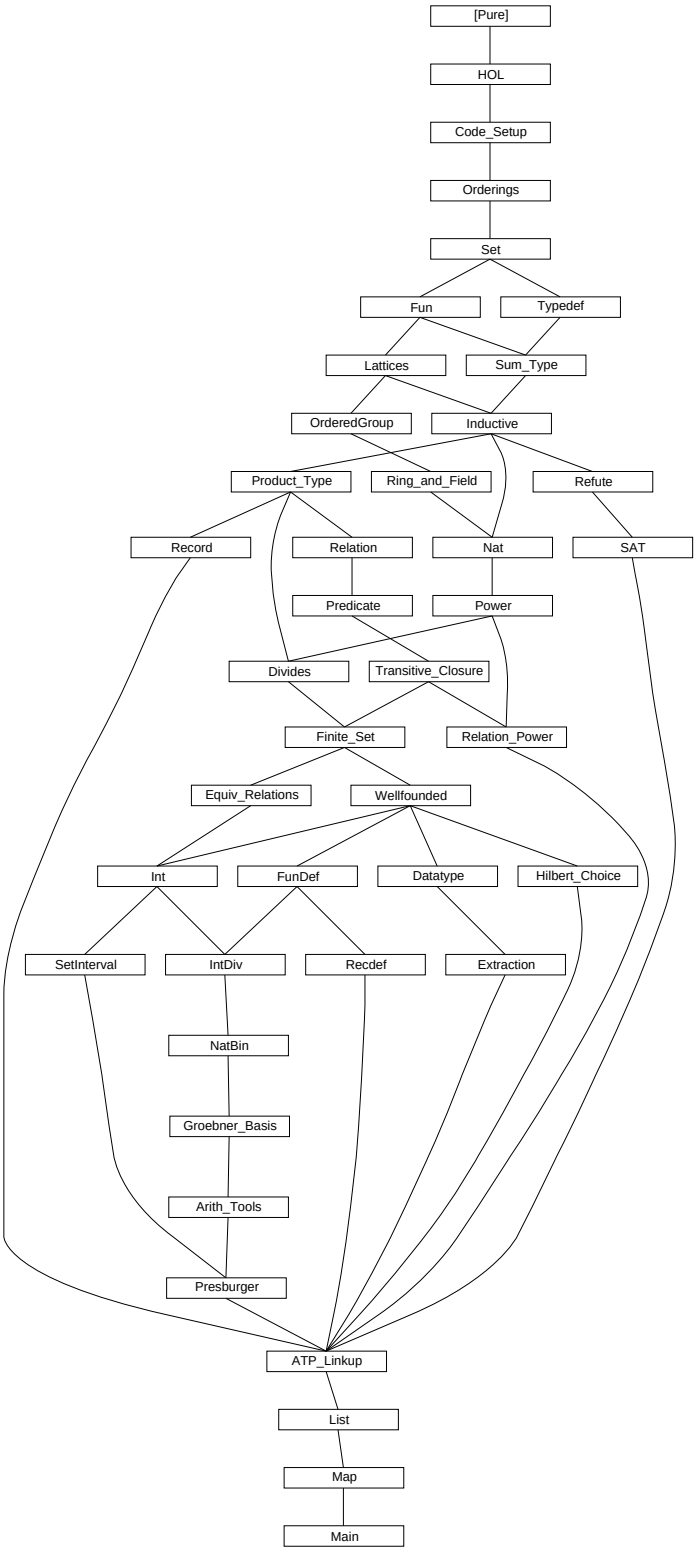
Isabelle/HOL — Higher-Order Logic

June 8, 2008

Contents


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

1 HOL: The basis of Higher-Order Logic

theory *HOL*
imports *Pure*
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*/*random-word.ML*
~~/*src*/*Tools*/*atomize-elim.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*
setup ⟪ *ObjectLogic.add-base-sort* @{*sort type*} ⟫

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

global

typedecl *bool*

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

consts

Not :: *bool* => *bool* (~ - [*40*] *40*)
True :: *bool*
False :: *bool*
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** $\longleftrightarrow$ *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 ⇒ 'a ⇒ 'a* (**infixl** + *65*)

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

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

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

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

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

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

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

end

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

class *ord = type +*
fixes *less-eq* :: *'a ⇒ 'a ⇒ bool*

and *less* :: *'a* ⇒ *'a* ⇒ *bool*
begin

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

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

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

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

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

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

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*) ⋄

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

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

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

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

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

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

For calculational reasoning:

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

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

1.2.2 Congruence rules for application

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

done

lemma *arg-cong*: $x=y$ ==> $f(x)=f(y)$
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* ⟹ ¬ *Pa*) ⟹ (*P* ⟹ *Pa*) ⟹ ¬ *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*: $\sim Q$
shows $\sim P$
apply (*rule nq* [*THEN contrapos-nn*])
apply (*erule pq*)
done

1.2.9 Existential quantifier

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

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

1.2.10 Conjunction

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

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

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

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

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

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

1.2.11 Disjunction

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

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

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

1.2.12 Classical logic

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

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

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

lemma *notnotD*: $\sim\sim P \Longrightarrow P$
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 $^{\sim}Q\ ==>\ P$ **shows** $P|Q$
apply (*rule classical*)

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

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

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

lemma *case-split-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* $\exists\, x.\ P\ x$])
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*: $\sim P$
    and *2*: $\sim P$ $==> P$
  shows R
proof −
  from *2* **and** *1* **have** P **.**
  with *1* **show** R **by** (*rule notE*)
qed

lemma *TrueE*: *True* $==> P$ $==> P$ **.**
lemma *notFalseE*: $\sim$ *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*

1.2.18 Atomizing elimination rules

setup *AtomizeElim.setup*

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

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

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

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

1.3 Package setup

1.3.1 Classical Reasoner setup

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

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

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

ML ⟪
structure Hypsubst = HypsubstFun(
struct
structure Simplifier = Simplifier
val dest-eq = HOLogic.dest-eq
val dest-Trueprop = HOLogic.dest-Trueprop
val dest-imp = HOLogic.dest-imp
val eq-reflection = @{thm eq-reflection}
val rev-eq-reflection = @{thm meta-eq-to-obj-eq}
val imp-intr = @{thm impI}

```
  val rev-mp = @{thm rev-mp}
  val subst = @{thm subst}
  val sym = @{thm sym}
  val thin-refl = @{thm thin-refl};
end);
open Hypsubst;

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

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

ML-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 = The-
orems blacklisted for ATP);
⟩⟩
```

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

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

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

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

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

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

ML ⟪ *val HOL-cs* = @{*claset*} ⟫

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

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

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

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

ML ⟪
structure Blast = BlastFun
(
  type claset = Classical.claset
  val equality-name = @{const-name op =}
  val not-name = @{const-name Not}
  val notE = @{thm notE}
  val ccontr = @{thm ccontr}
  val contr-tac = Classical.contr-tac
  val dup-intr = Classical.dup-intr
  val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
  val 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
    $(True \longrightarrow P) = P \ \ (False \longrightarrow P) = True$
    $(P \longrightarrow True) = True \ \ (P \longrightarrow P) = True$
    $(P \longrightarrow False) = (\sim P) \ \ (P \longrightarrow \sim P) = (\sim P)$
    $(P \ \& \ True) = P \ \ (True \ \& \ P) = P$
    $(P \ \& \ False) = False \ \ (False \ \& \ P) = False$
    $(P \ \& \ P) = P \ \ (P \ \& \ (P \ \& \ Q)) = (P \ \& \ Q)$
    $(P \ \& \sim P) = False \quad (\sim P \ \& \ P) = False$

$(P \mid \mathit{True}) = \mathit{True}$ $(\mathit{True} \mid P) = \mathit{True}$
$(P \mid \mathit{False}) = P$ $(\mathit{False} \mid P) = P$
$(P \mid P) = P$ $(P \mid (P \mid Q)) = (P \mid Q)$ **and**
$(\mathit{ALL}\ x.\ P) = P$ $(\mathit{EX}\ x.\ P) = P$ $\mathit{EX}\ x.\ x{=}t$ $\mathit{EX}\ x.\ t{=}x$
— needed for the one-point-rule quantifier simplification procs
— essential for termination!! **and**
$!!P.\ (\mathit{EX}\ x.\ x{=}t\ \&\ P(x)) = P(t)$
$!!P.\ (\mathit{EX}\ x.\ t{=}x\ \&\ P(x)) = P(t)$
$!!P.\ (\mathit{ALL}\ x.\ x{=}t \longrightarrow P(x)) = P(t)$
$!!P.\ (\mathit{ALL}\ x.\ t{=}x \longrightarrow P(x)) = P(t)$
by (*blast*, *blast*, *blast*, *blast*, *blast*, *iprover*+)

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

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

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

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

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

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

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

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

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

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

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

lemma *imp-conjR*: $(P \dashrightarrow (Q\&R)) = ((P\dashrightarrow Q) \;\&\; (P\dashrightarrow R))$ **by** *iprover*
lemma *imp-conjL*: $((P\&Q) \dashrightarrow R) = (P \dashrightarrow (Q \dashrightarrow R))$ **by** *iprover*
lemma *imp-disjL*: $((P|Q) \dashrightarrow R) = ((P\dashrightarrow R)\&(Q\dashrightarrow R))$ **by** *iprover*

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

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

lemma *imp-disj1*: $((P\dashrightarrow Q)|R) = (P\dashrightarrow Q|R)$ **by** *blast*
lemma *imp-disj2*: $(Q|(P\dashrightarrow R)) = (P\dashrightarrow Q|R)$ **by** *blast*

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

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

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

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

lemma *not-all*: $(\sim (!\ x.\ P(x))) = (?\ x.\sim P(x))$ **by** *blast*
lemma *imp-all*: $((!\ x.\ P\ x) \dashrightarrow Q) = (?\ x.\ P\ x \dashrightarrow Q)$ **by** *blast*
lemma *not-ex*: $(\sim (?\ x.\ P(x))) = (!\ x.\sim P(x))$ **by** *iprover*
lemma *imp-ex*: $((?\ x.\ P\ x) \dashrightarrow Q) = (!\ x.\ P\ x \dashrightarrow Q)$ **by** *iprover*
lemma *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*:
$(Q = Q') ==> (Q' ==> (P = P')) ==> ((P \ \& \ Q) = (P' \ \& \ Q'))$
by *iprover*

The | congruence rule: not included by default!

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

if-then-else rules

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

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

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

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

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

lemma *split-if-asm*: $P\ (if\ Q\ then\ x\ else\ y) = (\sim((Q \ \& \ \sim P\ x) \mid (\sim Q \ \& \ \sim P\ y)))$
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*: $(if\ P\ then\ Q\ else\ R) = ((P-->Q) \ \& \ (\sim P-->R))$
— This form is useful for expanding *if*s on the RIGHT of the ==> symbol.
by (*rule split-if*)

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

lemma *Eq-TrueI*: $P \Longrightarrow P == True$ **by** (*unfold atomize-eq*) *iprover*
lemma *Eq-FalseI*: $\sim P \Longrightarrow P == False$ **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*)
[*code func del*]: *simp-implies* $\equiv$ *op* ==>

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

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

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

and P: $PROP\ P$
from PP' **and** P **have** P': $PROP\ P'$ **by** (*rule equal-elim-rule1*)
then have $PROP\ Q'$ **by** (*rule* $P'Q'$)
with $P'QQ'$ [*OF* P', *symmetric*] **show** $PROP\ Q$
by (*rule equal-elim-rule1*)
qed

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

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

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

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

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

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

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

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

simproc-setup *neq* $(x = y) =$ ⟪ *fn - =>*
let
val neq-to-EQ-False = @{thm not-sym} RS @{thm Eq-FalseI};
fun is-neq eq lhs rhs thm =
(case Thm.prop-of thm of
- $ (Not $ (eq' $ l' $ r')) =>
Not = HOLogic.Not andalso eq' = eq andalso

```
        r' aconv lhs andalso l' aconv rhs
    | - => false);
  fun proc ss ct =
    (case Thm.term-of ct of
      eq $ lhs $ rhs =>
        (case find-first (is-neq eq lhs rhs) (Simplifier.prems-of-ss ss) of
          SOME thm => SOME (thm RS neq-to-EQ-False)
        | NONE => NONE)
    | - => NONE);
in proc end;
⟩⟩

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

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

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

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

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

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

Method setup.

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

setup *Induct.setup*

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

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

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

code-datatype *Trueprop prop*

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

lemma *Let-case-cert*:

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

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

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

class *eq* = *type* +
fixes *eq* :: *'a* ⇒ *'a* ⇒ *bool*
assumes *eq*: *eq x y* ⟷ *x* = *y*
begin

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

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

end

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

setup ⟪
CodeUnit.add-const-alias @{*thm equals-eq*}
#> *CodeName.setup*
#> *CodeTarget.setup*
#> *Nbe.setup*
⟫

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*)+

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

⟩⟩

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

setup ⟪
let

fun eq-codegen thy defs 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*, *Codegen.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 func*, *code unfold*, *code post*] = *imp-conv-disj*

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

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

code-reserved *SML*
bool true false not

code-reserved *OCaml*
bool not

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' (fn k => TRY (rtac TrueI k))
  ))
⟩⟩ solve goal by normalization
```

end

3 Orderings: Abstract orderings

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

3.1 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*

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

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

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

Transitivity rules for calculational reasoning

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

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

Asymmetry.

lemma *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*)

Dual order

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

end

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

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

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

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

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

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

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

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

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

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

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

Replacing the old Nat.leI

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

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

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

Dual order

lemma *dual-linorder*:
linorder (*op* $\geq$) (*op* $>$)
by *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

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

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

3.4 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*

3.5 Bounded quantifiers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.6 Transitivity reasoning

context *ord*
begin

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

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

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

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

end

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.7 Order on bool

instantiation *bool* :: *order*
begin

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

definition
less-bool-def [*code func del*]: $(P::bool) < Q \longleftrightarrow P \leq Q \wedge P \neq Q$

instance
by *intro-classes* (*auto simp add*: *le-bool-def less-bool-def*)

end

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

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

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

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

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

3.8 Order on functions

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

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

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

instance ..

end

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

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

lemma *le-funE*: $f \leq g \Longrightarrow (f\ x \leq g\ x \Longrightarrow P) \Longrightarrow P$

unfolding *le-fun-def* **by** *simp*

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

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

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

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

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

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

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

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

3.9 Monotonicity, least value operator and min/max

context *order*
begin

definition

mono :: (*'a* ⇒ *'b*::*order*) ⇒ *bool*
where
mono f ⟷ (∀ *x y*. *x* ≤ *y* ⟶ *f x* ≤ *f y*)

lemma *monoI* [*intro?*]:
fixes *f* :: *'a* ⇒ *'b*::*order*
shows (⋀*x y*. *x* ≤ *y* ⟹ *f x* ≤ *f y*) ⟹ *mono f*
unfolding *mono-def* **by** *iprover*

lemma *monoD* [*dest?*]:
fixes *f* :: *'a* ⇒ *'b*::*order*
shows *mono f* ⟹ *x* ≤ *y* ⟹ *f x* ≤ *f y*
unfolding *mono-def* **by** *iprover*

end

context *linorder*
begin

lemma *min-of-mono*:
fixes *f* :: *'a* ⇒ *'b*::*linorder*
shows *mono f* ⟹ *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* ⇒ *'b*::*linorder*
shows *mono f* ⟹ *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* ≤ *y* |] ==> *Q x* |]
==> *Q* (*Least P*)
apply (*unfold Least-def*)
apply (*rule theI2*)
apply (*blast 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*: (⋀*x*::*'a*::*order*. *least* ≤ *x*) ⟹ *min x least* = *least*
apply (*simp add*: *min-def*)

apply (*blast intro*: *order-antisym*)
done

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

end

4 Set: Set theory for higher-order logic

theory *Set*
imports *Orderings*
begin

A set in HOL is simply a predicate.

4.1 Basic syntax

global

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

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

4.2 Additional concrete syntax

abbreviation
range :: $('a \Rightarrow 'b) \Rightarrow 'b\ set$ **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** $\cap$ *70*) **and**
op Un (**infixl** $\cup$ *65*) **and**
op : (*op* $\in$) **and**
op : ((-/ $\in$ -) [*50*, *51*] *50*) **and**
not-mem (*op* $\notin$) **and**
not-mem ((-/ $\notin$ -) [*50*, *51*] *50*) **and**
Union ($\bigcup$- [*90*] *90*) **and**
Inter ($\bigcap$- [*90*] *90*)

notation (*HTML* **output**)
op Int (**infixl** $\cap$ *70*) **and**
op Un (**infixl** $\cup$ *65*) **and**
op : (*op* $\in$) **and**
op : ((-/ $\in$ -) [*50*, *51*] *50*) **and**
not-mem (*op* $\notin$) **and**
not-mem ((-/ $\notin$ -) [*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.

