗ *'a list*)*set*
— The lexicographic ordering for lists of the specified length
primrec
lexn r 0 = {}
lexn r (*Suc n*) =
(*prod-fun* (%(*x*,*xs*). *x*#*xs*) (%(*x*,*xs*). *x*#*xs*) ‘ (*r* <∗*lex*∗> *lexn r n*)) *Int*
{(*xs*,*ys*). *length xs* = *Suc n* ∧ *length ys* = *Suc n*}

constdefs
lex :: (*'a* × *'a*) *set* => (*'a list* × *'a list*) *set*
lex r == ⋃*n*. *lexn r n*
— Holds only between lists of the same length

lenlex :: (*'a* × *'a*) *set* => (*'a list* × *'a list*) *set*
lenlex r == *inv-image* (*less-than* <∗*lex*∗> *lex r*) (%*xs*. (*length xs*, *xs*))
— Compares lists by their length and then lexicographically

lemma *wf-lexn*: *wf r* ==> *wf* (*lexn r n*)
apply (*induct n*, *simp*, *simp*)
apply(*rule wf-subset*)
prefer *2* **apply** (*rule Int-lower1*)
apply(*rule wf-prod-fun-image*)
prefer *2* **apply** (*rule inj-onI*, *auto*)
done

lemma *lexn-length*:
(*xs*, *ys*) : *lexn r n* ==> *length xs* = *n* ∧ *length ys* = *n*
by (*induct n arbitrary*: *xs ys*) *auto*

lemma *wf-lex* [*intro*!]: *wf r* ==> *wf* (*lex r*)
apply (*unfold lex-def*)
apply (*rule wf-UN*)
apply (*blast intro*: *wf-lexn*, *clarify*)

apply (*rename-tac m n*)
apply (*subgoal-tac m* $\neq$ *n*)
prefer *2* **apply** *blast*
apply (*blast dest*: *lexn-length not-sym*)
done

lemma *lexn-conv*:
lexn r n =
{(*xs*,*ys*). *length xs* = *n* ∧ *length ys* = *n* ∧
(∃ *xys x y xs′ ys′*. *xs*= *xys* @ *x*#*xs′* ∧ *ys*= *xys* @ *y* # *ys′* ∧ (*x*, *y*):*r*)}
apply (*induct n*, *simp*)
apply (*simp add*: *image-Collect lex-prod-def*, *safe*, *blast*)
apply (*rule-tac x* = *ab* # *xys* **in** *exI*, *simp*)
apply (*case-tac xys*, *simp-all*, *blast*)
done

lemma *lex-conv*:
lex r =
{(*xs*,*ys*). *length xs* = *length ys* ∧
(∃ *xys x y xs′ ys′*. *xs* = *xys* @ *x* # *xs′* ∧ *ys* = *xys* @ *y* # *ys′* ∧ (*x*, *y*):*r*)}
by (*force simp add*: *lex-def lexn-conv*)

lemma *wf-lenlex* [*intro*!]: *wf r* ==> *wf* (*lenlex r*)
by (*unfold lenlex-def*) *blast*

lemma *lenlex-conv*:
lenlex r = {(*xs*,*ys*). *length xs* < *length ys* |
length xs = *length ys* ∧ (*xs*, *ys*) : *lex r*}
by (*simp add*: *lenlex-def diag-def lex-prod-def inv-image-def*)

lemma *Nil-notin-lex* [*iff*]: ([], *ys*) ∉ *lex r*
by (*simp add*: *lex-conv*)

lemma *Nil2-notin-lex* [*iff*]: (*xs*, []) ∉ *lex r*
by (*simp add*:*lex-conv*)

lemma *Cons-in-lex* [*simp*]:
((*x* # *xs*, *y* # *ys*) : *lex r*) =
((*x*, *y*) : *r* ∧ *length xs* = *length ys* | *x* = *y* ∧ (*xs*, *ys*) : *lex r*)
apply (*simp add*: *lex-conv*)
apply (*rule iffI*)
prefer *2* **apply** (*blast intro*: *Cons-eq-appendI*, *clarify*)
apply (*case-tac xys*, *simp*, *simp*)
apply *blast*
done

48.3.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" ¡ "ab" ¡ "b". This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

constdefs
$lexord$ :: $('a * 'a)set \Rightarrow ('a\ list * 'a\ list)\ set$
$lexord\ \ r == \{(x,y).\ \exists\ \ a\ v.\ y = x\ @\ a\ \#\ v \lor$
$(\exists\ \ u\ a\ b\ v\ w.\ (a,b) \in r \land x = u\ @\ (a\ \#\ v) \land y = u\ @\ (b\ \#\ w))\}$

lemma *lexord-Nil-left*[*simp*]: $([],y) \in lexord\ r = (\exists\ \ a\ x.\ y = a\ \#\ x)$
by (*unfold lexord-def*, *induct-tac y*, *auto*)

lemma *lexord-Nil-right*[*simp*]: $(x,[]) \notin lexord\ r$
by (*unfold lexord-def*, *induct-tac x*, *auto*)

lemma *lexord-cons-cons*[*simp*]:
$((a\ \#\ x,\ b\ \#\ y) \in lexord\ r) = ((a,b)\in r \mid (a = b\ \&\ (x,y)\in lexord\ r))$
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*: $\exists\ \ b\ z.\ y = b\ \#\ z \Longrightarrow (x,\ x\ @\ y) \in lexord\ r$
by (*induct-tac x*, *auto*)