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

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

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

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

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

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

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

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

4.2.1 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;
⟫
```

4.3 Rules and definitions

Isomorphisms between predicates and sets.

```
defs
  mem_def: x : S == S x
  Collect_def: Collect P == P

defs
  Ball_def:    Ball A P      == ALL x. x:A --> P(x)
  Bex_def:     Bex A P       == EX x. x:A & P(x)
  Bex1_def:    Bex1 A P      == EX! x. x:A & P(x)

instantiation fun :: (type, minus) minus
begin

definition
  fun_diff_def: A - B = (%x. A x - B x)

instance ..
```

end

instantiation *bool* :: *minus*
begin

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

instance ..

end

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

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

instance ..

end

instantiation *bool* :: *uminus*
begin

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

instance ..

end

defs
Un-def: $A\ Un\ B == \{x.\ x{:}A \mid x{:}B\}$
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: $f`A == \{y.\ EX\ x{:}A.\ y = f(x)\}$

4.4 Lemmas and proof tool setup

4.4.1 Relating predicates and sets

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

by (*simp add*: *Collect-def mem-def*)

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

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

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

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

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

4.4.2 Bounded quantifiers

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

lemmas *strip* = *impI allI ballI*

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

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

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

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

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

Gives better instantiation for bound:

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

lemma *bexI* [*intro*]: $P\ x \Longrightarrow x{:}A \Longrightarrow EX\ x{:}A.\ P\ x$
— Normally the best argument order: $P\ x$ constrains the choice of $x \in A$.
by (*unfold Bex-def*) *blast*

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

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

lemma *bexCI*: (*ALL x*:*A*. ~*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 ⟪
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*];
⟩⟩

4.4.3 Congruence rules

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

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

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

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

4.4.4 Subsets

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

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

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

declare *subsetD* [*intro?*] — FIXME

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

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

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

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

⟫

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

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

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

ML ⟪
fun set-mp-tac i = etac @{thm subsetCE} i THEN mp-tac i
⟫

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

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

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

4.4.5 Equality

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

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

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

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

Equality rules from ZF set theory – are they appropriate here?

lemma *equalityD1*: $A = B \Longrightarrow A \subseteq B$
by (*simp add*: *subset-refl*)

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

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

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

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

lemma *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*

4.4.6 The universal set – UNIV

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

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

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

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

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

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

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

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

4.4.7 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*: $A = \{\} \Longrightarrow a \notin A$
— 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*)

4.4.8 The Powerset operator – Pow

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

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

lemma *PowD*: $A \in Pow\ B \Longrightarrow A \subseteq B$
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*)

4.4.9 Set complement

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

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

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

lemma *ComplD* [*dest*!]: $c : -A \Longrightarrow c\tilde{}:A$

by (*simp add*: *mem-def fun-Compl-def bool-Compl-def*)

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

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

4.4.10 Binary union – Un

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

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

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

Classical introduction rule: no commitment to A vs B.

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

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

4.4.11 Binary intersection – Int

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

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

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

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

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

4.4.12 Set difference

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

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

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

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

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

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

4.4.13 Augmenting a set − insert

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

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

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

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

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

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

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

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

4.4.14 Singletons, using insert

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

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

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

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

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

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

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

lemma *subset-singletonD*: $A \subseteq \{x\} \Longrightarrow A = \{\} \mid A = \{x\}$
by *fast*

lemma *singleton-conv* [*simp*]: $\{x.\ x = a\} = \{a\}$
by *blast*

lemma *singleton-conv2* [*simp*]: $\{x.\ a = x\} = \{a\}$
by *blast*

lemma *diff-single-insert*: $A - \{x\} \subseteq B \Longrightarrow x \in A \Longrightarrow A \subseteq \mathit{insert}\ x\ B$
by *blast*

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

4.4.15 Unions of families

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

declare *UNION-def* [*noatp*]

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

lemma *UN-I* [*intro*]: $a{:}A \Longrightarrow b{:}\ B\ a \Longrightarrow b{:}\ (\mathit{UN}\ x{:}A.\ B\ x)$

— The order of the premises presupposes that A is rigid; b may be flexible.
by *auto*

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

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

4.4.16 Intersections of families

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

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

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

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

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

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

4.4.17 Union

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

lemma *UnionI* [*intro*]: $X{:}C \Longrightarrow A{:}X \Longrightarrow A : Union\ C$
— The order of the premises presupposes that C is rigid; A may be flexible.
by *auto*

lemma *UnionE* [*elim*!]: $A : Union\ C \Longrightarrow (!!X.\ A{:}X \Longrightarrow X{:}C \Longrightarrow R) \Longrightarrow R$
by (*unfold Union-def*) *blast*

4.4.18 Inter

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

lemma *InterI* [*intro*!]: $(!!X.\ X{:}C \Longrightarrow A{:}X) \Longrightarrow A : Inter\ C$
by (*simp add*: *Inter-def*)

A "destruct" rule – every X in C contains A as an element, but $A \in X$ can hold when $X \in C$ does not! This rule is analogous to *spec*.

lemma *InterD* [*elim*]: $A : Inter\ C \Longrightarrow X{:}C \Longrightarrow A{:}X$
by *auto*

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

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

declare *image-def* [*noatp*]

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

lemma *imageI*: $x : A \Longrightarrow f\ x : f\ `\ A$
by (*rule image-eqI*) (*rule refl*)

lemma *rev-image-eqI*: $x{:}A \Longrightarrow b = f\ x \Longrightarrow b : f`A$
— This version's more effective when we already have the required x.
by (*unfold image-def*) *blast*

lemma *imageE* [*elim*!]:
$b : (\%x.\ f\ x)`A \Longrightarrow (!!x.\ b = f\ x \Longrightarrow x{:}A \Longrightarrow P) \Longrightarrow P$
— The eta-expansion gives variable-name preservation.
by (*unfold image-def*) *blast*

lemma *image-Un*: $f`(A\ Un\ B) = f`A\ Un\ f`B$
by *blast*

lemma *image-eq-UN*: $f`A = (UN\ x{:}A.\ \{f\ x\})$
by *blast*

lemma *image-iff*: $(z : f`A) = (EX\ x{:}A.\ z = f\ x)$
by *blast*

lemma *image-subset-iff*: $(f`A \subseteq B) = (\forall x{\in}A.\ f\ x \in B)$
— This rewrite rule would confuse users if made default.

by *blast*

lemma *subset-image-iff*: $(B \subseteq f`A) = (EX\ AA.\ AA \subseteq A\ \&\ B = f`AA)$
apply *safe*
prefer *2* **apply** *fast*
apply (*rule-tac* $x = \{a.\ a : A\ \&\ f\ a : B\}$ **in** *exI*, *fast*)
done

lemma *image-subsetI*: $(!!x.\ x \in A ==> f\ x \in B) ==> f`A \subseteq B$
— 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*

4.4.19 Set reasoning tools

Rewrite rules for boolean case-splitting: faster than *split-if* [*split*].

lemma *split-if-eq1*: $((if\ Q\ then\ x\ else\ y) = b) = ((Q --> x = b)\ \&\ (\sim Q --> y = b))$
by (*rule split-if*)

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

lemma *split-if-mem2*: $(a : (if\ Q\ then\ x\ else\ y)) = ((Q --> a : x)\ \&\ (\sim Q --> a : y))$
by (*rule split-if* [**where** P=%$S.\ a : S$])

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

lemmas *mem-simps* =
insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff

mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [*simp*] above.

ML ⟪
val mksimps-pairs = [(Ball, @{thms bspec})] @ mksimps-pairs;
⟫
declaration ⟪ *fn - =>*
Simplifier.map-ss (fn ss => ss setmksimps (mksimps mksimps-pairs))
⟫

4.4.20 The "proper subset" relation

lemma *psubsetI* [*intro*!,*noatp*]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$
by (*unfold less-le*) *blast*

lemma *psubsetE* [*elim*!,*noatp*]:
$[|A \subset B;\ [|A \subseteq B; \sim (B \subseteq A)|] \Longrightarrow R|] \Longrightarrow R$
by (*unfold less-le*) *blast*

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

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

lemma *psubset-imp-subset*: $A \subset B \Longrightarrow A \subseteq B$
by (*simp add*: *psubset-eq*)

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

lemma *psubsetD*: $[|\ A \subset B;\ c \in A\ |] \Longrightarrow c \in B$
apply (*unfold less-le*)
apply (*auto dest*: *subsetD*)
done

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

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

lemma *psubset-imp-ex-mem*: $A \subset B \Longrightarrow \exists\, b.\ b \in (B - A)$

by (*unfold less-le*) *blast*

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

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

4.5 Further set-theory lemmas

4.5.1 Derived rules involving subsets.

insert.

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

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

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

Big Union – least upper bound of a set.

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

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

General union.

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

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

Big Intersection – greatest lower bound of a set.

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

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

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

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

lemma *INT-greatest*: $(!!x.\ x \in A \Longrightarrow C \subseteq B\ x) \Longrightarrow C \subseteq (\bigcap x{\in}A.\ B\ x)$
by (*iprover intro*: *INT-I subsetI dest*: *subsetD*)

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
by *blast*

lemma *Un-upper2*: $B \subseteq A \cup B$
by *blast*

lemma *Un-least*: $A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow A \cup B \subseteq C$
by *blast*

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
by *blast*

lemma *Int-lower2*: $A \cap B \subseteq B$
by *blast*

lemma *Int-greatest*: $C \subseteq A \Longrightarrow C \subseteq B \Longrightarrow C \subseteq A \cap B$
by *blast*

Set difference.

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

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

4.5.2 Equalities involving union, intersection, inclusion, etc.

{}.

lemma *Collect-const* [*simp*]: $\{s.\ P\} = (\mathit{if}\ P\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{\})$
— supersedes *Collect-False-empty*
by *auto*

lemma *subset-empty* [*simp*]: $(A \subseteq \{\}) = (A = \{\})$
by *blast*

lemma *not-psubset-empty* [*iff*]: $\neg\ (A < \{\})$
by (*unfold less-le*) *blast*

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

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

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

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

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

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

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

insert.

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

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

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

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

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

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

lemma *insert-subset* [*simp*]: (*insert x A* ⊆ *B*) = (*x* ∈ *B* & *A* ⊆ *B*)
by *blast*

lemma *mk-disjoint-insert*: *a* ∈ *A* ==> ∃ *B*. *A* = *insert a B* & *a* ∉ *B*
— use new *B* rather than *A* − {*a*} to avoid infinite unfolding
apply (*rule-tac x* = *A* − {*a*} **in** *exI*, *blast*)
done

lemma *insert-Collect*: *insert a* (*Collect P*) = {*u*. *u* ≠ *a* −−> *P u*}
by *auto*

lemma *UN-insert-distrib*: *u* ∈ *A* ==> (⋃ *x*∈*A*. *insert a* (*B x*)) = *insert a* (⋃ *x*∈*A*. *B x*)
by *blast*

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

lemma *insert-disjoint* [*simp*,*noatp*]:
(*insert a A* ∩ *B* = {}) = (*a* ∉ *B* ∧ *A* ∩ *B* = {})
({} = *insert a A* ∩ *B*) = (*a* ∉ *B* ∧ {} = *A* ∩ *B*)
by *auto*

lemma *disjoint-insert* [*simp*,*noatp*]:
(*B* ∩ *insert a A* = {}) = (*a* ∉ *B* ∧ *B* ∩ *A* = {})
({} = *A* ∩ *insert b B*) = (*b* ∉ *A* ∧ {} = *A* ∩ *B*)
by *auto*

image.

lemma *image-empty* [*simp*]: *f*‘{} = {}
by *blast*

lemma *image-insert* [*simp*]: *f* ‘ *insert a B* = *insert* (*f a*) (*f*‘*B*)
by *blast*

lemma *image-constant*: *x* ∈ *A* ==> (λ*x*. *c*) ‘ *A* = {*c*}
by *auto*

lemma *image-constant-conv*: (%*x*. *c*) ‘ *A* = (*if A* = {} *then* {} *else* {*c*})
by *auto*

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

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

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

lemma *image-Collect* [*noatp*]: $f ` \{x.\ P\ x\} = \{f\ x \mid x.\ P\ x\}$
— NOT suitable as a default simprule: the RHS isn't simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.
by *blast*

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

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

range.

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

lemma *range-composition* [*simp*]: $range\ (\lambda x.\ f\ (g\ x)) = f`range\ g$
by (*subst image-image*, *simp*)

Int

lemma *Int-absorb* [*simp*]: $A \cap A = A$
by *blast*

lemma *Int-left-absorb*: $A \cap (A \cap B) = A \cap B$
by *blast*

lemma *Int-commute*: $A \cap B = B \cap A$
by *blast*

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

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

lemmas *Int-ac* = *Int-assoc Int-left-absorb Int-commute Int-left-commute*
— Intersection is an AC-operator

lemma *Int-absorb1*: $B \subseteq A \Longrightarrow A \cap B = B$
by *blast*

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

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

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

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

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

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

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

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

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

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

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

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

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

Un.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Set complement

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

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

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

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

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

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

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

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

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

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

by *blast*

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

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

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

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

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

Union.

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

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

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

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

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

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

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

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

Inter.

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

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

by *blast*

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

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

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

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

UN and *INT*.

Basic identities:

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

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

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

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

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

lemma *INT-absorb*: $k \in I \Longrightarrow A\ k \cap (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ A\ i)$
by *blast*

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

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

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

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

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

by *blast*

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

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

lemma *INT-insert-distrib*:
$u \in A \Longrightarrow (\bigcap x{\in}A.\ insert\ a\ (B\ x)) = insert\ a\ (\bigcap x{\in}A.\ B\ x)$
by *blast*

lemma *Union-image-eq* [*simp*]: $\bigcup(B`A) = (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *image-Union*: $f\ `\ \bigcup S = (\bigcup x{\in}S.\ f\ `\ x)$
by *blast*

lemma *Inter-image-eq* [*simp*]: $\bigcap(B`A) = (\bigcap x{\in}A.\ B\ x)$
by *blast*

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

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

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

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

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

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

Distributive laws:

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

lemma *Int-Union2*: $\bigcup B \cap A = (\bigcup C{\in}B.\ C \cap A)$

by *blast*

lemma *Un-Union-image*: $(\bigcup x{\in}C.\ A\ x \cup B\ x) = \bigcup(A`C) \cup \bigcup(B`C)$
— Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
— Union of a family of unions
by *blast*

lemma *UN-Un-distrib*: $(\bigcup i{\in}I.\ A\ i \cup B\ i) = (\bigcup i{\in}I.\ A\ i) \cup (\bigcup i{\in}I.\ B\ i)$
— Equivalent version
by *blast*

lemma *Un-Inter*: $A \cup \bigcap B = (\bigcap C{\in}B.\ A \cup C)$
by *blast*

lemma *Int-Inter-image*: $(\bigcap x{\in}C.\ A\ x \cap B\ x) = \bigcap(A`C) \cap \bigcap(B`C)$
by *blast*

lemma *INT-Int-distrib*: $(\bigcap i{\in}I.\ A\ i \cap B\ i) = (\bigcap i{\in}I.\ A\ i) \cap (\bigcap i{\in}I.\ B\ i)$
— Equivalent version
by *blast*

lemma *Int-UN-distrib*: $B \cap (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ B \cap A\ i)$
— Halmos, Naive Set Theory, page 35.
by *blast*

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

lemma *Int-UN-distrib2*: $(\bigcup i{\in}I.\ A\ i) \cap (\bigcup j{\in}J.\ B\ j) = (\bigcup i{\in}I.\ \bigcup j{\in}J.\ A\ i \cap B\ j)$
by *blast*

lemma *Un-INT-distrib2*: $(\bigcap i{\in}I.\ A\ i) \cup (\bigcap j{\in}J.\ B\ j) = (\bigcap i{\in}I.\ \bigcap j{\in}J.\ A\ i \cup B\ j)$
by *blast*

Bounded quantifiers.

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

lemma *ball-Un*: $(\forall x \in A \cup B.\ P\ x) = ((\forall x{\in}A.\ P\ x)\ \&\ (\forall x{\in}B.\ P\ x))$
by *blast*

lemma *bex-Un*: $(\exists x \in A \cup B.\ P\ x) = ((\exists x{\in}A.\ P\ x)\ |\ (\exists x{\in}B.\ P\ x))$
by *blast*

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

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

Set difference.

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

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

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

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

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

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

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

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

lemma *Diff-insert0* [*simp*,*noatp*]: $x \notin A \Longrightarrow A - \mathit{insert}\ x\ B = A - B$
by *blast*

lemma *Diff-insert*: $A - \mathit{insert}\ a\ B = A - B - \{a\}$
— NOT SUITABLE FOR REWRITING since $\{a\} == \mathit{insert}\ a\ 0$
by *blast*

lemma *Diff-insert2*: $A - \mathit{insert}\ a\ B = A - \{a\} - B$
— NOT SUITABLE FOR REWRITING since $\{a\} == \mathit{insert}\ a\ 0$
by *blast*

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

lemma *insert-Diff1* [*simp*]: $x \in B \Longrightarrow \mathit{insert}\ x\ A - B = A - B$
by *blast*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Quantification over type *bool*.

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

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

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

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

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

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

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

Pow

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

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

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

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

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

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

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

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

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

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

Miscellany.

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

lemma *subset-iff*: $(A \subseteq B) = (\forall t.\ t \in A \longrightarrow t \in B)$
by *blast*

lemma *subset-iff-psubset-eq*: $(A \subseteq B) = ((A \subset B) \mid (A = B))$
by (*unfold less-le*) *blast*

lemma *all-not-in-conv* [*simp*]: $(\forall x.\ x \notin A) = (A = \{\})$
by *blast*

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

lemma *distinct-lemma*: $f\ x \neq f\ y \Longrightarrow x \neq y$
by *iprover*

Miniscoping: pushing in quantifiers and big Unions and Intersections.

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

lemma *INT-simps* [*simp*]:
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Int\ B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x)\ Int\ B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ Int\ B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A\ Int\ (INT\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x - B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x) - B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A - B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A - (UN\ x{:}C.\ B\ x))$
$!!a\ B\ C.\ (INT\ x{:}C.\ insert\ a\ (B\ x)) = insert\ a\ (INT\ x{:}C.\ B\ x)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Un\ B) = ((INT\ x{:}C.\ A\ x)\ Un\ B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ Un\ B\ x) = (A\ Un\ (INT\ x{:}C.\ B\ x))$

!!$A\ B.\ (INT\ x{:}\ Union\ A.\ B\ x) = (INT\ y{:}A.\ INT\ x{:}y.\ B\ x)$
!!$A\ B\ C.\ (INT\ z{:}\ UNION\ A\ B.\ C\ z) = (INT\ x{:}A.\ INT\ z{:}\ B(x).\ C\ z)$
!!$A\ B\ f.\ (INT\ x{:}f`A.\ B\ x) \quad = (INT\ a{:}A.\ B\ (f\ a))$
by *auto*

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

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

lemma *ball-conj-distrib*:
$(ALL\ x{:}A.\ P\ x\ \&\ Q\ x) = ((ALL\ x{:}A.\ P\ x)\ \&\ (ALL\ x{:}A.\ Q\ x))$
by *blast*

lemma *bex-disj-distrib*:
$(EX\ x{:}A.\ P\ x\ |\ Q\ x) = ((EX\ x{:}A.\ P\ x)\ |\ (EX\ x{:}A.\ Q\ x))$
by *blast*

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:
!!$a\ B\ C.\ insert\ a\ (UN\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ \{a\}\ else\ (UN\ x{:}C.\ insert\ a\ (B\ x)))$
!!$A\ B\ C.\ (UN\ x{:}C.\ A\ x)\ Un\ B \quad = (if\ C{=}\{\}\ then\ B\ else\ (UN\ x{:}C.\ A\ x\ Un\ B))$
!!$A\ B\ C.\ A\ Un\ (UN\ x{:}C.\ B\ x) \quad = (if\ C{=}\{\}\ then\ A\ else\ (UN\ x{:}C.\ A\ Un\ B\ x))$
!!$A\ B\ C.\ ((UN\ x{:}C.\ A\ x)\ Int\ B) = (UN\ x{:}C.\ A\ x\ Int\ B)$
!!$A\ B\ C.\ (A\ Int\ (UN\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A\ Int\ B\ x)$

!!*A B C*. ((*UN x*:*C*. *A x*) − *B*) = (*UN x*:*C*. *A x* − *B*)
!!*A B C*. (*A* − (*INT x*:*C*. *B x*)) = (*UN x*:*C*. *A* − *B x*)
!!*A B*. (*UN y*:*A*. *UN x*:*y*. *B x*) = (*UN x*: *Union A*. *B x*)
!!*A B C*. (*UN x*:*A*. *UN z*: *B*(*x*). *C z*) = (*UN z*: *UNION A B*. *C z*)
!!*A B f*. (*UN a*:*A*. *B* (*f a*)) = (*UN x*:*f*‘*A*. *B x*)
by *auto*

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

4.5.3 Monotonicity of various operations

lemma *image-mono*: *A* ⊆ *B* ==> *f*‘*A* ⊆ *f*‘*B*
by *blast*

lemma *Pow-mono*: *A* ⊆ *B* ==> *Pow A* ⊆ *Pow B*
by *blast*

lemma *Union-mono*: *A* ⊆ *B* ==> ⋃ *A* ⊆ ⋃ *B*
by *blast*

lemma *Inter-anti-mono*: *B* ⊆ *A* ==> ⋂ *A* ⊆ ⋂ *B*
by *blast*

lemma *UN-mono*:
A ⊆ *B* ==> (!!*x*. *x* ∈ *A* ==> *f x* ⊆ *g x*) ==>
(⋃ *x*∈*A*. *f x*) ⊆ (⋃ *x*∈*B*. *g x*)
by (*blast dest*: *subsetD*)

lemma *INT-anti-mono*:
B ⊆ *A* ==> (!!*x*. *x* ∈ *A* ==> *f x* ⊆ *g x*) ==>
(⋂ *x*∈*A*. *f x*) ⊆ (⋂ *x*∈*A*. *g x*)
— The last inclusion is POSITIVE!
by (*blast dest*: *subsetD*)

lemma *insert-mono*: *C* ⊆ *D* ==> *insert a C* ⊆ *insert a D*
by *blast*

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

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

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

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

Monotonicity of implications.

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

lemma *conj-mono*: *P1* $-->$ *Q1* $==>$ *P2* $-->$ *Q2* $==>$ (*P1* & *P2*) $-->$ (*Q1* & *Q2*)
by *iprover*

lemma *disj-mono*: *P1* $-->$ *Q1* $==>$ *P2* $-->$ *Q2* $==>$ (*P1* | *P2*) $-->$ (*Q1* | *Q2*)
by *iprover*

lemma *imp-mono*: *Q1* $-->$ *P1* $==>$ *P2* $-->$ *Q2* $==>$ (*P1* $-->$ *P2*) $-->$ (*Q1* $-->$ *Q2*)
by *iprover*

lemma *imp-refl*: $P --> P$ **..**

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

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

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

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

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

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

lemma *eq-to-mono2*: $a = b \Longrightarrow c = d \Longrightarrow a^{\sim} \longrightarrow b^{\sim} \Longrightarrow c^{\sim} \longrightarrow d^{\sim}$
by *iprover*

4.6 Inverse image of a function

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

4.6.1 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*

4.6.2 Equations

lemma *vimage-empty* [*simp*]: $f\ -`\ \{\} = \{\}$
by *blast*

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

lemma *vimage-Un* [*simp*]: $f\ -`\ (A\ Un\ B) = (f\ -`\ A)\ Un\ (f\ -`\ B)$
by *blast*

lemma *vimage-Int* [*simp*]: $f\ -`\ (A\ Int\ B) = (f\ -`\ A)\ Int\ (f\ -`\ B)$
by *fast*

lemma *vimage-Union*: $f\ -`\ (Union\ A) = (UN\ X{:}A.\ f\ -`\ X)$
by *blast*

lemma *vimage-UN*: $f-`(UN\ x{:}A.\ B\ x) = (UN\ x{:}A.\ f\ -`\ B\ x)$
by *blast*

lemma *vimage-INT*: $f-`(INT\ x{:}A.\ B\ x) = (INT\ x{:}A.\ f\ -`\ B\ x)$
by *blast*

lemma *vimage-Collect-eq* [*simp*]: $f\ -`\ Collect\ P = \{y.\ P\ (f\ y)\}$
by *blast*

lemma *vimage-Collect*: $(!!x.\ P\ (f\ x) = Q\ x) ==> f\ -`\ (Collect\ P) = Collect\ Q$
by *blast*

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

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

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

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

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

lemma *vimage-image-eq* [*noatp*]: $f\ -`\ (f\ `\ A) = \{y.\ EX\ x{:}A.\ f\ x = f\ y\}$
by (*blast intro*: *sym*)

lemma *image-vimage-subset*: $f\ `\ (f\ -`\ A) <= A$
by *blast*

lemma *image-vimage-eq* [*simp*]: $f\ `\ (f\ -`\ A) = A\ Int\ range\ f$
by *blast*

lemma *image-Int-subset*: $f`(A\ Int\ B) <= f`A\ Int\ f`B$
by *blast*

lemma *image-diff-subset*: $f`A - f`B <= f`(A - B)$
by *blast*

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

4.7 Getting the Contents of a Singleton Set

definition
contents :: *'a set* ⇒ *'a*
where
contents X = (*THE x*. *X* = {*x*})

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

4.8 Transitivity rules for calculational reasoning

lemma *set-rev-mp*: *x*:*A* ==> *A* ⊆ *B* ==> *x*:*B*
by (*rule subsetD*)

lemma *set-mp*: *A* ⊆ *B* ==> *x*:*A* ==> *x*:*B*
by (*rule subsetD*)

lemmas *basic-trans-rules* [*trans*] =
order-trans-rules set-rev-mp set-mp

4.9 Dense orders

class *dense-linear-order* = *linorder* +
assumes *gt-ex*: ∃ *y*. *x* < *y*
and *lt-ex*: ∃ *y*. *y* < *x*
and *dense*: *x* < *y* ⟹ (∃ *z*. *x* < *z* ∧ *z* < *y*)

begin

lemma *interval-empty-iff*:
{*y*. *x* < *y* ∧ *y* < *z*} = {} ⟷ ¬ *x* < *z*
by (*auto dest*: *dense*)

end

4.10 Least value operator

lemma *Least-mono*:
mono (*f*::*'a*::*order* => *'b*::*order*) ==> *EX x*:*S*. *ALL y*:*S*. *x* <= *y*
==> (*LEAST y*. *y* : *f* ‘ *S*) = *f* (*LEAST x*. *x* : *S*)
— Courtesy of Stephan Merz
apply *clarify*
apply (*erule-tac P* = %*x*. *x* : *S* **in** *LeastI2-order*, *fast*)
apply (*rule LeastI2-order*)
apply (*auto elim*: *monoD intro*!: *order-antisym*)
done

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

4.11 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

5 Fun: Notions about functions

theory *Fun*
imports *Set*
begin

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

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

lemma *apply-inverse*:
$f\ x = u \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow g\ (f\ x) = x) \Longrightarrow P\ x \Longrightarrow x = g\ u$
by *auto*

5.1 The Identity Function *id*

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

lemma *id-apply* [*simp*]: $id\ x = x$
by (*simp add*: *id-def*)

lemma *image-ident* [*simp*]: $(\%x.\ x)\ `\ Y = Y$

by *blast*

lemma *image-id* [*simp*]: *id ‘ Y = Y*
by (*simp add*: *id-def*)

lemma *vimage-ident* [*simp*]: *(%x. x) −‘ Y = Y*
by *blast*

lemma *vimage-id* [*simp*]: *id −‘ A = A*
by (*simp add*: *id-def*)

5.2 The Composition Operator $f \circ g$

definition
comp :: *('b ⇒ 'c) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ 'c* (**infixl** *o 55*)
where
f o g = (λx. f (g x))

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

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

compatibility

lemmas *o-def = comp-def*

lemma *o-apply* [*simp*]: *(f o g) x = f (g x)*
by (*simp add*: *comp-def*)

lemma *o-assoc*: *f o (g o h) = f o g o h*
by (*simp add*: *comp-def*)

lemma *id-o* [*simp*]: *id o g = g*
by (*simp add*: *comp-def*)

lemma *o-id* [*simp*]: *f o id = f*
by (*simp add*: *comp-def*)

lemma *image-compose*: *(f o g) ‘ r = f‘(g‘r)*
by (*simp add*: *comp-def*, *blast*)

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

5.3 The Forward Composition Operator *fcomp*

definition
fcomp :: *('a ⇒ 'b) ⇒ ('b ⇒ 'c) ⇒ 'a ⇒ 'c* (**infixl** *o> 60*)

where
f o> g = (λx. g (f x))

lemma *fcomp-apply*: *(f o> g) x = g (f x)*
by (*simp add*: *fcomp-def*)

lemma *fcomp-assoc*: *(f o> g) o> h = f o> (g o> h)*
by (*simp add*: *fcomp-def*)

lemma *id-fcomp* [*simp*]: *id o> g = g*
by (*simp add*: *fcomp-def*)

lemma *fcomp-id* [*simp*]: *f o> id = f*
by (*simp add*: *fcomp-def*)

no-notation *fcomp* (**infixl** *o>* *60*)

5.4 Injectivity and Surjectivity

constdefs
inj-on :: [*'a => 'b*, *'a set*] => *bool* — injective
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

definition
bij-betw :: *('a => 'b) => 'a set => 'b set => bool* **where** — bijective
bij-betw f A B ⟷ inj-on f A & f ' A = B

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

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

lemma *injI*:
assumes ⋀*x y. f x = f y ⟹ x = y*
shows *inj f*
using *assms* **unfolding** *inj-on-def* **by** *auto*

For Proofs in *Tools/datatype-rep-proofs*

lemma *datatype-injI*:
(!! *x. ALL y. f(x) = f(y) --> x=y*) ==> *inj(f)*
by (*simp add*: *inj-on-def*)

theorem *range-ex1-eq*: *inj f ⟹ b : range f = (EX! x. b = f x)*

by (*unfold inj-on-def*, *blast*)

lemma *injD*: [| *inj*(*f*); *f*(*x*) = *f*(*y*) |] ==> *x*=*y*
by (*simp add*: *inj-on-def*)

lemma *inj-eq*: *inj*(*f*) ==> (*f*(*x*) = *f*(*y*)) = (*x*=*y*)
by (*force simp add*: *inj-on-def*)

lemma *inj-on-id*[*simp*]: *inj-on id A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-id2*[*simp*]: *inj-on* (%*x*. *x*) *A*
by (*simp add*: *inj-on-def*)

lemma *surj-id*[*simp*]: *surj id*
by (*simp add*: *surj-def*)

lemma *bij-id*[*simp*]: *bij id*
by (*simp add*: *bij-def inj-on-id surj-id*)

lemma *inj-onI*:
(!! *x y*. [| *x*:*A*; *y*:*A*; *f*(*x*) = *f*(*y*) |] ==> *x*=*y*) ==> *inj-on f A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-inverseI*: (!!*x*. *x*:*A* ==> *g*(*f*(*x*)) = *x*) ==> *inj-on f A*
by (*auto dest*: *arg-cong* [*of concl*: *g*] *simp add*: *inj-on-def*)

lemma *inj-onD*: [| *inj-on f A*; *f*(*x*)=*f*(*y*); *x*:*A*; *y*:*A* |] ==> *x*=*y*
by (*unfold inj-on-def*, *blast*)

lemma *inj-on-iff*: [| *inj-on f A*; *x*:*A*; *y*:*A* |] ==> (*f*(*x*)=*f*(*y*)) = (*x*=*y*)
by (*blast dest*!: *inj-onD*)

lemma *comp-inj-on*:
[| *inj-on f A*; *inj-on g* (*f*‘*A*) |] ==> *inj-on* (*g o f*) *A*
by (*simp add*: *comp-def inj-on-def*)

lemma *inj-on-imageI*: *inj-on* (*g o f*) *A* ⟹ *inj-on g* (*f* ‘ *A*)
apply(*simp add*:*inj-on-def image-def*)
apply *blast*
done

lemma *inj-on-image-iff*: ⟦ *ALL x*:*A*. *ALL y*:*A*. (*g*(*f x*) = *g*(*f y*)) = (*g x* = *g y*);
inj-on f A ⟧ ⟹ *inj-on g* (*f* ‘ *A*) = *inj-on g A*
apply(*unfold inj-on-def*)
apply *blast*
done

lemma *inj-on-contraD*: [| *inj-on f A*; ~*x=y*; *x:A*; *y:A* |] ==> ~ *f(x)=f(y)*
by (*unfold inj-on-def*, *blast*)

lemma *inj-singleton*: *inj* (%*s*. {*s*})
by (*simp add*: *inj-on-def*)

lemma *inj-on-empty*[*iff*]: *inj-on f* {}
by(*simp add*: *inj-on-def*)

lemma *subset-inj-on*: [| *inj-on f B*; *A* <= *B* |] ==> *inj-on f A*
by (*unfold inj-on-def*, *blast*)

lemma *inj-on-Un*:
inj-on f (*A Un B*) =
(*inj-on f A* & *inj-on f B* & *f*‘(*A*−*B*) *Int f*‘(*B*−*A*) = {})
apply(*unfold inj-on-def*)
apply (*blast intro*:*sym*)
done

lemma *inj-on-insert*[*iff*]:
inj-on f (*insert a A*) = (*inj-on f A* & *f a* ~: *f*‘(*A*−{*a*}))
apply(*unfold inj-on-def*)
apply (*blast intro*:*sym*)
done

lemma *inj-on-diff*: *inj-on f A* ==> *inj-on f* (*A*−*B*)
apply(*unfold inj-on-def*)
apply (*blast*)
done

lemma *surjI*: (!! *x*. *g*(*f x*) = *x*) ==> *surj g*
apply (*simp add*: *surj-def*)
apply (*blast intro*: *sym*)
done

lemma *surj-range*: *surj f* ==> *range f* = *UNIV*
by (*auto simp add*: *surj-def*)

lemma *surjD*: *surj f* ==> *EX x*. *y* = *f x*
by (*simp add*: *surj-def*)

lemma *surjE*: *surj f* ==> (!!*x*. *y* = *f x* ==> *C*) ==> *C*
by (*simp add*: *surj-def*, *blast*)

lemma *comp-surj*: [| *surj f*; *surj g* |] ==> *surj* (*g o f*)
apply (*simp add*: *comp-def surj-def*, *clarify*)
apply (*drule-tac x* = *y* **in** *spec*, *clarify*)
apply (*drule-tac x* = *x* **in** *spec*, *blast*)
done

lemma *bijI*: [| *inj f*; *surj f* |] ==> *bij f*
by (*simp add*: *bij-def*)

lemma *bij-is-inj*: *bij f* ==> *inj f*
by (*simp add*: *bij-def*)

lemma *bij-is-surj*: *bij f* ==> *surj f*
by (*simp add*: *bij-def*)

lemma *bij-betw-imp-inj-on*: *bij-betw f A B* ⟹ *inj-on f A*
by (*simp add*: *bij-betw-def*)

lemma *bij-betw-inv*: **assumes** *bij-betw f A B* **shows** *EX g*. *bij-betw g B A*
proof −
have *i*: *inj-on f A* **and** *s*: *f* ‘ *A* = *B*
using *assms* **by**(*auto simp*:*bij-betw-def*)
let *?P* = %*b a*. *a*:*A* ∧ *f a* = *b* **let** *?g* = %*b*. *The* (*?P b*)
{ fix *a b* **assume** *P*: *?P b a*
hence *ex1*: ∃ *a*. *?P b a* **using** *s* **unfolding** *image-def* **by** *blast*
hence *uex1*: ∃!*a*. *?P b a* **by**(*blast dest*:*inj-onD*[*OF i*])
hence *?g b* = *a* **using** *the1-equality*[*OF uex1*, *OF P*] *P* **by** *simp*
} note *g* = *this*
have *inj-on ?g B*
proof(*rule inj-onI*)
fix *x y* **assume** *x*:*B y*:*B ?g x* = *?g y*
from *s* ‹*x*:*B*› **obtain** *a1* **where** *a1*: *?P x a1* **unfolding** *image-def* **by** *blast*
from *s* ‹*y*:*B*› **obtain** *a2* **where** *a2*: *?P y a2* **unfolding** *image-def* **by** *blast*
from *g*[*OF a1*] *a1 g*[*OF a2*] *a2* ‹*?g x* = *?g y*› **show** *x*=*y* **by** *simp*
qed
moreover have *?g* ‘ *B* = *A*
proof(*auto simp*:*image-def*)
fix *b* **assume** *b*:*B*
with *s* **obtain** *a* **where** *P*: *?P b a* **unfolding** *image-def* **by** *blast*
thus *?g b* ∈ *A* **using** *g*[*OF P*] **by** *auto*
next
fix *a* **assume** *a*:*A*
then obtain *b* **where** *P*: *?P b a* **using** *s* **unfolding** *image-def* **by** *blast*
then have *b*:*B* **using** *s* **unfolding** *image-def* **by** *blast*
with *g*[*OF P*] **show** ∃ *b*∈*B*. *a* = *?g b* **by** *blast*
qed
ultimately show *?thesis* **by**(*auto simp*:*bij-betw-def*)
qed

lemma *surj-image-vimage-eq*: *surj f* ==> *f* ‘ (*f* −‘ *A*) = *A*
by (*simp add*: *surj-range*)

lemma *inj-vimage-image-eq*: *inj f* ==> *f* −‘ (*f* ‘ *A*) = *A*
by (*simp add*: *inj-on-def*, *blast*)

lemma *vimage-subsetD*: *surj f ==> f −‘ B <= A ==> B <= f ‘ A*
apply (*unfold surj-def*)
apply (*blast intro*: *sym*)
done

lemma *vimage-subsetI*: *inj f ==> B <= f ‘ A ==> f −‘ B <= A*
by (*unfold inj-on-def*, *blast*)

lemma *vimage-subset-eq*: *bij f ==> (f −‘ B <= A) = (B <= f ‘ A)*
apply (*unfold bij-def*)
apply (*blast del*: *subsetI intro*: *vimage-subsetI vimage-subsetD*)
done

lemma *inj-on-image-Int*:
[| *inj-on f C*; *A<=C*; *B<=C* |] *==> f‘(A Int B) = f‘A Int f‘B*
apply (*simp add*: *inj-on-def*, *blast*)
done

lemma *inj-on-image-set-diff*:
[| *inj-on f C*; *A<=C*; *B<=C* |] *==> f‘(A−B) = f‘A − f‘B*
apply (*simp add*: *inj-on-def*, *blast*)
done

lemma *image-Int*: *inj f ==> f‘(A Int B) = f‘A Int f‘B*
by (*simp add*: *inj-on-def*, *blast*)

lemma *image-set-diff*: *inj f ==> f‘(A−B) = f‘A − f‘B*
by (*simp add*: *inj-on-def*, *blast*)

lemma *inj-image-mem-iff*: *inj f ==> (f a : f‘A) = (a : A)*
by (*blast dest*: *injD*)

lemma *inj-image-subset-iff*: *inj f ==> (f‘A <= f‘B) = (A<=B)*
by (*simp add*: *inj-on-def*, *blast*)

lemma *inj-image-eq-iff*: *inj f ==> (f‘A = f‘B) = (A = B)*
by (*blast dest*: *injD*)

lemma *image-INT*:
[| *inj-on f C*; *ALL x:A. B x <= C*; *j:A* |]
==> f ‘ (INTER A B) = (INT x:A. f ‘ B x)
apply (*simp add*: *inj-on-def*, *blast*)
done

lemma *bij-image-INT*: *bij f ==> f ‘ (INTER A B) = (INT x:A. f ‘ B x)*
apply (*simp add*: *bij-def*)

apply (*simp add*: *inj-on-def surj-def*, *blast*)
done

lemma *surj-Compl-image-subset*: *surj f* ==> $-(f`A) <= f`(-A)$
by (*auto simp add*: *surj-def*)

lemma *inj-image-Compl-subset*: *inj f* ==> $f`(-A) <= -(f`A)$
by (*auto simp add*: *inj-on-def*)

lemma *bij-image-Compl-eq*: *bij f* ==> $f`(-A) = -(f`A)$
apply (*simp add*: *bij-def*)
apply (*rule equalityI*)
apply (*simp-all* (*no-asm-simp*) *add*: *inj-image-Compl-subset surj-Compl-image-subset*)
done

5.5 Function Updating

constdefs
fun-upd :: $('a => 'b) => 'a => 'b => ('a => 'b)$
fun-upd f a b == % *x*. *if x=a then b else f x*

nonterminals
updbinds updbind
syntax
-updbind :: $['a, 'a] =>$ *updbind* ((*2*- :=/ -))
:: *updbind* => *updbinds* (-)
-updbinds:: [*updbind*, *updbinds*] => *updbinds* (-,/ -)
-Update :: $['a,$ *updbinds*$] => 'a$ (-/'((-)') [*1000*,*0*] *900*)

translations
-Update f (*-updbinds b bs*) == *-Update* (*-Update f b*) *bs*
f(*x*:=*y*) == *fun-upd f x y*

lemma *fun-upd-idem-iff*: $(f(x:=y) = f) = (f\ x = y)$
apply (*simp add*: *fun-upd-def*, *safe*)
apply (*erule subst*)
apply (*rule-tac* [*2*] *ext*, *auto*)
done

lemmas *fun-upd-idem* = *fun-upd-idem-iff* [*THEN iffD2*, *standard*]

lemmas *fun-upd-triv* = *refl* [*THEN fun-upd-idem*]
declare *fun-upd-triv* [*iff*]

lemma *fun-upd-apply* [*simp*]: $(f(x:=y))z = (if\ z=x\ then\ y\ else\ f\ z)$

by (*simp add*: *fun-upd-def*)

lemma *fun-upd-same*: $(f(x:=y))\ x = y$
by *simp*

lemma *fun-upd-other*: $z^{\sim}=x \Longrightarrow (f(x:=y))\ z = f\ z$
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*: $a \sim= c \Longrightarrow (m(a:=b))(c:=d) = (m(c:=d))(a:=b)$
by (*rule ext*, *auto*)

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

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

5.6 *override-on*

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

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

lemma *override-on-apply-notin*[*simp*]: $a \sim: A \Longrightarrow (\mathit{override\text{-}on}\ f\ g\ A)\ a = f\ a$
by(*simp add*:*override-on-def*)

lemma *override-on-apply-in*[*simp*]: $a : A \Longrightarrow (\mathit{override\text{-}on}\ f\ g\ A)\ a = g\ a$
by(*simp add*:*override-on-def*)

5.7 *swap*

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

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

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

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

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

lemma *inj-on-swap-iff* [*simp*]:
assumes *A*: *a* ∈ *A b* ∈ *A* **shows** *inj-on* (*swap a b f*) *A* = *inj-on f A*
proof
assume *inj-on* (*swap a b f*) *A*
with *A* **have** *inj-on* (*swap a b* (*swap a b f*)) *A*
by (*iprover intro*: *inj-on-imp-inj-on-swap*)
thus *inj-on f A* **by** *simp*
next
assume *inj-on f A*
with *A* **show** *inj-on* (*swap a b f*) *A* **by** (*iprover intro*: *inj-on-imp-inj-on-swap*)
qed

lemma *surj-imp-surj-swap*: *surj f* ==> *surj* (*swap a b f*)
apply (*simp add*: *surj-def swap-def*, *clarify*)
apply (*rule-tac P* = *y* = *f b* **in** *case-split-thm*, *blast*)
apply (*rule-tac P* = *y* = *f a* **in** *case-split-thm*, *auto*)
— We don't yet have *case-tac*
done

lemma *surj-swap-iff* [*simp*]: *surj* (*swap a b f*) = *surj f*
proof
assume *surj* (*swap a b f*)
hence *surj* (*swap a b* (*swap a b f*)) **by** (*rule surj-imp-surj-swap*)
thus *surj f* **by** *simp*
next
assume *surj f*
thus *surj* (*swap a b f*) **by** (*rule surj-imp-surj-swap*)
qed

lemma *bij-swap-iff*: *bij* (*swap a b f*) = *bij f*
by (*simp add*: *bij-def*)

5.8 Proof tool setup

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

simproc-setup *fun-upd2* (*f*(*v* := *w*, *x* := *y*)) = ⟪ *fn* - =>
let
fun gen-fun-upd NONE T - - = *NONE*
| *gen-fun-upd* (*SOME f*) *T x y* = *SOME* (*Const* (@{*const-name fun-upd*}, *T*)
$ *f* $ *x* $ *y*)
fun dest-fun-T1 (*Type* (-, *T* :: *Ts*)) = *T*