lemma *lexord-append-left-rightI*:
$(a,b) \in r \Longrightarrow (u\ @\ a\ \#\ x,\ u\ @\ b\ \#\ y) \in lexord\ r$
by (*induct-tac u*, *auto*)

lemma *lexord-append-leftI*: $(u,v) \in lexord\ r \Longrightarrow (x\ @\ u,\ x\ @\ v) \in lexord\ r$
by (*induct x*, *auto*)

lemma *lexord-append-leftD*:
$[\![\ (x\ @\ u,\ x\ @\ v) \in lexord\ r;\ (!\ a.\ (a,a) \notin r)\ ]\!] \Longrightarrow (u,v) \in lexord\ r$
by (*erule rev-mp*, *induct-tac x*, *auto*)

lemma *lexord-take-index-conv*:
$((x,y) : lexord\ r) =$
$((length\ x < length\ y \land take\ (length\ x)\ y = x) \lor$
$(\exists i.\ i < min(length\ x)(length\ y)\ \&\ take\ i\ x = take\ i\ y\ \&\ (x!i,y!i) \in r))$
apply (*unfold lexord-def Let-def*, *clarsimp*)
apply (*rule-tac* $f = (\%\ a\ b.\ a \lor b)$ **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*: $(x,y) \in lex\ r = ((x,y) \in lexord\ r \wedge length\ x = length\ y)$
apply (*rule-tac x* = *y* **in** *spec*)
apply (*induct-tac x*, *clarsimp*)
by (*clarify*, *case-tac x*, *simp*, *force*)

lemma *lexord-irreflexive*: $(!\ x.\ (x,x) \notin r) \Longrightarrow (y,y) \notin lexord\ r$
by (*induct y*, *auto*)

lemma *lexord-trans*:
$[\![(x, y) \in lexord\ r; (y, z) \in lexord\ r; trans\ r]\!] \Longrightarrow (x, z) \in lexord\ r$
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*: $trans\ r \Longrightarrow trans\ (lexord\ r)$
by (*rule transI*, *drule lexord-trans*, *blast*)

lemma *lexord-linear*: $(!\ a\ b.\ (a,b)\in r \mid a = b \mid (b,a) \in r) \Longrightarrow (x,y) : lexord\ r \mid x = y \mid (y,x) : lexord\ r$
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*

48.4 Lexicographic combination of measure functions

These are useful for termination proofs

definition
measures fs = *inv-image* (*lex less-than*) (%*a*. *map* (%*f*. *f a*) *fs*)

lemma *wf-measures*[*recdef-wf*, *simp*]: *wf* (*measures fs*)
unfolding *measures-def*
by *blast*

lemma *in-measures*[*simp*]:
$(x, y) \in measures\ [] = False$
$(x, y) \in measures\ (f\ \#\ fs)$
$= (f\ x < f\ y \lor (f\ x = f\ y \land (x, y) \in measures\ fs))$
unfolding *measures-def*
by *auto*

lemma *measures-less*: $f\ x < f\ y ==> (x, y) \in measures\ (f\#fs)$
by *simp*

lemma *measures-lesseq*: $f\ x <= f\ y ==> (x, y) \in measures\ fs ==> (x, y) \in measures\ (f\#fs)$
by *auto*

48.4.1 Lifting a Relation on List Elements to the Lists

inductive-set
listrel :: $('a * 'a)set => ('a\ list * 'a\ list)set$
for r :: $('a * 'a)set$
where
Nil: $([],[]) \in listrel\ r$
| *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*)

48.5 Miscellany

48.5.1 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* (-)

setup *StringSyntax.setup*

48.6 Code generator

48.6.1 Setup

types-code
list (- *list*)
attach (*term-of*) ⟪
fun term-of-list f T = *HOLogic.mk-list T o map f*;
⟫
attach (*test*) ⟪
fun gen-list′ aG 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 term-of-char = *HOLogic.mk-char o ord*;
⟫
attach (*test*) ⟪
fun gen-char i = *chr* (*random-range* (*ord a*) (*Int.min* (*ord a* + *i*, *ord z*)));
⟫

consts-code *Cons* ((- ::/ -))

code-type *list*
(*SML* - *list*)
(*OCaml* - *list*)
(*Haskell* ![-])

code-reserved *SML*
list

code-reserved *OCaml*
list

code-const *Nil*
(*SML* [])
(*OCaml* [])
(*Haskell* [])

setup ⟪
fold (*fn target* => *CodeTarget.add-pretty-list target*
@{*const-name Nil*} @{*const-name Cons*}
) [*SML*, *OCaml*, *Haskell*]
⟫

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

code-const *op* = :: ′*a*::*eq list* ⇒ ′*a list* ⇒ *bool*

```
  (Haskell infixl 4 ==)

setup ⟪
let

fun list-codegen thy defs gr dep thyname b t =
  let
    val ts = HOLogic.dest-list t;
    val (gr', -) = Codegen.invoke-tycodegen thy defs dep thyname false
      (gr, fastype-of t);
    val (gr'', ps) = foldl-map
      (Codegen.invoke-codegen thy defs dep thyname false) (gr', ts)
  in SOME (gr'', Pretty.list [ ] ps) end handle TERM - => NONE;

fun char-codegen thy defs gr dep thyname b t =
  let
    val i = HOLogic.dest-char t;
    val (gr', -) = Codegen.invoke-tycodegen thy defs dep thyname false
      (gr, fastype-of t)
  in SOME (gr', Pretty.str (ML-Syntax.print-string (chr i)))
  end handle TERM - => NONE;

in
  Codegen.add-codegen list-codegen list-codegen
  #> Codegen.add-codegen char-codegen char-codegen
end;
⟫
```