```
  fun find-double (t as Const (@{const-name fun-upd},T) $ f $ x $ y) =
    let
      fun find (Const (@{const-name fun-upd},T) $ g $ v $ w) =
            if v aconv x then SOME g else gen-fun-upd (find g) T v w
        | find t = NONE
    in (dest-fun-T1 T, gen-fun-upd (find f) T x y) end

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

5.9 Code generator setup

types-code

```
  fun  ((- ->/ -))
```

attach (*term-of*) ⟨⟨

```
fun term-of-fun-type - aT - bT - = Free (<function>, aT --> bT);
```

⟩⟩

attach (*test*) ⟨⟨

```
fun gen-fun-type aF aT bG bT i =
  let
    val tab = ref [];
    fun mk-upd (x, (-, y)) t = Const (Fun.fun-upd,
      (aT --> bT) --> aT --> bT --> aT --> bT) $ t $ aF x $ y ()
  in
    (fn x =>
       case AList.lookup op = (!tab) x of
         NONE =>
           let val p as (y, -) = bG i
           in (tab := (x, p) :: !tab; y) end
       | SOME (y, -) => y,
     fn () => Basics.fold mk-upd (!tab) (Const (arbitrary, aT --> bT)))
  end;
```

⟩⟩

code-const *op* ∘

(*SML* **infixl** *5 o*)
(*Haskell* **infixr** *9* .)

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

end

6 Lattices: Abstract lattices

theory *Lattices*
imports *Fun*
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*: $y \sqsubseteq x \Longrightarrow z \sqsubseteq x \Longrightarrow y \sqcup z \sqsubseteq x$
begin

Dual lattice

lemma *dual-lattice*:
lower-semilattice (*op* $\geq$) (*op* $>$) *sup*
by *unfold-locales*
(*auto simp add*: *sup-least*)

end

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*)
from *assms* **show** $a \sqsubseteq x$ **.**
show $a \sqcap b \sqsubseteq a$ **by** *simp*
qed
lemmas (**in** −) [*rule del*] = *le-infI1*

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

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

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

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

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

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

end

context *upper-semilattice*
begin

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

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

lemmas (**in** −) [*rule del*] = *le-supI2*

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

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

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

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

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

end

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*: $\bigsqcap A = \bigsqcup \{b.\ \forall a \in A.\ b \leq a\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Sup-Inf*: $\bigsqcup A = \bigsqcap \{b.\ \forall a \in A.\ a \leq b\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Inf-Univ*: $\bigsqcap UNIV = \bigsqcup \{\}$
unfolding *Sup-Inf* **by** *auto*

lemma *Sup-Univ*: $\bigsqcup UNIV = \bigsqcap \{\}$
unfolding *Inf-Sup* **by** *auto*

lemma *Inf-insert*: $\bigsqcap$ *insert a A* = *a* $\sqcap$ $\bigsqcap$ *A*
by (*auto intro*: *antisym Inf-greatest Inf-lower*)

lemma *Sup-insert*: $\bigsqcup$ *insert a A* = *a* $\sqcup$ $\bigsqcup$ *A*
by (*auto intro*: *antisym Sup-least Sup-upper*)

lemma *Inf-singleton* [*simp*]:
$\bigsqcap \{a\} = a$
by (*auto intro*: *antisym Inf-lower Inf-greatest*)

lemma *Sup-singleton* [*simp*]:
$\bigsqcup \{a\} = a$
by (*auto intro*: *antisym Sup-upper Sup-least*)

lemma *Inf-insert-simp*:
$\bigsqcap$ *insert a A* = (*if A* = {} *then a else a* $\sqcap$ $\bigsqcap$ *A*)
by (*cases A* = {}) (*simp-all*, *simp add*: *Inf-insert*)

lemma *Sup-insert-simp*:
$\bigsqcup$ *insert a A* = (*if A* = {} *then a else a* $\sqcup$ $\bigsqcup$ *A*)
by (*cases A* = {}) (*simp-all*, *simp add*: *Sup-insert*)

lemma *Inf-binary*:
$\bigsqcap \{a, b\} = a \sqcap b$
by (*simp add*: *Inf-insert-simp*)

lemma *Sup-binary*:
$\bigsqcup \{a, b\} = a \sqcup b$
by (*simp add*: *Sup-insert-simp*)

definition
top :: *'a* **where**
top = $\bigsqcap$ {}

definition
bot :: *'a* **where**
bot = $\bigsqcup$ {}

lemma *top-greatest* [*simp*]: *x* $\leq$ *top*
by (*unfold top-def*, *rule Inf-greatest*, *simp*)

lemma *bot-least* [*simp*]: *bot* $\leq$ *x*
by (*unfold bot-def*, *rule Sup-least*, *simp*)

definition
SUPR :: *'b set* $\Rightarrow$ (*'b* $\Rightarrow$ *'a*) $\Rightarrow$ *'a*
where
SUPR A f == $\bigsqcup$ (*f* ' *A*)

definition
INFI :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a*
where
INFI A f == ⨅ (*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
⟫

context *complete-lattice*
begin

lemma *le-SUPI*: *i* : *A* ⟹ *M i* ≤ (*SUP i*:*A. M i*)
by (*auto simp add*: *SUPR-def intro*: *Sup-upper*)

lemma *SUP-leI*: (⋀*i. i* : *A* ⟹ *M i* ≤ *u*) ⟹ (*SUP i*:*A. M i*) ≤ *u*
by (*auto simp add*: *SUPR-def intro*: *Sup-least*)

lemma *INF-leI*: *i* : *A* ⟹ (*INF i*:*A. M i*) ≤ *M i*
by (*auto simp add*: *INFI-def intro*: *Inf-lower*)

lemma *le-INFI*: (⋀*i. i* : *A* ⟹ *u* ≤ *M i*) ⟹ *u* ≤ (*INF i*:*A. M i*)

by (*auto simp add*: *INFI-def intro*: *Inf-greatest*)

lemma *SUP-const*[*simp*]: $A \neq \{\} \Longrightarrow (SUP\ i{:}A.\ M) = M$
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

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

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

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

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

end

instantiation *bool* :: *complete-lattice*
begin

definition
Inf-bool-def: $\bigsqcap A \longleftrightarrow (\forall x{\in}A.\ x)$

definition
Sup-bool-def: $\bigsqcup A \longleftrightarrow (\exists x{\in}A.\ x)$

instance
by *intro-classes* (*auto simp add*: *Inf-bool-def Sup-bool-def le-bool-def*)

end

lemma *Inf-empty-bool* [*simp*]:
$\bigsqcap \{\}$
unfolding *Inf-bool-def* **by** *auto*

lemma *not-Sup-empty-bool* [*simp*]:
$\neg$ *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 Fun as lattice

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

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

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

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

end

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

instantiation *fun* :: (*type*, *complete-lattice*) *complete-lattice*
begin

definition
Inf-fun-def [*code func del*]: $\bigsqcap A = (\lambda x.\ \bigsqcap \{y.\ \exists f{\in}A.\ y = f\ x\})$

definition
Sup-fun-def [*code func del*]: $\bigsqcup A = (\lambda x.\ \bigsqcup \{y.\ \exists f{\in}A.\ y = f\ x\})$

instance
by *intro-classes*
(*auto simp add*: *Inf-fun-def Sup-fun-def le-fun-def*
intro: *Inf-lower Sup-upper Inf-greatest Sup-least*)

end

lemma *Inf-empty-fun*:
$\bigsqcap \{\} = (\lambda\text{-}.\ \bigsqcap \{\})$

by *rule* (*auto simp add*: *Inf-fun-def*)

lemma *Sup-empty-fun*:
$\bigsqcup\{\} = (\lambda\text{-}.\ \bigsqcup\{\})$
by *rule* (*auto simp add*: *Sup-fun-def*)

lemma *top-fun-eq*: $top = (\lambda x.\ top)$
by (*iprover intro*!: *order-antisym le-funI top-greatest*)

lemma *bot-fun-eq*: $bot = (\lambda x.\ bot)$
by (*iprover intro*!: *order-antisym le-funI bot-least*)

6.8 Set as lattice

lemma *inf-set-eq*: $A \sqcap B = A \cap B$
apply (*rule subset-antisym*)
apply (*rule Int-greatest*)
apply (*rule inf-le1*)
apply (*rule inf-le2*)
apply (*rule inf-greatest*)
apply (*rule Int-lower1*)
apply (*rule Int-lower2*)
done

lemma *sup-set-eq*: $A \sqcup B = A \cup B$
apply (*rule subset-antisym*)
apply (*rule sup-least*)
apply (*rule Un-upper1*)
apply (*rule Un-upper2*)
apply (*rule Un-least*)
apply (*rule sup-ge1*)
apply (*rule sup-ge2*)
done

lemma *mono-Int*: $mono\ f \Longrightarrow f\ (A \cap B) \subseteq f\ A \cap f\ B$
apply (*fold inf-set-eq sup-set-eq*)
apply (*erule mono-inf*)
done

lemma *mono-Un*: $mono\ f \Longrightarrow f\ A \cup f\ B \subseteq f\ (A \cup B)$
apply (*fold inf-set-eq sup-set-eq*)
apply (*erule mono-sup*)
done

lemma *Inf-set-eq*: $\bigsqcap S = \bigcap S$
apply (*rule subset-antisym*)
apply (*rule Inter-greatest*)
apply (*erule Inf-lower*)
apply (*rule Inf-greatest*)

apply (*erule Inter-lower*)
done

lemma *Sup-set-eq*: $\bigsqcup S = \bigcup S$
apply (*rule subset-antisym*)
apply (*rule Sup-least*)
apply (*erule Union-upper*)
apply (*rule Union-least*)
apply (*erule Sup-upper*)
done

lemma *top-set-eq*: *top* = *UNIV*
by (*iprover intro*!: *subset-antisym subset-UNIV top-greatest*)

lemma *bot-set-eq*: *bot* = {}
by (*iprover intro*!: *subset-antisym empty-subsetI bot-least*)

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* ∈ *A*

and *Rep-inverse*: *Abs* (*Rep* *x*) = *x*
and *Abs-inverse*: *y* ∈ *A* ==> *Rep* (*Abs* *y*) = *y*
— This will be axiomatized for each typedef!
begin

lemma *Rep-inject*:
(*Rep* *x* = *Rep* *y*) = (*x* = *y*)
proof
assume *Rep* *x* = *Rep* *y*
then have *Abs* (*Rep* *x*) = *Abs* (*Rep* *y*) **by** (*simp only*:)
moreover have *Abs* (*Rep* *x*) = *x* **by** (*rule Rep-inverse*)
moreover have *Abs* (*Rep* *y*) = *y* **by** (*rule Rep-inverse*)
ultimately show *x* = *y* **by** *simp*
next
assume *x* = *y*
thus *Rep* *x* = *Rep* *y* **by** (*simp only*:)
qed

lemma *Abs-inject*:
assumes *x*: *x* ∈ *A* **and** *y*: *y* ∈ *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* ∈ *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*: $y \in A$
    and *hyp*: $!!x.\ P\ (Rep\ x)$
  shows $P\ y$
proof −
  have $P\ (Rep\ (Abs\ y))$ **by** (*rule hyp*)
  moreover from *y* **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
  ultimately show $P\ y$ **by** *simp*
qed

lemma *Abs-induct* [*induct type*]:
  assumes *r*: $!!y.\ y \in A \Longrightarrow P\ (Abs\ y)$
  shows $P\ x$
proof −
  have $Rep\ x \in A$ **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 ⟪
  TypedefPackage.setup
  #> *TypecopyPackage.setup*
  #> *TypedefCodegen.setup*
⟫

This class is just a workaround for classes without parameters; it shall disappear as soon as possible.

class *itself* = *type* +
  fixes *itself* :: *'a itself*

```
setup ⟪
let fun add-itself tyco thy =
  let
    val vs = Name.names Name.context 'a
      (replicate (Sign.arity-number thy tyco) @{sort type});
    val ty = Type (tyco, map TFree vs);
    val lhs = Const (@{const-name itself}, Term.itselfT ty);
    val rhs = Logic.mk-type ty;
    val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
  in
    thy
    |> TheoryTarget.instantiation ([tyco], vs, @{sort itself})
    |> '(fn lthy => Syntax.check-term lthy eq)
    |-> (fn eq => Specification.definition (NONE, ((, []), eq)))
    |> snd
    |> Class.prove-instantiation-instance (K (Class.intro-classes-tac []))
    |> LocalTheory.exit
    |> ProofContext.theory-of
  end
in TypedefPackage.interpretation add-itself end
⟫

instantiation bool :: itself
begin

definition itself = TYPE(bool)

instance ..

end

instantiation fun :: (type, type) itself
begin

definition itself = TYPE('a ⇒ 'b)

instance ..

end

hide (open) const itself

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* $(\exists!y.\ x = Inl\ y)$
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*!]: $a : A \Longrightarrow Inl(a) : A <+> B$
by (*simp add*: *Plus-def*)

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

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

Exhaustion rule for sums, a degenerate form of induction

lemma *sumE*:
$[|\ !!x::'a.\ s = Inl(x) \Longrightarrow P;\ \ !!y::'b.\ s = Inr(y) \Longrightarrow P$

```
    |] ==> 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
```

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/old-primrec-package.ML*)
(*Tools/primrec-package.ML*)
(*Tools/datatype-codegen.ML*)
begin

9.1 Least and greatest fixed points

context *complete-lattice*
begin

definition
lfp :: ('*a* ⇒ '*a*) ⇒ '*a* **where**
lfp f = *Inf* {*u*. *f u* ≤ *u*} — least fixed point

definition
gfp :: ('*a* ⇒ '*a*) ⇒ '*a* **where**
gfp f = *Sup* {*u*. *u* ≤ *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* ≤ *u*}

lemma *lfp-lowerbound*: *f A* ≤ *A* ==> *lfp f* ≤ *A*
by (*auto simp add*: *lfp-def intro*: *Inf-lower*)

lemma *lfp-greatest*: (!!*u*. *f u* ≤ *u* ==> *A* ≤ *u*) ==> *A* ≤ *lfp f*
by (*auto simp add*: *lfp-def intro*: *Inf-greatest*)

end

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

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

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

lemma *lfp-const*: *lfp* (λ*x*. *t*) = *t*
by (*rule lfp-unfold*) (*simp add*:*mono-def*)

9.3 General induction rules for least fixed points

theorem *lfp-induct*:

```
  assumes mono: mono f and ind: f (inf (lfp f) P) <= P
  shows lfp f <= P
proof −
  have inf (lfp f) P <= lfp f by (rule inf-le1)
  with mono have f (inf (lfp f) P) <= f (lfp f) ..
  also from mono have f (lfp f) = lfp f by (rule lfp-unfold [symmetric])
  finally have f (inf (lfp f) P) <= lfp f .
  from this and ind have f (inf (lfp f) P) <= inf (lfp f) P by (rule le-infI)
  hence lfp f <= inf (lfp f) P by (rule lfp-lowerbound)
  also have inf (lfp f) P <= P by (rule inf-le2)
  finally show ?thesis .
qed

lemma lfp-induct-set:
  assumes lfp: a: lfp(f)
      and mono: mono(f)
      and indhyp: !!x. [| x: f(lfp(f) Int {x. P(x)}) |] ==> P(x)
  shows P(a)
  by (rule lfp-induct [THEN subsetD, THEN CollectD, OF mono - lfp])
    (auto simp: inf-set-eq intro: indhyp)

lemma lfp-ordinal-induct:
  fixes f :: 'a::complete-lattice ⇒ 'a
  assumes mono: mono f
  and P-f: ⋀S. P S ⟹ P (f S)
  and P-Union: ⋀M. ∀S∈M. P S ⟹ P (Sup M)
  shows P (lfp f)
proof −
  let ?M = {S. S ≤ lfp f ∧ P S}
  have P (Sup ?M) using P-Union by simp
  also have Sup ?M = lfp f
  proof (rule antisym)
    show Sup ?M ≤ lfp f by (blast intro: Sup-least)
    hence f (Sup ?M) ≤ f (lfp f) by (rule mono [THEN monoD])
    hence f (Sup ?M) ≤ lfp f using mono [THEN lfp-unfold] by simp
    hence f (Sup ?M) ∈ ?M using P-f P-Union by simp
    hence f (Sup ?M) ≤ Sup ?M by (rule Sup-upper)
    thus lfp f ≤ Sup ?M by (rule lfp-lowerbound)
  qed
  finally show ?thesis .
qed

lemma lfp-ordinal-induct-set:
  assumes mono: mono f
  and P-f: !!S. P S ==> P(f S)
  and P-Union: !!M. !S:M. P S ==> P(Union M)
  shows P(lfp f)
  using assms unfolding Sup-set-eq [symmetric]
  by (rule lfp-ordinal-induct [where P=P])
```

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

lemma *def-lfp-unfold*: [| h==*lfp*(f); *mono*(f) |] ==> $h = f(h)$
by (*auto intro*!: *lfp-unfold*)

lemma *def-lfp-induct*:
[| A == *lfp*(f); *mono*(f);
f (*inf* A P) $\leq P$
|] ==> $A \leq P$
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 $Z \leq g$ Z) ==> *lfp* $f \leq$ *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*: $X \leq f\ X$ ==> $X \leq$ *gfp* f
by (*auto simp add*: *gfp-def intro*: *Sup-upper*)

lemma *gfp-least*: (!!u. $u \leq f\ u$ ==> $u \leq X$) ==> *gfp* $f \leq X$
by (*auto simp add*: *gfp-def intro*: *Sup-least*)

lemma *gfp-lemma2*: *mono* f ==> *gfp* $f \leq f$ (*gfp* f)
by (*iprover intro*: *gfp-least order-trans monoD gfp-upperbound*)

lemma *gfp-lemma3*: *mono* f ==> f (*gfp* f) $\leq$ *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* [**where** *f*=*f*], *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*} *RS* @{*thm eq-reflection*}
val inf-fun-eq = @{*thm inf-fun-eq*} *RS* @{*thm eq-reflection*}
val sup-bool-eq = @{*thm sup-bool-eq*} *RS* @{*thm eq-reflection*}
val sup-fun-eq = @{*thm sup-fun-eq*} *RS* @{*thm eq-reflection*}
val le-boolI = @{*thm le-boolI*}
val le-boolI' = @{*thm le-boolI'*}
val le-funI = @{*thm le-funI*}
val le-boolE = @{*thm le-boolE*}
val le-funE = @{*thm le-funE*}
val le-boolD = @{*thm le-boolD*}
val le-funD = @{*thm le-funD*}
val le-bool-def = @{*thm le-bool-def*} *RS* @{*thm eq-reflection*}
val le-fun-def = @{*thm le-fun-def*} *RS* @{*thm eq-reflection*}
⟫

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

theorems [*mono*] =
  imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj

imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
not-all not-ex
Ball-def Bex-def
induct-rulify-fallback

9.8 Inductive datatypes and primitive recursion

Package setup.

use *Tools/datatype-aux.ML*
use *Tools/datatype-prop.ML*
use *Tools/datatype-rep-proofs.ML*
use *Tools/datatype-abs-proofs.ML*
use *Tools/datatype-case.ML*
use *Tools/datatype-package.ML*
setup *DatatypePackage.setup*
use *Tools/old-primrec-package.ML*
use *Tools/primrec-package.ML*

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

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

Lambda-abstractions with pattern matching:

syntax
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((%-) *10*)
syntax (*xsymbols*)
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((λ-) *10*)

parse-translation (**advanced**) ⟪
let
fun fun-tr ctxt [*cs*] =
let
val x = *Free* (*Name.variant* (*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

lemma [*code func*]:
shows $False = P \longleftrightarrow \neg P$
and $True = P \longleftrightarrow P$
and $P = False \longleftrightarrow \neg P$
and $P = True \longleftrightarrow P$ **by** *simp-all*

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

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

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 ⟪
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*

code generator setup

instance *unit* :: *eq* **..**

lemma [*code func*]:
(*u*::*unit*) = *v* ⟷ *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

10.3 Pairs

10.3.1 Product type, basic operations and concrete syntax

definition
Pair-Rep :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where
Pair-Rep a b = $(\lambda x\ y.\ x = a \wedge y = b)$

global

typedef (*Prod*)
$('a, 'b) *$ (**infixr** $*$ *20*)
$= \{f.\ \exists a\ b.\ f =$ *Pair-Rep* $(a::'a)\ (b::'b)\}$
proof
fix *a b* **show** *Pair-Rep a b* $\in$ *?Prod*
by *rule+*
qed

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

consts
Pair :: $'a \Rightarrow 'b \Rightarrow 'a \times 'b$
fst :: $'a \times 'b \Rightarrow 'a$
snd :: $'a \times 'b \Rightarrow 'b$
split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
curry :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$

local

defs
Pair-def: *Pair a b* == *Abs-Prod* (*Pair-Rep a b*)
fst-def: *fst p* == *THE a. EX b. p* = *Pair a b*
snd-def: *snd p* == *THE b. EX a. p* = *Pair a b*
split-def: *split* == (%*c p. c* (*fst p*) (*snd p*))
curry-def: *curry* == (%*c x y. c* (*Pair x y*))

Patterns – extends pre-defined type *pttrn* used in abstractions.

nonterminals
tuple-args patterns

syntax
-tuple :: $'a$ => *tuple-args* => $'a * 'b$ ((1 $'$(-,/ -$'$)))
-tuple-arg :: $'a$ => *tuple-args* (-)