48.6.2 Generation of efficient code

```
consts
  null:: 'a list ⇒ bool
  list-inter :: 'a list ⇒ 'a list ⇒ 'a list
  list-ex :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
  list-all :: ('a ⇒ bool) ⇒ ('a list ⇒ bool)
  filtermap :: ('a ⇒ 'b option) ⇒ 'a list ⇒ 'b list
  map-filter :: ('a ⇒ 'b) ⇒ ('a ⇒ bool) ⇒ 'a list ⇒ 'b list

setup ⟪ snd o Sign.declare-const [] (*authentic syntax*)
  (member, @{typ 'a ⇒ 'a list ⇒ bool}, InfixlName (mem, 55)) ⟫
primrec
  x mem [] = False
  x mem (y#ys) = (if y=x then True else x mem ys)

primrec
  null [] = True
  null (x#xs) = False

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)
```

Only use *mem* for generating executable code. Otherwise use $x \in$ *set xs* instead — it is much easier to reason about. The same is true for *list-all* and *list-ex*: write $\forall x{\in}$*set xs* and $\exists x{\in}$*set xs* instead because the HOL quantifiers are aleady known to the automatic provers. In fact, the declarations in the code subsection make sure that $\in$, $\forall x{\in}$*set xs* and $\exists x{\in}$*set xs* are implemented efficiently.

Efficient emptyness check is implemented by *null*.

The functions *filtermap* and *map-filter* are just there to generate efficient code. Do not use them for modelling and proving.

```
lemma rev-foldl-cons [code]:
  rev xs = foldl (λxs x. x # xs) [] xs
proof (induct xs)
  case Nil then show ?case by simp
next
  case Cons
  {
    fix x xs ys
    have foldl (λxs x. x # xs) ys xs @ [x]
      = foldl (λxs x. x # xs) (ys @ [x]) xs
    by (induct xs arbitrary: ys) auto
  }
  note aux = this
  show ?case by (induct xs) (auto simp add: Cons aux)
qed
```

lemma *mem-iff* [*code post*]:
$x\ mem\ xs \longleftrightarrow x \in set\ xs$
by (*induct xs*) *auto*

lemmas *in-set-code* [*code unfold*] = *mem-iff* [*symmetric*]

lemma *empty-null* [*code inline*]:
$xs = [] \longleftrightarrow null\ xs$
by (*cases xs*) *simp-all*

lemmas *null-empty* [*code post*] =
empty-null [*symmetric*]

lemma *list-inter-conv*:
$set\ (list\text{-}inter\ xs\ ys) = set\ xs \cap set\ ys$
by (*induct xs*) *auto*

lemma *list-all-iff* [*code post*]:
$list\text{-}all\ P\ xs \longleftrightarrow (\forall x \in set\ xs.\ P\ x)$
by (*induct xs*) *auto*

lemmas *list-ball-code* [*code unfold*] = *list-all-iff* [*symmetric*]

lemma *list-all-append* [*simp*]:
$list\text{-}all\ P\ (xs\ @\ ys) \longleftrightarrow (list\text{-}all\ P\ xs \wedge list\text{-}all\ P\ ys)$
by (*induct xs*) *auto*

lemma *list-all-rev* [*simp*]:
$list\text{-}all\ P\ (rev\ xs) \longleftrightarrow list\text{-}all\ P\ xs$
by (*simp add*: *list-all-iff*)

lemma *list-all-length*:
$list\text{-}all\ P\ xs \longleftrightarrow (\forall n < length\ xs.\ P\ (xs\ !\ n))$
unfolding *list-all-iff* **by** (*auto intro*: *all-nth-imp-all-set*)

lemma *list-ex-iff* [*code post*]:
$list\text{-}ex\ P\ xs \longleftrightarrow (\exists x \in set\ xs.\ P\ x)$
by (*induct xs*) *simp-all*

lemmas *list-bex-code* [*code unfold*] =
list-ex-iff [*symmetric*]

lemma *list-ex-length*:
$list\text{-}ex\ P\ xs \longleftrightarrow (\exists n < length\ xs.\ P\ (xs\ !\ n))$
unfolding *list-ex-iff set-conv-nth* **by** *auto*

lemma *filtermap-conv*:
$filtermap\ f\ xs = map\ (\lambda x.\ the\ (f\ x))\ (filter\ (\lambda x.\ f\ x \neq None)\ xs)$
by (*induct xs*) (*simp-all split*: *option.split*)

lemma *map-filter-conv* [*simp*]:
  map-filter f P xs = *map f* (*filter P xs*)
by (*induct xs*) *auto*

Code for bounded quantification and summation over nats.

lemma *atMost-upto* [*code unfold*]:
  $\{..n\}$ = *set* $[0..<Suc\ n]$
by *auto*

lemma *atLeast-upt* [*code unfold*]:
  $\{..<n\}$ = *set* $[0..<n]$
by *auto*

lemma *greaterThanLessThan-upt* [*code unfold*]:
  $\{n<..<m\}$ = *set* $[Suc\ n..<m]$
by *auto*

lemma *atLeastLessThan-upt* [*code unfold*]:
  $\{n..<m\}$ = *set* $[n..<m]$
by *auto*

lemma *greaterThanAtMost-upto* [*code unfold*]:
  $\{n<..m\}$ = *set* $[Suc\ n..<Suc\ m]$
by *auto*

lemma *atLeastAtMost-upto* [*code unfold*]:
  $\{n..m\}$ = *set* $[n..<Suc\ m]$
by *auto*

lemma *all-nat-less-eq* [*code unfold*]:
  $(\forall\, m<n::nat.\ P\ m) \longleftrightarrow (\forall\, m \in \{0..<n\}.\ P\ m)$
by *auto*

lemma *ex-nat-less-eq* [*code unfold*]:
  $(\exists\, m<n::nat.\ P\ m) \longleftrightarrow (\exists\, m \in \{0..<n\}.\ P\ m)$
by *auto*

lemma *all-nat-less* [*code unfold*]:
  $(\forall\, m\leq n::nat.\ P\ m) \longleftrightarrow (\forall\, m \in \{0..n\}.\ P\ m)$
by *auto*

lemma *ex-nat-less* [*code unfold*]:
  $(\exists\, m\leq n::nat.\ P\ m) \longleftrightarrow (\exists\, m \in \{0..n\}.\ P\ m)$
by *auto*

lemma *setsum-set-upt-conv-listsum*[*code unfold*]:
  setsum f (*set*$[k..<n]$) = *listsum* (*map f* $[k..<n]$)
apply(*subst atLeastLessThan-upt*[*symmetric*])

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

48.6.3 List partitioning

consts
  partition :: ($'a \Rightarrow bool$) $\Rightarrow 'a$ *list* $\Rightarrow$ $'a$ *list* $\times$ $'a$ *list*
primrec
  partition P [] = ([], [])
  partition P (*x* # *xs*) =
    (*let* (*yes*, *no*) = *partition P xs*
    in if P x then (*x* # *yes*, *no*) *else* (*yes*, *x* # *no*))

lemma *partition-P*:
  partition P xs = (*yes*, *no*) $\Longrightarrow$ ($\forall$ *p*$\in$ *set yes*. *P p*) $\wedge$ ($\forall$ *p*$\in$ *set no*. $\neg$ *P p*)
proof (*induct xs arbitrary*: *yes no rule*: *partition.induct*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons a as*)
  let *?p* = *partition P as*
  let *?p'* = *partition P* (*a* # *as*)
  note *prem* = ‹*?p'* = (*yes*, *no*)›
  show *?case*
  proof (*cases P a*)
    case *True*
    with *prem* **have** *yes*: *yes* = *a* # *fst ?p* **and** *no*: *no* = *snd ?p*
      by (*simp-all add*: *Let-def split-def*)
    have ($\forall$ *p*$\in$ *set* (*fst ?p*). *P p*) $\wedge$ ($\forall$ *p*$\in$ *set no*. $\neg$ *P p*)
      by (*rule Cons.hyps*) (*simp add*: *yes no*)
    with *True yes* **show** *?thesis* **by** *simp*
  next
    case *False*
    with *prem* **have** *yes*: *yes* = *fst ?p* **and** *no*: *no* = *a* # *snd ?p*
      by (*simp-all add*: *Let-def split-def*)
    have ($\forall$ *p*$\in$ *set yes*. *P p*) $\wedge$ ($\forall$ *p*$\in$ *set* (*snd ?p*). $\neg$ *P p*)
      by (*rule Cons.hyps*) (*simp add*: *yes no*)
    with *False no* **show** *?thesis* **by** *simp*
  qed
qed

lemma *partition-filter1*:
    fst (*partition P xs*) = *filter P xs*
by (*induct xs rule*: *partition.induct*) (*auto simp add*: *Let-def split-def*)

lemma *partition-filter2*:
    snd (*partition P xs*) = *filter* (*Not o P*) *xs*
by (*induct xs rule*: *partition.induct*) (*auto simp add*: *Let-def split-def*)

lemma *partition-set*:
  assumes *partition P xs* = (*yes*, *no*)

shows *set yes* ∪ *set no* = *set xs*
proof −
have *set xs* = {*x*. *x* ∈ *set xs* ∧ *P x*} ∪ {*x*. *x* ∈ *set xs* ∧ ¬ *P x*} **by** *blast*
also have ... = *set* (*List.filter P xs*) *Un* (*set* (*List.filter* (*Not o P*) *xs*)) **by** *simp*
also have ... = *set* (*fst* (*partition P xs*)) ∪ *set* (*snd* (*partition P xs*))
using *partition-filter1* [*of P xs*] *partition-filter2* [*of P xs*] **by** *simp*
finally show *set yes Un set no* = *set xs* **using** *assms* **by** *simp*
qed

end

49 Map: Maps

theory *Map*
imports *List*
begin

types ('*a*,'*b*) ~=> = '*a* => '*b option* (**infixr** *0*)
translations (*type*) *a* ~=> *b* <= (*type*) *a* => *b option*

syntax (*xsymbols*)
~=> :: [*type*, *type*] => *type* (**infixr** ⇀ *0*)

abbreviation
empty :: '*a* ~=> '*b* **where**
empty == %*x*. *None*

definition
map-comp :: ('*b* ~=> '*c*) => ('*a* ~=> '*b*) => ('*a* ~=> '*c*) (**infixl** *o*'-*m* *55*)
where
f o-m g = (λ*k*. *case g k of None* ⇒ *None* | *Some v* ⇒ *f v*)

notation (*xsymbols*)
map-comp (**infixl** $\circ_m$ *55*)