```
  -tuple-args :: 'a => tuple-args => tuple-args      (-,/ -)
  -pattern    :: [pttrn, patterns] => pttrn          ('(-,/ -'))
              :: pttrn => patterns                   (-)
  -patterns   :: [pttrn, patterns] => patterns       (-,/ -)
```

translations

```
  (x, y)       == Pair x y
  -tuple x (-tuple-args y z) == -tuple x (-tuple-arg (-tuple y z))
  %(x,y,zs).b  == split(%x (y,zs).b)
  %(x,y).b     == split(%x y. b)
  -abs (Pair x y) t => %(x,y).t
```

print-translation ⟪

```
let fun split-tr' [Abs (x,T,t as (Abs abs))] =
      (* split (%x y. t) => %(x,y) t *)
      let val (y,t') = atomic-abs-tr' abs;
          val (x',t'') = atomic-abs-tr' (x,T,t');

      in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end
    | split-tr' [Abs (x,T,(s as Const (split,-)$t))] =
      (* split (%x. (split (%y z. t))) => %(x,y,z). t *)
      let val (Const (-abs,-)$(Const (-pattern,-)$y$z)$t') = split-tr' [t];
          val (x',t'') = atomic-abs-tr' (x,T,t');
      in Syntax.const -abs$
          (Syntax.const -pattern$x'$(Syntax.const -patterns$y$z))$t'' end
    | split-tr' [Const (split,-)$t] =
      (* split (split (%x y z. t)) => %((x,y),z). t *)
      split-tr' [(split-tr' [t])] (* inner split-tr' creates next pattern *)
    | split-tr' [Const (-abs,-)$x-y$(Abs abs)] =
      (* split (%pttrn z. t) => %(pttrn,z). t *)
      let val (z,t) = atomic-abs-tr' abs;
      in Syntax.const -abs $ (Syntax.const -pattern $x-y$z) $ t end
    | split-tr' - =  raise Match;
in [(split, split-tr')]
end
⟫
```

typed-print-translation ⟪

```
let
  fun split-guess-names-tr' - T [Abs (x,-,Abs -)] = raise Match
    | split-guess-names-tr' - T [Abs (x,xT,t)] =
        (case (head-of t) of
           Const (split,-) => raise Match
         | - => let
                  val (-::yT::-) = binder-types (domain-type T) handle Bind => raise
```

```
Match;
                val (y,t') = atomic-abs-tr' (y,yT,(incr-boundvars 1 t)$Bound 0);
                val (x',t'') = atomic-abs-tr' (x,xT,t');
              in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end)
  | split-guess-names-tr' - T [t] =
    (case (head-of t) of
        Const (split,-) => raise Match
      | - => let
                val (xT::yT::-) = binder-types (domain-type T) handle Bind =>
raise Match;
                val (y,t') =
                    atomic-abs-tr' (y,yT,(incr-boundvars 2 t)$Bound 1$Bound 0);
                val (x',t'') = atomic-abs-tr' (x,xT,t');
              in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end)
  | split-guess-names-tr' - - - = raise Match;
in [(split, split-guess-names-tr')]
end
⟫
```

Towards a datatype declaration

lemma *surj-pair* [*simp*]: *EX x y. p = (x, y)*
  apply (*unfold Pair-def*)
  apply (*rule Rep-Prod* [*unfolded Prod-def*, *THEN CollectE*])
  apply (*erule exE*, *erule exE*, *rule exI*, *rule exI*)
  apply (*rule Rep-Prod-inverse* [*symmetric*, *THEN trans*])
  apply (*erule arg-cong*)
  done

lemma *PairE* [*cases type*: *∗*]:
  obtains *x y* **where** *p = (x, y)*
  using *surj-pair* [*of p*] **by** *blast*

lemma *prod-induct* [*induct type*: *∗*]: *(⋀a b. P (a, b)) ⟹ P x*
  by (*cases x*) *simp*

lemma *ProdI*: *Pair-Rep a b ∈ Prod*
  unfolding *Prod-def* **by** *rule+*

lemma *Pair-Rep-inject*: *Pair-Rep a b = Pair-Rep a' b' ⟹ a = a' ∧ b = b'*
  unfolding *Pair-Rep-def* **by** (*drule fun-cong*, *drule fun-cong*) *blast*

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

lemma *Pair-inject*:
  assumes *(a, b) = (a', b')*

and $a = a' \Longrightarrow b = b' \Longrightarrow R$
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*

rep-datatype *prod*
inject *Pair-eq*
induction *prod-induct*

10.3.2 Basic rules and proof tools

lemma *fst-eqD*: *fst* $(x, y) = a \Longrightarrow x = a$
by *simp*

lemma *snd-eqD*: *snd* $(x, y) = a \Longrightarrow y = a$
by *simp*

lemma *pair-collapse* [*simp*]: (*fst p*, *snd p*) = *p*
by (*cases p*) *simp*

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

lemma *split-paired-all*: $(!!x.\ PROP\ P\ x) == (!!a\ b.\ PROP\ P\ (a, b))$
proof
fix *a b*
assume $!!x.\ PROP\ P\ x$
then show *PROP P* (a, b) **.**
next
fix *x*
assume $!!a\ b.\ PROP\ P\ (a, b)$
from ‹*PROP P* (*fst x*, *snd x*)› **show** *PROP P x* **by** *simp*
qed

The rule *split-paired-all* does not work with the Simplifier because it also affects premises in congrence rules, where this can lead to premises of the form $!!a\ b.\ ... = ?P(a, b)$ which cannot be solved by reflexivity.

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

ML ⟪
```
(* replace parameters of product type by individual component parameters *)
val safe-full-simp-tac = generic-simp-tac true (true, false, false);
local (* filtering with exists-paired-all is an essential optimization *)
  fun exists-paired-all (Const (all, -) $ Abs (-, T, t)) =
        can HOLogic.dest-prodT T orelse exists-paired-all t
    | exists-paired-all (t $ u) = exists-paired-all t orelse exists-paired-all u
    | exists-paired-all (Abs (-, -, t)) = exists-paired-all t
    | exists-paired-all - = false;
  val ss = HOL-basic-ss
   addsimps [@{thm split-paired-all}, @{thm unit-all-eq2}, @{thm unit-abs-eta-conv}]
    addsimprocs [unit-eq-proc];
in
  val split-all-tac = SUBGOAL (fn (t, i) =>
    if exists-paired-all t then safe-full-simp-tac ss i else no-tac);
  val unsafe-split-all-tac = SUBGOAL (fn (t, i) =>
    if exists-paired-all t then full-simp-tac ss i else no-tac);
  fun split-all th =
   if exists-paired-all (Thm.prop-of th) then full-simplify ss th else th;
end;
```
⟫

declaration ⟪ *fn - =>*
```
Classical.map-cs (fn cs => cs addSbefore (split-all-tac, split-all-tac))
```
⟫

lemma *split-paired-All* [*simp*]: $(ALL\ x.\ P\ x) = (ALL\ a\ b.\ P\ (a, b))$
— [*iff*] is not a good idea because it makes *blast* loop
by *fast*

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

lemma *Pair-fst-snd-eq*: $s = t \longleftrightarrow fst\ s = fst\ t \wedge snd\ s = snd\ t$
by (*cases s*, *cases t*) *simp*

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

10.3.3 *split* and *curry*

lemma *split-conv* [*simp*, *code func*]: $split\ f\ (a, b) = f\ a\ b$
by (*simp add*: *split-def*)

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

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

lemma *splitI*: $f\ a\ b \Longrightarrow split\ f\ (a,\ b)$
by (*rule split-conv* [*THEN iffD2*])

lemma *splitD*: $split\ f\ (a,\ b) \Longrightarrow f\ a\ b$
by (*rule split-conv* [*THEN iffD1*])

lemma *curryI* [*intro*!]: $f\ (a,\ b) \Longrightarrow curry\ f\ a\ b$
by (*simp add*: *curry-def*)

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

lemma *curryE*: $curry\ f\ a\ b \Longrightarrow (f\ (a,\ b) \Longrightarrow Q) \Longrightarrow Q$
by (*simp add*: *curry-def*)

lemma *curry-split* [*simp*]: $curry\ (split\ f) = f$
by (*simp add*: *curry-def split-def*)

lemma *split-curry* [*simp*]: $split\ (curry\ f) = f$
by (*simp add*: *curry-def split-def*)

lemma *split-Pair* [*simp*]: $(\lambda(x,\ y).\ (x,\ y)) = id$
by (*simp add*: *split-def id-def*)

lemma *split-eta*: $(\lambda(x,\ y).\ f\ (x,\ y)) = f$
— Subsumes the old *split-Pair* when f is the identity function.
by (*rule ext*) *auto*

lemma *split-comp*: $split\ (f \circ g)\ x = f\ (g\ (fst\ x))\ (snd\ x)$
by (*cases x*) *simp*

lemma *split-twice*: $split\ f\ (split\ g\ p) = split\ (\lambda x\ y.\ split\ f\ (g\ x\ y))\ p$
unfolding *split-def* **..**

lemma *split-paired-The*: $(THE\ x.\ P\ x) = (THE\ (a,\ b).\ P\ (a,\ b))$
— Can't be added to simpset: loops!
by (*simp add*: *split-eta*)

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

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

lemma *cond-split-eta*: $(!!x\ y.\ f\ x\ y = g\ (x,\ y)) ==> (\%(x,\ y).\ f\ x\ y) = g$
by (*simp add*: *split-eta*)

Simplification procedure for *cond-split-eta*. Using *split-eta* as a rewrite rule

is not general enough, and using *cond-split-eta* directly would render some existing proofs very inefficient; similarly for *split-beta*.

ML ⟪

```
local
  val cond-split-eta-ss = HOL-basic-ss addsimps [thm cond-split-eta]
  fun  Pair-pat k 0 (Bound m) = (m = k)
  |    Pair-pat k i (Const (Pair,  -) $ Bound m $ t) = i > 0 andalso
                      m = k+i andalso Pair-pat k (i-1) t
  |    Pair-pat - - - = false;
  fun no-args k i (Abs (-, -, t)) = no-args (k+1) i t
  |   no-args k i (t $ u) = no-args k i t andalso no-args k i u
  |   no-args k i (Bound m) = m < k orelse m > k+i
  |   no-args - - - = true;
  fun split-pat tp i (Abs  (-,-,t)) = if tp 0 i t then SOME (i,t) else NONE
  |   split-pat tp i (Const (split, -) $ Abs (-, -, t)) = split-pat tp (i+1) t
  |   split-pat tp i - = NONE;
  fun metaeq ss lhs rhs = mk-meta-eq (Goal.prove (Simplifier.the-context ss) [] []
        (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs,rhs)))
        (K (simp-tac (Simplifier.inherit-context ss cond-split-eta-ss) 1)));

  fun beta-term-pat k i (Abs (-, -, t)) = beta-term-pat (k+1) i t
  |   beta-term-pat k i (t $ u) = Pair-pat k i (t $ u) orelse
                      (beta-term-pat k i t andalso beta-term-pat k i u)
  |   beta-term-pat k i t = no-args k i t;
  fun  eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
  |    eta-term-pat - - - = false;
  fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
  |   subst arg k i (t $ u) = if Pair-pat k i (t $ u) then incr-boundvars k arg
                          else (subst arg k i t $ subst arg k i u)
  |   subst arg k i t = t;
  fun beta-proc ss (s as Const (split, -) $ Abs (-, -, t) $ arg) =
        (case split-pat beta-term-pat 1 t of
        SOME (i,f) => SOME (metaeq ss s (subst arg 0 i f))
        | NONE => NONE)
  |   beta-proc - - = NONE;
  fun eta-proc ss (s as Const (split, -) $ Abs (-, -, t)) =
        (case split-pat eta-term-pat 1 t of
          SOME (-,ft) => SOME (metaeq ss s (let val (f $ arg) = ft in f end))
        | NONE => NONE)
  |   eta-proc - - = NONE;
in
  val split-beta-proc = Simplifier.simproc @{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* [*mono*]: (%(*x*, *y*). *P x y*) *z* = *P* (*fst z*) (*snd z*)
by (*subst surjective-pairing*, *rule split-conv*)

lemma *split-split* [*noatp*]: *R*(*split c p*) = (*ALL x y*. *p* = (*x*, *y*) −−> *R*(*c x y*))
— For use with *split* and the Simplifier.
by (*insert surj-pair* [*of p*], *clarify*, *simp*)

split-split could be declared as [*split*] done after the Splitter has been speeded up significantly; precompute the constants involved and don't do anything unless the current goal contains one of those constants.

lemma *split-split-asm* [*noatp*]: *R* (*split c p*) = (~(*EX x y*. *p* = (*x*, *y*) & (~*R* (*c x y*))))
by (*subst split-split*, *simp*)

split used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *split* as rewrite.

lemma *splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *split c p*
apply (*simp only*: *split-tupled-all*)
apply (*simp* (*no-asm-simp*))
done

lemma *splitI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *split c p x*
apply (*simp only*: *split-tupled-all*)
apply (*simp* (*no-asm-simp*))
done

lemma *splitE*: *split c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
by (*induct p*) (*auto simp add*: *split-def*)

lemma *splitE'*: *split c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
by (*induct p*) (*auto simp add*: *split-def*)

lemma *splitE2*:
[| *Q* (*split P z*); !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R* |] ==> *R*
proof −
assume *q*: *Q* (*split P z*)
assume *r*: !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R*
show *R*
apply (*rule r surjective-pairing*)+
apply (*rule split-beta* [*THEN subst*], *rule q*)
done
qed

lemma *splitD'*: *split R* (*a*,*b*) *c* ==> *R a b c*
by *simp*

lemma *mem-splitI*: *z*: *c a b* ==> *z*: *split c* (*a*, *b*)
by *simp*

lemma *mem-splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *z*: *c a b* |] ==> *z*: *split c p*
by (*simp only*: *split-tupled-all*, *simp*)

lemma *mem-splitE*:
assumes *major*: *z*: *split c p*
and *cases*: !!*x y*. [| *p* = (*x*,*y*); *z*: *c x y* |] ==> *Q*
shows *Q*
by (*rule major* [*unfolded split-def*] *cases surjective-pairing*)+

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

ML ⟪
local (∗ *filtering with exists-p-split is an essential optimization* ∗)
fun exists-p-split (*Const* (*split*,-) $ - $ (*Const* (*Pair*,-)$-$-)) = *true*
| *exists-p-split* (*t* $ *u*) = *exists-p-split t orelse exists-p-split u*
| *exists-p-split* (*Abs* (-, -, *t*)) = *exists-p-split t*
| *exists-p-split* - = *false*;
val ss = *HOL-basic-ss addsimps* [*thm split-conv*];
in
val split-conv-tac = *SUBGOAL* (*fn* (*t*, *i*) =>
if exists-p-split t then safe-full-simp-tac ss i else no-tac);
end;
⟫

declaration ⟪ *fn* - =>
Classical.map-cs (*fn cs* => *cs addSbefore* (*split-conv-tac*, *split-conv-tac*))
⟫