definition
map-add :: ('*a* ~=> '*b*) => ('*a* ~=> '*b*) => ('*a* ~=> '*b*) (**infixl** ++ *100*)
where
m1 ++ *m2* = (λ*x*. *case m2 x of None* => *m1 x* | *Some y* => *Some y*)

definition
restrict-map :: ('*a* ~=> '*b*) => '*a set* => ('*a* ~=> '*b*) (**infixl** |' *110*) **where**
m|'*A* = (λ*x*. *if x* : *A then m x else None*)

notation (*latex* **output**)
restrict-map (-↾- [*111*,*110*] *110*)

definition

dom :: (*'a* ~=> *'b*) => *'a set* **where**
dom m = {*a*. *m a* ~= *None*}

definition
ran :: (*'a* ~=> *'b*) => *'b set* **where**
ran m = {*b*. *EX a*. *m a* = *Some b*}

definition
map-le :: (*'a* ~=> *'b*) => (*'a* ~=> *'b*) => *bool* (**infix** $\subseteq_m$ *50*) **where**
($m_1 \subseteq_m m_2$) = ($\forall\, a \in dom\ m_1.\ m_1\ a = m_2\ a$)

consts
map-of :: (*'a* ∗ *'b*) *list* => *'a* ~=> *'b*
map-upds :: (*'a* ~=> *'b*) => *'a list* => *'b list* => (*'a* ~=> *'b*)

nonterminals
maplets maplet

syntax
-maplet :: [*'a*, *'a*] => *maplet* (- /|−>/ -)
-maplets :: [*'a*, *'a*] => *maplet* (- /[|−>]/ -)
:: *maplet* => *maplets* (-)
-Maplets :: [*maplet*, *maplets*] => *maplets* (-,/ -)
-MapUpd :: [*'a* ~=> *'b*, *maplets*] => *'a* ~=> *'b* (-/'(-') [*900*,*0*]*900*)
-Map :: *maplets* => *'a* ~=> *'b* ((*1*[-]))

syntax (*xsymbols*)
-maplet :: [*'a*, *'a*] => *maplet* (- /↦/ -)
-maplets :: [*'a*, *'a*] => *maplet* (- /[↦]/ -)

translations
-MapUpd m (*-Maplets xy ms*) == *-MapUpd* (*-MapUpd m xy*) *ms*
-MapUpd m (*-maplet x y*) == *m*(*x*:=*Some y*)
-MapUpd m (*-maplets x y*) == *map-upds m x y*
-Map ms == *-MapUpd* (*CONST empty*) *ms*
-Map (*-Maplets ms1 ms2*) <= *-MapUpd* (*-Map ms1*) *ms2*
-Maplets ms1 (*-Maplets ms2 ms3*) <= *-Maplets* (*-Maplets ms1 ms2*) *ms3*

primrec
map-of [] = *empty*
map-of (*p*#*ps*) = (*map-of ps*)(*fst p* |−> *snd p*)

defs
map-upds-def [*code func*]: *m*(*xs* [|−>] *ys*) == *m* ++ *map-of* (*rev*(*zip xs ys*))

49.1 *empty*

lemma *empty-upd-none* [*simp*]: *empty*(*x* := *None*) = *empty*
by (*rule ext*) *simp*

49.2 *map-upd*

lemma *map-upd-triv*: $t\ k = Some\ x ==> t(k|->x) = t$
by (*rule ext*) *simp*

lemma *map-upd-nonempty* [*simp*]: $t(k|->x)$ ~= *empty*
proof
  assume $t(k \mapsto x) = empty$
  then have $(t(k \mapsto x))\ k = None$ **by** *simp*
  then show *False* **by** *simp*
qed

lemma *map-upd-eqD1*:
  assumes $m(a\mapsto x) = n(a\mapsto y)$
  shows $x = y$
proof −
  from *prems* **have** $(m(a\mapsto x))\ a = (n(a\mapsto y))\ a$ **by** *simp*
  then show *?thesis* **by** *simp*
qed

lemma *map-upd-Some-unfold*:
  $((m(a|->b))\ x = Some\ y) = (x = a \wedge b = y \vee x \neq a \wedge m\ x = Some\ y)$
by *auto*

lemma *image-map-upd* [*simp*]: $x \notin A \Longrightarrow m(x \mapsto y)$ ‘ $A = m$ ‘ A
by *auto*

lemma *finite-range-updI*: *finite* (*range f*) ==> *finite* (*range* $(f(a|->b)))$
unfolding *image-def*
apply (*simp* (*no-asm-use*) *add:full-SetCompr-eq*)
apply (*rule finite-subset*)
 prefer *2* **apply** *assumption*
apply (*auto*)
done