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

lemma *split-eta-SetCompr2* [*simp*,*noatp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P x y*) = *split P*
by (*rule ext*) *fast*

lemma *split-part* [*simp*]: (%(*a*,*b*). *P* & *Q a b*) = (%*ab*. *P* & *split Q ab*)
— Allows simplifications of nested splits in case of independent predicates.
by (*rule ext*) *blast*

lemma *split-comp-eq*:
fixes *f* :: *'a* => *'b* => *'c* **and** *g* :: *'d* => *'a*

shows (%*u*. *f* (*g* (*fst u*)) (*snd u*)) = (*split* (%*x*. *f* (*g x*)))
by (*rule ext*) *auto*

lemma *pair-imageI* [*intro*]: (*a*, *b*) : *A* ==> *f a b* : (%(*a*, *b*). *f a b*) ' *A*
apply (*rule-tac x* = (*a*, *b*) **in** *image-eqI*)
apply *auto*
done

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

Setup of internal *split-rule*.

definition
internal-split :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a* × '*b* ⇒ '*c*
where
internal-split == *split*

lemma *internal-split-conv*: *internal-split c* (*a*, *b*) = *c a b*
by (*simp only*: *internal-split-def split-conv*)

hide *const internal-split*

use *Tools/split-rule.ML*
setup *SplitRule.setup*

lemmas *prod-caseI* = *prod.cases* [*THEN iffD2*, *standard*]

lemma *prod-caseI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *prod-case c p*
by *auto*

lemma *prod-caseI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *prod-case c p x*
by (*auto simp*: *split-tupled-all*)

lemma *prod-caseE*: *prod-case c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
by (*induct p*) *auto*

lemma *prod-caseE'*: *prod-case c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
by (*induct p*) *auto*

lemma *prod-case-unfold*: *prod-case* = (%*c p*. *c* (*fst p*) (*snd p*))
by (*simp add*: *expand-fun-eq*)

declare *prod-caseI2'* [*intro*!] *prod-caseI2* [*intro*!] *prod-caseI* [*intro*!]
declare *prod-caseE'* [*elim*!] *prod-caseE* [*elim*!]

lemma *prod-case-split*:
prod-case = *split*

by (*auto simp add*: *expand-fun-eq*)

lemma *prod-case-beta*:
prod-case f p = *f* (*fst p*) (*snd p*)
unfolding *prod-case-split split-beta* **..**

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*

10.4.1 Derived operations

The composition-uncurry combinator.

notation *fcomp* (**infixl** *o>* *60*)

definition
  scomp :: (*'a* ⇒ *'b* × *'c*) ⇒ (*'b* ⇒ *'c* ⇒ *'d*) ⇒ *'a* ⇒ *'d* (**infixl** *o→* *60*)
where
  f o→ g = (λ*x*. *split g* (*f x*))

lemma *scomp-apply*: (*f o→ g*) *x* = *split g* (*f x*)
  by (*simp add*: *scomp-def*)

lemma *Pair-scomp*: *Pair x o→ f* = *f x*
  by (*simp add*: *expand-fun-eq scomp-apply*)

lemma *scomp-Pair*: *x o→ Pair* = *x*
  by (*simp add*: *expand-fun-eq scomp-apply*)

lemma *scomp-scomp*: (*f o→ g*) *o→ h* = *f o→* (λ*x*. *g x o→ h*)
  by (*simp add*: *expand-fun-eq split-twice scomp-def*)

lemma *scomp-fcomp*: (*f o→ g*) *o> h* = *f o→* (λ*x*. *g x o> h*)
  by (*simp add*: *expand-fun-eq scomp-apply fcomp-def split-def*)

lemma *fcomp-scomp*: (*f o> g*) *o→ h* = *f o>* (*g o→ h*)
  by (*simp add*: *expand-fun-eq scomp-apply fcomp-apply*)

no-notation *fcomp* (**infixl** *o>* *60*)
no-notation *scomp* (**infixl** *o→* *60*)

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

definition *prod-fun* :: (*'a* ⇒ *'c*) ⇒ (*'b* ⇒ *'d*) ⇒ *'a* × *'b* ⇒ *'c* × *'d* **where**
  [*code func del*]: *prod-fun f g* = (λ(*x*, *y*). (*f x*, *g y*))

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*)
  by (*rule ext*) *auto*

lemma *prod-fun-ident* [*simp*]: *prod-fun* (%*x*. *x*) (%*y*. *y*) = (%*z*. *z*)
  by (*rule ext*) *auto*

lemma *prod-fun-imageI* [*intro*]: (*a*, *b*) : *r* ==> (*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
apfst :: (′*a* ⇒ ′*c*) ⇒ ′*a* × ′*b* ⇒ ′*c* × ′*b*
where
[*code func del*]: *apfst f* = *prod-fun f id*

definition
apsnd :: (′*b* ⇒ ′*c*) ⇒ ′*a* × ′*b* ⇒ ′*a* × ′*c*
where
[*code func del*]: *apsnd f* = *prod-fun id f*

lemma *apfst-conv* [*simp*, *code*]:
apfst f (*x*, *y*) = (*f x*, *y*)
by (*simp add*: *apfst-def*)

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

Disjoint union of a family of sets – Sigma.

definition *Sigma* :: [′*a set*, ′*a* => ′*b set*] => (′*a* × ′*b*) *set* **where**
Sigma-def: *Sigma A B* == *UN x*:*A*. *UN y*:*B x*. {*Pair x y*}

abbreviation
Times :: [′*a set*, ′*b set*] => (′*a* ∗ ′*b*) *set*
(**infixr** <∗> *80*) **where**
A <∗> *B* == *Sigma A* (%-. *B*)

notation (*xsymbols*)
Times (**infixr** × *80*)

notation (*HTML* **output**)
Times (**infixr** × *80*)

syntax
@*Sigma* ::[*pttrn*, ′*a set*, ′*b set*] => (′*a* ∗ ′*b*) *set* ((*3SIGMA* -:-./ -) [*0*, *0*, *10*] *10*)

translations

SIGMA x:A. B == Product-Type.Sigma A (%x. B)

lemma *SigmaI* [*intro*!]: [| *a*:*A*; *b*:*B*(*a*) |] ==> (*a*,*b*) : *Sigma A B*
by (*unfold Sigma-def*) *blast*

lemma *SigmaE* [*elim*!]:
[| *c*: *Sigma A B*;
!!*x y*.[| *x*:*A*; *y*:*B*(*x*); *c*=(*x*,*y*) |] ==> *P*
|] ==> *P*
— The general elimination rule.
by (*unfold Sigma-def*) *blast*

Elimination of $(a, b) \in A \times B$ – introduces no eigenvariables.

lemma *SigmaD1*: (*a*, *b*) : *Sigma A B* ==> *a* : *A*
by *blast*

lemma *SigmaD2*: (*a*, *b*) : *Sigma A B* ==> *b* : *B a*
by *blast*

lemma *SigmaE2*:
[| (*a*, *b*) : *Sigma A B*;
[| *a*:*A*; *b*:*B*(*a*) |] ==> *P*
|] ==> *P*
by *blast*

lemma *Sigma-cong*:
⟦*A* = *B*; !!*x*. *x* ∈ *B* ⟹ *C x* = *D x*⟧
⟹ (*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*]: $((a,b)\text{: } Sigma\ A\ B) = (a\text{:}A\ \&\ b\text{:}B(a))$
by *blast*

lemma *Times-subset-cancel2*: $x\text{:}C \Longrightarrow (A <*> C <= B <*> C) = (A <= B)$
by *blast*

lemma *Times-eq-cancel2*: $x\text{:}C \Longrightarrow (A <*> C = B <*> C) = (A = B)$
by (*blast elim*: *equalityE*)

lemma *SetCompr-Sigma-eq*:
$Collect\ (split\ (\%x\ y.\ P\ x\ \&\ Q\ x\ y)) = (SIGMA\ x\text{:}Collect\ P.\ Collect\ (Q\ x))$
by *blast*

lemma *Collect-split* [*simp*]: $\{(a,b).\ P\ a\ \&\ Q\ b\} = Collect\ P <*> Collect\ Q$
by *blast*

lemma *UN-Times-distrib*:
$(UN\ (a,b)\text{:}(A <*> B).\ E\ a <*> F\ b) = (UNION\ A\ E) <*> (UNION\ B\ F)$
— Suggested by Pierre Chartier
by *blast*

lemma *split-paired-Ball-Sigma* [*simp*,*noatp*]:
$(ALL\ z\text{: }Sigma\ A\ B.\ P\ z) = (ALL\ x\text{:}A.\ ALL\ y\text{: }B\ x.\ P(x,y))$
by *blast*

lemma *split-paired-Bex-Sigma* [*simp*,*noatp*]:
$(EX\ z\text{: }Sigma\ A\ B.\ P\ z) = (EX\ x\text{:}A.\ EX\ y\text{: }B\ x.\ P(x,y))$
by *blast*

lemma *Sigma-Un-distrib1*: $(SIGMA\ i\text{:}I\ Un\ J.\ C(i)) = (SIGMA\ i\text{:}I.\ C(i))\ Un\ (SIGMA\ j\text{:}J.\ C(j))$
by *blast*

lemma *Sigma-Un-distrib2*: $(SIGMA\ i\text{:}I.\ A(i)\ Un\ B(i)) = (SIGMA\ i\text{:}I.\ A(i))\ Un\ (SIGMA\ i\text{:}I.\ B(i))$
by *blast*

lemma *Sigma-Int-distrib1*: $(SIGMA\ i\text{:}I\ Int\ J.\ C(i)) = (SIGMA\ i\text{:}I.\ C(i))\ Int\ (SIGMA\ j\text{:}J.\ C(j))$
by *blast*

lemma *Sigma-Int-distrib2*: $(SIGMA\ i\text{:}I.\ A(i)\ Int\ B(i)) = (SIGMA\ i\text{:}I.\ A(i))\ Int\ (SIGMA\ i\text{:}I.\ B(i))$
by *blast*

lemma *Sigma-Diff-distrib1*: $(SIGMA\ i\text{:}I - J.\ C(i)) = (SIGMA\ i\text{:}I.\ C(i)) - (SIGMA\ j\text{:}J.\ C(j))$
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*

10.4.2 Code generator setup

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 $CASE \equiv split\ f$
shows $CASE\ (a, b) \equiv f\ a\ b$
using *assms* **by** *simp*

setup ⟪
Code.add-case @{*thm split-case-cert*}
⟫

code-type $*$
(*SML* **infix** *2* $*$)
(*OCaml* **infix** *2* $*$)
(*Haskell* !((-),/ (-)))

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

code-const *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 aF aT bF bT (x, y) = HOLogic.pair-const aT bT $ aF x $ bF y;
⟫
attach (*test*) ⟪
fun gen-id-42 aG aT bG bT i =
let
val (x, t) = aG i;
val (y, u) = bG i
in ((x, y), fn () => HOLogic.pair-const aT bT $ t () $ u ()) end;
⟫

consts-code
Pair ((-,/ -))

setup ⟪

let

fun strip-abs-split 0 t = ([], t)
| strip-abs-split i (Abs (s, T, t)) =
let
val s' = Codegen.new-name t s;
val v = Free (s', T)
in apfst (cons v) (strip-abs-split (i−1) (subst-bound (v, t))) end
| strip-abs-split i (u as Const (split, -) $ t) = (case strip-abs-split (i+1) t of
(v :: v' :: vs, u) => (HOLogic.mk-prod (v, v') :: vs, u)
| - => ([], u))
| strip-abs-split i t = ([], 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, [Codegen.str let , Pretty.blk (0, List.concat
                (separate [Codegen.str ;, Pretty.brk 1] (map (fn (pl, pr) =>
                  [Pretty.block [Codegen.str val , pl, Codegen.str  =,
                    Pretty.brk 1, pr]]) qs))),
               Pretty.brk 1, Codegen.str in , pu,
               Pretty.brk 1, Codegen.str end])) pargs)
         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 [Codegen.str (fn , q, Codegen.str  =>,
                 Pretty.brk 1, pu, Codegen.str )]) pargs)
           end
       | - => NONE)
  | - => NONE);

in

  Codegen.add-codegen let-codegen let-codegen
  #> Codegen.add-codegen split-codegen split-codegen

end
⟩⟩
```

10.5 Legacy bindings

ML ⟪
val Collect-split = thm Collect-split;
val Compl-Times-UNIV1 = thm Compl-Times-UNIV1;
val Compl-Times-UNIV2 = thm Compl-Times-UNIV2;
val PairE = thm PairE;
val Pair-Rep-inject = thm Pair-Rep-inject;
val Pair-def = thm Pair-def;
val Pair-eq = thm Pair-eq;
val Pair-fst-snd-eq = thm Pair-fst-snd-eq;
val ProdI = thm ProdI;
val SetCompr-Sigma-eq = thm SetCompr-Sigma-eq;
val SigmaD1 = thm SigmaD1;
val SigmaD2 = thm SigmaD2;
val SigmaE = thm SigmaE;
val SigmaE2 = thm SigmaE2;
val SigmaI = thm SigmaI;
val Sigma-Diff-distrib1 = thm Sigma-Diff-distrib1;
val Sigma-Diff-distrib2 = thm Sigma-Diff-distrib2;
val Sigma-Int-distrib1 = thm Sigma-Int-distrib1;
val Sigma-Int-distrib2 = thm Sigma-Int-distrib2;
val Sigma-Un-distrib1 = thm Sigma-Un-distrib1;
val Sigma-Un-distrib2 = thm Sigma-Un-distrib2;
val Sigma-Union = thm Sigma-Union;
val Sigma-def = thm Sigma-def;
val Sigma-empty1 = thm Sigma-empty1;
val Sigma-empty2 = thm Sigma-empty2;
val Sigma-mono = thm Sigma-mono;
val The-split = thm The-split;
val The-split-eq = thm The-split-eq;
val The-split-eq = thm The-split-eq;
val Times-Diff-distrib1 = thm Times-Diff-distrib1;
val Times-Int-distrib1 = thm Times-Int-distrib1;
val Times-Un-distrib1 = thm Times-Un-distrib1;
val Times-eq-cancel2 = thm Times-eq-cancel2;
val Times-subset-cancel2 = thm Times-subset-cancel2;
val UNIV-Times-UNIV = thm UNIV-Times-UNIV;
val UN-Times-distrib = thm UN-Times-distrib;
val Unity-def = thm Unity-def;
val cond-split-eta = thm cond-split-eta;
val fst-conv = thm fst-conv;
val fst-def = thm fst-def;
val fst-eqD = thm fst-eqD;
val inj-on-Abs-Prod = thm inj-on-Abs-Prod;
val mem-Sigma-iff = thm mem-Sigma-iff;
val mem-splitE = thm mem-splitE;
val mem-splitI = thm mem-splitI;
val mem-splitI2 = thm mem-splitI2;
val prod-eqI = thm prod-eqI;

val prod-fun = thm prod-fun;
val prod-fun-compose = thm prod-fun-compose;
val prod-fun-def = thm prod-fun-def;
val prod-fun-ident = thm prod-fun-ident;
val prod-fun-imageE = thm prod-fun-imageE;
val prod-fun-imageI = thm prod-fun-imageI;
val prod-induct = thm prod-induct;
val snd-conv = thm snd-conv;
val snd-def = thm snd-def;
val snd-eqD = thm snd-eqD;
val split = thm split;
val splitD = thm splitD;
val splitD' = thm splitD';
val splitE = thm splitE;
val splitE' = thm splitE';
val splitE2 = thm splitE2;
val splitI = thm splitI;
val splitI2 = thm splitI2;
val splitI2' = thm splitI2';
val split-beta = thm split-beta;
val split-conv = thm split-conv;
val split-def = thm split-def;
val split-eta = thm split-eta;
val split-eta-SetCompr = thm split-eta-SetCompr;
val split-eta-SetCompr2 = thm split-eta-SetCompr2;
val split-paired-All = thm split-paired-All;
val split-paired-Ball-Sigma = thm split-paired-Ball-Sigma;
val split-paired-Bex-Sigma = thm split-paired-Bex-Sigma;
val split-paired-Ex = thm split-paired-Ex;
val split-paired-The = thm split-paired-The;
val split-paired-all = thm split-paired-all;
val split-part = thm split-part;
val split-split = thm split-split;
val split-split-asm = thm split-split-asm;
val split-tupled-all = thms split-tupled-all;
val split-weak-cong = thm split-weak-cong;
val surj-pair = thm surj-pair;
val surjective-pairing = thm surjective-pairing;
val unit-abs-eta-conv = thm unit-abs-eta-conv;
val unit-all-eq1 = thm unit-all-eq1;
val unit-all-eq2 = thm unit-all-eq2;
val unit-eq = thm unit-eq;
⟩⟩

10.6 Further inductive packages

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

use *Tools/inductive-set-package.ML*
setup *InductiveSetPackage.setup*

use *Tools/datatype-realizer.ML*
setup *DatatypeRealizer.setup*

end

11 Record: Extensible records with structural subtyping

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

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
by *simp*

lemma *rec-UNIV-I*: $\bigwedge x.\ x{\in}UNIV \equiv True$
by *simp*

lemma *rec-True-simp*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
by *simp*

lemma *K-record-comp*: $(\lambda x.\ c) \circ f = (\lambda x.\ c)$
by (*simp add*: *comp-def*)

11.1 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
```

12 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.