49.3 *map-of*

lemma *map-of-eq-None-iff*:
  (*map-of xys x* = *None*) = ($x \notin$ *fst* ‘ (*set xys*))
by (*induct xys*) *simp-all*

lemma *map-of-is-SomeD*: *map-of xys x* = *Some y* $\Longrightarrow$ $(x,y) \in$ *set xys*
apply (*induct xys*)
 apply *simp*
apply (*clarsimp split*: *if-splits*)
done

lemma *map-of-eq-Some-iff* [*simp*]:
  distinct(*map fst xys*) $\Longrightarrow$ (*map-of xys x* = *Some y*) = ($(x,y) \in$ *set xys*)
apply (*induct xys*)

apply *simp*
apply (*auto simp*: *map-of-eq-None-iff* [*symmetric*])
done

lemma *Some-eq-map-of-iff* [*simp*]:
distinct(*map fst xys*) $\Longrightarrow$ (*Some y* = *map-of xys x*) = ((*x*,*y*) $\in$ *set xys*)
by (*auto simp del*:*map-of-eq-Some-iff simp add*: *map-of-eq-Some-iff* [*symmetric*])

lemma *map-of-is-SomeI* [*simp*]: $[\![$ *distinct*(*map fst xys*); (*x*,*y*) $\in$ *set xys* $]\!]$
$\Longrightarrow$ *map-of xys x* = *Some y*
apply (*induct xys*)
apply *simp*
apply *force*
done

lemma *map-of-zip-is-None* [*simp*]:
length xs = *length ys* $\Longrightarrow$ (*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 add*: *image-constant*)
apply (*rule finite-subset*)
prefer *2* **apply** *assumption*
apply *auto*
done

lemma *map-of-SomeD*: *map-of xs k* = *Some y* $\Longrightarrow$ (*k*, *y*) $\in$ *set xs*
by (*induct xs*) (*simp*, *atomize* (*full*), *auto*)

lemma *map-of-mapk-SomeI*:
inj f ==> *map-of t k* = *Some x* ==>
map-of (*map* (*split* (%*k*. *Pair* (*f k*))) *t*) (*f k*) = *Some x*
by (*induct t*) (*auto simp add*: *inj-eq*)

lemma *weak-map-of-SomeI*: (*k*, *x*) : *set l* ==> $\exists$ *x*. *map-of l k* = *Some x*
by (*induct l*) *auto*

lemma *map-of-filter-in*:
map-of xs k = *Some z* $\Longrightarrow$ *P k z* $\Longrightarrow$ *map-of* (*filter* (*split P*) *xs*) *k* = *Some z*
by (*induct xs*) *auto*

lemma *map-of-map*: *map-of* (*map* (%(*a*,*b*). (*a*,*f b*)) *xs*) *x* = *option-map f* (*map-of xs x*)
by (*induct xs*) *auto*

49.4 *option-map* related

lemma *option-map-o-empty* [*simp*]: *option-map f o empty* = *empty*

by (*rule ext*) *simp*

lemma *option-map-o-map-upd* [*simp*]:
option-map f o m(*a*|−>*b*) = (*option-map f o m*)(*a*|−>*f b*)
by (*rule ext*) *simp*

49.5 *map-comp* related

lemma *map-comp-empty* [*simp*]:
$m \circ_m empty = empty$
$empty \circ_m m = empty$
by (*auto simp add*: *map-comp-def intro*: *ext split*: *option.splits*)

lemma *map-comp-simps* [*simp*]:
$m2\ k = None \Longrightarrow (m1 \circ_m m2)\ k = None$
$m2\ k = Some\ k' \Longrightarrow (m1 \circ_m m2)\ k = m1\ k'$
by (*auto simp add*: *map-comp-def*)

lemma *map-comp-Some-iff*:
$((m1 \circ_m m2)\ k = Some\ v) = (\exists k'.\ m2\ k = Some\ k' \wedge m1\ k' = Some\ v)$
by (*auto simp add*: *map-comp-def split*: *option.splits*)

lemma *map-comp-None-iff*:
$((m1 \circ_m m2)\ k = None) = (m2\ k = None \vee (\exists k'.\ m2\ k = Some\ k' \wedge m1\ k' = None))$
by (*auto simp add*: *map-comp-def split*: *option.splits*)

49.6 ++

lemma *map-add-empty*[*simp*]: *m* ++ *empty* = *m*
by(*simp add*: *map-add-def*)

lemma *empty-map-add*[*simp*]: *empty* ++ *m* = *m*
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-assoc*[*simp*]: *m1* ++ (*m2* ++ *m3*) = (*m1* ++ *m2*) ++ *m3*
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-Some-iff*:
((*m* ++ *n*) *k* = *Some x*) = (*n k* = *Some x* | *n k* = *None* & *m k* = *Some x*)
by (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-SomeD* [*dest*!]:
$(m \mathbin{++} n)\ k = Some\ x \Longrightarrow n\ k = Some\ x \vee n\ k = None \wedge m\ k = Some\ x$
by (*rule map-add-Some-iff* [*THEN iffD1*])

lemma *map-add-find-right* [*simp*]: !!*xx*. *n k* = *Some xx* ==> (*m* ++ *n*) *k* = *Some xx*
by (*subst map-add-Some-iff*) *fast*

lemma *map-add-None* [*iff*]: $((m ++ n)\ k = None) = (n\ k = None\ \&\ m\ k = None)$
by (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-upd*[*simp*]: $f ++ g(x|->y) = (f ++ g)(x|->y)$
by (*rule ext*) (*simp add*: *map-add-def*)

lemma *map-add-upds*[*simp*]: $m1 ++ (m2(xs[\mapsto]ys)) = (m1{+}{+}m2)(xs[\mapsto]ys)$
by (*simp add*: *map-upds-def*)

lemma *map-of-append*[*simp*]: *map-of* (*xs* @ *ys*) = *map-of ys* ++ *map-of xs*
unfolding *map-add-def*
apply (*induct xs*)
 apply *simp*
apply (*rule ext*)
apply (*simp split add*: *option.split*)
done

lemma *finite-range-map-of-map-add*:
  finite (*range f*) ==> *finite* (*range* (*f* ++ *map-of l*))
apply (*induct l*)
 apply (*auto simp del*: *fun-upd-apply*)
apply (*erule finite-range-updI*)
done

lemma *inj-on-map-add-dom* [*iff*]:
  inj-on $(m ++ m')$ $(dom\ m')$ = *inj-on* m' $(dom\ m')$
by (*fastsimp simp*: *map-add-def dom-def inj-on-def split*: *option.splits*)

49.7 *restrict-map*

lemma *restrict-map-to-empty* [*simp*]: $m|`\{\} = empty$
by (*simp add*: *restrict-map-def*)

lemma *restrict-map-empty* [*simp*]: $empty|`D = empty$
by (*simp add*: *restrict-map-def*)

lemma *restrict-in* [*simp*]: $x \in A \Longrightarrow (m|`A)\ x = m\ x$
by (*simp add*: *restrict-map-def*)

lemma *restrict-out* [*simp*]: $x \notin A \Longrightarrow (m|`A)\ x = None$
by (*simp add*: *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*]:
$m(x := y)|`D = (\mathit{if}\ x \in D\ \mathit{then}\ (m|`(D-\{x\}))(x := y)\ \mathit{else}\ m|`D)$
by (*simp add*: *restrict-map-def expand-fun-eq*)

lemma *fun-upd-None-restrict* [*simp*]:
$(m|`D)(x := \mathit{None}) = (\mathit{if}\ x{:}D\ \mathit{then}\ m|`(D - \{x\})\ \mathit{else}\ m|`D)$
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*)

49.8 *map-upds*

lemma *map-upds-Nil1* [*simp*]: $m([]\ [|->]\ bs) = m$
by (*simp add*: *map-upds-def*)

lemma *map-upds-Nil2* [*simp*]: $m(as\ [|->]\ []) = m$
by (*simp add*:*map-upds-def*)

lemma *map-upds-Cons* [*simp*]: $m(a\#as\ [|->]\ b\#bs) = (m(a|->b))(as[|->]bs)$
by (*simp add*:*map-upds-def*)

lemma *map-upds-append1* [*simp*]: $\bigwedge ys\ m.\ \mathit{size}\ xs < \mathit{size}\ ys \Longrightarrow$
$m(xs@[x]\ [\mapsto]\ ys) = m(xs\ [\mapsto]\ ys)(x \mapsto ys!\mathit{size}\ xs)$
apply(*induct xs*)
 apply (*clarsimp simp add*: *neq-Nil-conv*)
apply (*case-tac ys*)
 apply *simp*
apply *simp*
done

lemma *map-upds-list-update2-drop* [*simp*]:
$[\![\mathit{size}\ xs \le i;\ i < \mathit{size}\ ys]\!]$
$\Longrightarrow m(xs[\mapsto]ys[i{:=}y]) = m(xs[\mapsto]ys)$
apply (*induct xs arbitrary*: *m ys i*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp*
apply (*simp split*: *nat.split*)

done

lemma *map-upd-upds-conv-if*:
 $(f(x|->y))(xs\ [|->]\ ys) =$
  $(if\ x : set(take\ (length\ ys)\ xs)\ then\ f(xs\ [|->]\ ys)$
   $else\ (f(xs\ [|->]\ ys))(x|->y))$
apply (*induct xs arbitrary*: *x y ys f*)
 apply *simp*
apply (*case-tac ys*)
 apply (*auto split*: *split-if simp*: *fun-upd-twist*)
done

lemma *map-upds-twist* [*simp*]:
 $a\ \tilde{}: set\ as ==> m(a|->b)(as[|->]bs) = m(as[|->]bs)(a|->b)$
using *set-take-subset* **by** (*fastsimp simp add*: *map-upd-upds-conv-if*)

lemma *map-upds-apply-nontin* [*simp*]:
 $x\ \tilde{}: set\ xs ==> (f(xs[|->]ys))\ x = f\ x$
apply (*induct xs arbitrary*: *ys*)
 apply *simp*
apply (*case-tac ys*)
 apply (*auto simp*: *map-upd-upds-conv-if*)
done

lemma *fun-upds-append-drop* [*simp*]:
 $size\ xs = size\ ys \Longrightarrow m(xs@zs[\mapsto]ys) = m(xs[\mapsto]ys)$
apply (*induct xs arbitrary*: *m ys*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp-all*
done

lemma *fun-upds-append2-drop* [*simp*]:
 $size\ xs = size\ ys \Longrightarrow m(xs[\mapsto]ys@zs) = m(xs[\mapsto]ys)$
apply (*induct xs arbitrary*: *m ys*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp-all*
done

lemma *restrict-map-upds*[*simp*]:
 $[\![\ length\ xs = length\ ys;\ set\ xs \subseteq D\ ]\!]$
  $\Longrightarrow m(xs\ [\mapsto]\ ys)|`D = (m|`(D - set\ xs))(xs\ [\mapsto]\ ys)$
apply (*induct xs arbitrary*: *m ys*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp*
apply (*simp add*: *Diff-insert* [*symmetric*] *insert-absorb*)

apply (*simp add*: *map-upd-upds-conv-if*)
done

49.9 *dom*

lemma *domI*: *m a = Some b ==> a : dom m*
by(*simp add*:*dom-def*)

lemma *domD*: *a : dom m ==>* $\exists$ *b. m a = Some b*
by (*cases m a*) (*auto simp add*: *dom-def*)

lemma *domIff* [*iff*, *simp del*]: (*a : dom m*) = (*m a ~= None*)
by(*simp add*:*dom-def*)

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

lemma *dom-fun-upd* [*simp*]:
  dom(*f*(*x := y*)) = (*if y=None then dom f* − {*x*} *else insert x* (*dom f*))
by(*auto simp add*:*dom-def*)

lemma *dom-map-of*: *dom*(*map-of xys*) = {*x*. $\exists$ *y*. (*x*,*y*) : *set xys*}
by (*induct xys*) (*auto simp del*: *fun-upd-apply*)

lemma *dom-map-of-conv-image-fst*:
  dom(*map-of xys*) = *fst* ' (*set xys*)
by(*force simp*: *dom-map-of*)

lemma *dom-map-of-zip* [*simp*]: [| *length xs = length ys*; *distinct xs* |] ==>
  dom(*map-of*(*zip xs ys*)) = *set xs*
by (*induct rule*: *list-induct2*) *simp-all*

lemma *finite-dom-map-of*: *finite* (*dom* (*map-of l*))
by (*induct l*) (*auto simp add*: *dom-def insert-Collect* [*symmetric*])

lemma *dom-map-upds* [*simp*]:
  dom(*m*(*xs*[|−>]*ys*)) = *set*(*take* (*length ys*) *xs*) *Un dom m*
apply (*induct xs arbitrary*: *m ys*)
 apply *simp*
apply (*case-tac ys*)
 apply *auto*
done

lemma *dom-map-add* [*simp*]: *dom*(*m++n*) = *dom n Un dom m*
by(*auto simp*:*dom-def*)

lemma *dom-override-on* [*simp*]:
  dom(*override-on f g A*) =

$(dom\ f\ -\ \{a.\ a : A - dom\ g\})\ Un\ \{a.\ a : A\ Int\ dom\ g\}$
by(*auto simp*: *dom-def override-on-def*)

lemma *map-add-comm*: $dom\ m1 \cap dom\ m2 = \{\} \Longrightarrow m1{+}{+}m2 = m2{+}{+}m1$
by (*rule ext*) (*force simp*: *map-add-def dom-def split*: *option.split*)

lemma *finite-map-freshness*:
$finite\ (dom\ (f :: 'a \rightharpoonup 'b)) \Longrightarrow \neg\ finite\ (UNIV :: 'a\ set) \Longrightarrow$
$\exists x.\ f\ x = None$
by(*bestsimp dest*:*ex-new-if-finite*)

49.10 *ran*

lemma *ranI*: $m\ a = Some\ b ==> b : ran\ m$
by(*auto simp*: *ran-def*)

lemma *ran-empty* [*simp*]: $ran\ empty = \{\}$
by(*auto simp*: *ran-def*)

lemma *ran-map-upd* [*simp*]: $m\ a = None ==> ran(m(a|->b)) = insert\ b\ (ran\ m)$
unfolding *ran-def*
apply *auto*
apply (*subgoal-tac* $aa\ {}^{\sim}{=}\ a$)
apply *auto*
done

49.11 *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*]:
$f \subseteq_m g ==> f(as\ [|->]\ bs) \subseteq_m g(as\ [|->]\ bs)$
apply (*induct as arbitrary*: *f g bs*)
apply *simp*
apply (*case-tac bs*)
apply *auto*

done

lemma *map-le-implies-dom-le*: $(f \subseteq_m g) \Longrightarrow (dom\ f \subseteq dom\ g)$
by (*fastsimp simp add*: *map-le-def dom-def*)

lemma *map-le-refl* [*simp*]: $f \subseteq_m f$
by (*simp add*: *map-le-def*)

lemma *map-le-trans*[*trans*]: $\llbracket\ m1 \subseteq_m m2;\ m2 \subseteq_m m3 \rrbracket \Longrightarrow m1 \subseteq_m m3$
by (*auto simp add*: *map-le-def dom-def*)

lemma *map-le-antisym*: $\llbracket\ f \subseteq_m g;\ g \subseteq_m f\ \rrbracket \Longrightarrow f = g$
unfolding *map-le-def*
apply (*rule ext*)
apply (*case-tac* $x \in dom\ f$, *simp*)
apply (*case-tac* $x \in dom\ g$, *simp*, *fastsimp*)
done

lemma *map-le-map-add* [*simp*]: $f \subseteq_m (g\ {+}{+}\ f)$
by (*fastsimp simp add*: *map-le-def*)

lemma *map-le-iff-map-add-commute*: $(f \subseteq_m f\ {+}{+}\ g) = (f{+}{+}g = g{+}{+}f)$
by(*fastsimp simp*: *map-add-def map-le-def expand-fun-eq split*: *option.splits*)

lemma *map-add-le-mapE*: $f{+}{+}g \subseteq_m h \Longrightarrow g \subseteq_m h$
by (*fastsimp simp add*: *map-le-def map-add-def dom-def*)

lemma *map-add-le-mapI*: $\llbracket\ f \subseteq_m h;\ g \subseteq_m h;\ f \subseteq_m f{+}{+}g\ \rrbracket \Longrightarrow f{+}{+}g \subseteq_m h$
by (*clarsimp simp add*: *map-le-def map-add-def dom-def split*: *option.splits*)

end

50 Main: Main HOL

theory *Main*
imports *Map*
begin

Theory *Main* includes everything. Note that theory *PreList* already includes most HOL theories.

ML ⟪ *val HOL-proofs* = ! *Proofterm.proofs* ⟫

end

References