12.1 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 *ab-semigroup-idem-mult* = *ab-semigroup-mult* +
assumes *mult-idem*: $x * x = x$
begin

lemma *mult-left-idem*: $x * (x * y) = x * y$
unfolding *mult-assoc* [*symmetric*, *of x*] *mult-idem* **..**

lemmas *mult-ac-idem* = *mult-ac mult-idem mult-left-idem*

end

lemmas *mult-ac-idem* = *mult-ac mult-idem mult-left-idem*

class *monoid-add* = *zero* + *semigroup-add* +
assumes *add-0-left* [*simp*]: $0 + a = a$
and *add-0-right* [*simp*]: $a + 0 = a$

lemma *zero-reorient*: $0 = x \longleftrightarrow x = 0$
by (*rule eq-commute*)

class *comm-monoid-add* = *zero* + *ab-semigroup-add* +
assumes *add-0*: $0 + a = a$
begin

subclass *monoid-add*
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*

lemma *one-reorient*: *1* = *x* ⟷ *x* = *1*
by (*rule eq-commute*)

class *comm-monoid-mult* = *one* + *ab-semigroup-mult* +
assumes *mult-1*: *1* ∗ *a* = *a*
begin

subclass *monoid-mult*
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* ⟹ *b* = *c*
assumes *add-right-imp-eq*: *b* + *a* = *c* + *a* ⟹ *b* = *c*

class *cancel-ab-semigroup-add* = *ab-semigroup-add* +
assumes *add-imp-eq*: *a* + *b* = *a* + *c* ⟹ *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* ⟷ *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

12.2 Groups

class *group-add* = *minus* + *uminus* + *monoid-add* +
assumes *left-minus* [*simp*]: $- a + a = 0$
assumes *diff-minus*: $a - b = a + (- b)$
begin

lemma *minus-add-cancel*: $- a + (a + b) = b$
by (*simp add*: *add-assoc*[*symmetric*])

lemma *minus-zero* [*simp*]: $- 0 = 0$
proof −
have $- 0 = - 0 + (0 + 0)$ **by** (*simp only*: *add-0-right*)
also have $\ldots = 0$ **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** ‹$a - b = 0$› **by** *simp*
finally show $a = b$ **.**
next
assume $a = b$ **thus** $a - b = 0$ **by** (*simp add*: *diff-minus*)
qed

lemma *equals-zero-I*:
assumes $a + b = 0$
shows $- a = b$
proof −
have $- a = - a + (a + b)$ **using** *assms* **by** *simp*
also have $\ldots = b$ **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* + *uminus* + *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

12.3 (Partially) Ordered Groups

class *pordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +
assumes *add-left-mono*: $a \leq b \Longrightarrow c + a \leq c + b$
begin

lemma *add-right-mono*:
$a \leq b \Longrightarrow a + c \leq b + c$
by (*simp add*: *add-commute* [*of* - *c*] *add-left-mono*)

non-strict, in both arguments

lemma *add-mono*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow a + c \leq b + d$
apply (*erule add-right-mono* [*THEN order-trans*])
apply (*simp add*: *add-commute add-left-mono*)
done

end

class *pordered-cancel-ab-semigroup-add* =
pordered-ab-semigroup-add + *cancel-ab-semigroup-add*
begin

lemma *add-strict-left-mono*:
$a < b \Longrightarrow c + a < c + b$
by (*auto simp add*: *less-le add-left-mono*)

lemma *add-strict-right-mono*:
  $a < b \Longrightarrow a + c < b + c$
  by (*simp add*: *add-commute* [*of* - *c*] *add-strict-left-mono*)

Strict monotonicity in both arguments

lemma *add-strict-mono*:
  $a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$
apply (*erule add-strict-right-mono* [*THEN less-trans*])
apply (*erule add-strict-left-mono*)
done

lemma *add-less-le-mono*:
  $a < b \Longrightarrow c \leq d \Longrightarrow a + c < b + d$
apply (*erule add-strict-right-mono* [*THEN less-le-trans*])
apply (*erule add-left-mono*)
done

lemma *add-le-less-mono*:
  $a \leq b \Longrightarrow c < d \Longrightarrow a + c < b + d$
apply (*erule add-right-mono* [*THEN le-less-trans*])
apply (*erule add-strict-left-mono*)
done

end

class *pordered-ab-semigroup-add-imp-le* =
  pordered-cancel-ab-semigroup-add +
  assumes *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
begin

lemma *add-less-imp-less-left*:
   assumes *less*: $c + a < c + b$
   shows $a < b$
proof −
  from *less* **have** *le*: $c + a <= c + b$ **by** (*simp add*: *order-le-less*)
  have $a <= b$
    apply (*insert le*)
    apply (*drule add-le-imp-le-left*)
    by (*insert le*, *drule add-le-imp-le-left*, *assumption*)
  moreover have $a \neq b$
  proof (*rule ccontr*)
    assume ~$(a \neq b)$
    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

12.4 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 *intro-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 *intro-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 $a \leq b$
thus $-b \leq -a$ **by** (*rule le-imp-neg-le*)
qed

lemma *neg-le-0-iff-le* [*simp*]: $- a \leq 0 \longleftrightarrow 0 \leq a$
by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-0-le-iff-le* [*simp*]: $0 \leq - a \longleftrightarrow a \leq 0$
by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-less-iff-less* [*simp*]: $- b < - a \longleftrightarrow a < b$
by (*force simp add*: *less-le*)

lemma *neg-less-0-iff-less* [*simp*]: $- a < 0 \longleftrightarrow 0 < a$
by (*subst neg-less-iff-less* [*symmetric*], *simp*)

lemma *neg-0-less-iff-less* [*simp*]: $0 < - a \longleftrightarrow a < 0$
by (*subst neg-less-iff-less* [*symmetric*], *simp*)

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < - b \longleftrightarrow b < - a$
proof −
have $(- (-a) < - b) = (b < - a)$ **by** (*rule neg-less-iff-less*)
thus *?thesis* **by** *simp*
qed

lemma *minus-less-iff*: $- a < b \longleftrightarrow - b < a$
proof −
have $(- a < - (-b)) = (- b < a)$ **by** (*rule neg-less-iff-less*)
thus *?thesis* **by** *simp*
qed

lemma *le-minus-iff*: $a \leq - b \longleftrightarrow b \leq - a$
proof −
have *mm*: !! *a* (*b*::'*a*). $(-(-a)) < -b \Longrightarrow -(-b) < -a$ **by** (*simp only*: *minus-less-iff*)
have $(- (- a) <= -b) = (b <= - a)$
apply (*auto simp only*: *le-less*)
apply (*drule mm*)
apply (*simp-all*)

apply (*drule mm*[*simplified*], *assumption*)
done
then show *?thesis* **by** *simp*
qed

lemma *minus-le-iff*: $- a \leq b \longleftrightarrow - b \leq a$
by (*auto simp add*: *le-less minus-less-iff*)

lemma *less-iff-diff-less-0*: $a < b \longleftrightarrow a - b < 0$
proof −
have $(a < b) = (a + (- b) < b + (-b))$
by (*simp only*: *add-less-cancel-right*)
also have ... = $(a - b < 0)$ **by** (*simp add*: *diff-minus*)
finally show *?thesis* **.**
qed

lemma *diff-less-eq*: $a - b < c \longleftrightarrow a < c + b$
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 *intro-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 *intro-locales*

lemma *neg-less-eq-nonneg*:
$- a \leq a \longleftrightarrow 0 \leq a$
proof
assume *A*: $- a \leq a$ **show** $0 \leq a$
proof (*rule classical*)
assume $\neg\ 0 \leq a$
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*: $0 \leq a$ **show** $- a \leq a$
proof (*rule order-trans*)
show $- a \leq 0$ **using** *A* **by** (*simp add*: *minus-le-iff*)
next
show $0 \leq a$ **using** *A* **.**
qed
qed

lemma *less-eq-neg-nonpos*:
$a \leq - a \longleftrightarrow a \leq 0$
proof
assume *A*: $a \leq - a$ **show** $a \leq 0$
proof (*rule classical*)
assume $\neg\ a \leq 0$
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*: $a \leq 0$ **show** $a \leq - a$
proof (*rule order-trans*)
show $0 \leq - a$ **using** *A* **by** (*simp add*: *minus-le-iff*)
next
show $a \leq 0$ **using** *A* **.**
qed
qed

lemma *equal-neg-zero*:
$a = - a \longleftrightarrow a = 0$
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 ⟷ 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≤a; b≤c|] ==> b ≤ 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≤c; b≤a|] ==> b ≤ 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≤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≤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| ≥ 0
    and abs-ge-self: a ≤ |a|
    and abs-leI: a ≤ b ⟹ - a ≤ b ⟹ |a| ≤ b
    and abs-minus-cancel [simp]: |-a| = |a|
    and abs-triangle-ineq: |a + b| ≤ |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*]: ¬ |*a*| < *0*
proof −
have *a*: ⋀*x y*. *x* ≤ *y* ⟹ ¬ *y* < *x* **by** *auto*
show *?thesis* **by** (*simp add*: *a*)
qed

lemma *abs-ge-minus-self*: − *a* ≤ |*a*|
proof −
have − *a* ≤ |−*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* ⟹ |*a*| = *a*
by (*rule abs-of-nonneg*, *rule less-imp-le*)

lemma *abs-of-nonpos* [*simp*]:
assumes *a* ≤ *0*
shows |*a*| = − *a*
proof −
let *?b* = − *a*
have − *?b* ≤ *0* ⟹ |− *?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* ⟹ |*a*| = − *a*
by (*rule abs-of-nonpos*, *rule less-imp-le*)

lemma *abs-le-D1*: |*a*| ≤ *b* ⟹ *a* ≤ *b*
by (*insert abs-ge-self*, *blast intro*: *order-trans*)

lemma *abs-le-D2*: |*a*| ≤ *b* ⟹ − *a* ≤ *b*
by (*insert abs-le-D1* [*of uminus a*], *simp*)

lemma *abs-le-iff*: |*a*| ≤ *b* ⟷ *a* ≤ *b* ∧ − *a* ≤ *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

12.5 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** *intro-locales*
subclass *lordered-ab-group-add-join* **by** *intro-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*) ≤ *b*
    by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
      (*simp*, *simp add*: *add-sup-distrib-left*)
next
  fix *a b c* :: ′*a*
  assume *a* ≤ *b a* ≤ *c*
  then show *a* ≤ − *sup* (−*b*) (−*c*) **by** (*subst neg-le-iff-le* [*symmetric*])
    (*simp add*: *le-supI*)
qed

lemma *sup-eq-neg-inf*: *sup a b* = − *inf* (−*a*) (−*b*)
proof (*rule sup-unique*)
  fix *a b* :: ′*a*
  show *a* ≤ − *inf* (−*a*) (−*b*)
    by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
      (*simp*, *simp add*: *add-inf-distrib-left*)
next
  fix *a b* :: ′*a*
  show *b* ≤ − *inf* (−*a*) (−*b*)
    by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
      (*simp*, *simp add*: *add-inf-distrib-left*)
next
  fix *a b c* :: ′*a*
  assume *a* ≤ *c b* ≤ *c*
  then show − *inf* (−*a*) (−*b*) ≤ *c* **by** (*subst neg-le-iff-le* [*symmetric*])
    (*simp add*: *le-infI*)
qed

lemma *neg-inf-eq-sup*: − *inf a b* = *sup* (−*a*) (−*b*)
  by (*simp add*: *inf-eq-neg-sup*)

lemma *neg-sup-eq-inf*: − *sup a b* = *inf* (−*a*) (−*b*)
  by (*simp add*: *sup-eq-neg-inf*)

lemma *add-eq-inf-sup*: *a* + *b* = *sup a b* + *inf a b*
proof −
  have *0* = − *inf 0* (*a*−*b*) + *inf* (*a*−*b*) *0* **by** (*simp add*: *inf-commute*)

hence *0 = sup 0 (b−a) + inf (a−b) 0* **by** (*simp add*: *inf-eq-neg-sup*)
hence *0 = (−a + sup a b) + (inf a b + (−b))*
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

12.6 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*]:
$a + a \leq 0 \longleftrightarrow a \leq 0$
proof −
have $a + a \leq 0 \longleftrightarrow 0 \leq - (a + a)$ **by** (*subst le-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a \leq 0$ **by** (*simp add*: *zero-le-double-add-iff-zero-le-single-add*)
ultimately show *?thesis* **by** *blast*
qed

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]:
$a + a < 0 \longleftrightarrow a < 0$
proof −
have $a + a < 0 \longleftrightarrow 0 < - (a + a)$ **by** (*subst less-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a < 0$ **by** (*simp add*: *zero-less-double-add-iff-zero-less-single-add*)
ultimately show *?thesis* **by** *blast*
qed

declare *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq - a \longleftrightarrow a \leq 0$
proof −
from *add-le-cancel-left* [*of uminus a plus a a zero*]
have $(a <= -a) = (a+a <= 0)$
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *minus-le-self-iff*: $- a \leq a \longleftrightarrow 0 \leq a$
proof −
from *add-le-cancel-left* [*of uminus a zero plus a a*]
have $(-a <= a) = (0 <= a+a)$
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
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 $0 \leq |a|$
proof −
have *a*: $a \leq |a|$ **and** *b*: $-\ a \leq |a|$ **by** (*auto simp add*: *abs-lattice*)
show *?thesis* **by** (*rule add-mono* [*OF a b*, *simplified*])
qed
then have $0 \leq sup\ a\ (-\ a)$ **unfolding** *abs-lattice* **.**
then have $sup\ (sup\ a\ (-\ a))\ 0 = sup\ a\ (-\ a)$ **by** (*rule sup-absorb1*)
then show *?thesis*
by (*simp add*: *add-sup-inf-distribs sup-ACI*
pprt-def nprt-def diff-minus abs-lattice)
qed

subclass *pordered-ab-group-add-abs*
proof −
have *abs-ge-zero* [*simp*]: $\bigwedge a.\ 0 \leq |a|$
proof −
fix *a b*
have *a*: $a \leq |a|$ **and** *b*: $-\ a \leq |a|$ **by** (*auto simp add*: *abs-lattice*)
show $0 \leq |a|$ **by** (*rule add-mono* [*OF a b*, *simplified*])
qed
have *abs-leI*: $\bigwedge a\ b.\ a \leq b \Longrightarrow -\ a \leq b \Longrightarrow |a| \leq b$
by (*simp add*: *abs-lattice le-supI*)
show *?thesis*
proof *unfold-locales*
fix *a*
show $0 \leq |a|$ **by** *simp*
next
fix *a*
show $a \leq |a|$
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 ≤ b ⟹ − a ≤ b ⟹ |a| ≤ b by (erule abs-leI)
  next
    fix a b
    show |a + b| ≤ |a| + |b|
    proof −
      have g:abs a + abs b = sup (a+b) (sup (−a−b) (sup (−a+b) (a + (−b))))
(is -=sup ?m ?n)
        by (simp add: abs-lattice add-sup-inf-distribs sup-ACI diff-minus)
      have a:a+b <= sup ?m ?n by (simp)
      have b:−a−b <= ?n by (simp)
      have c:?n <= sup ?m ?n by (simp)
      from b c have d: −a−b <= sup ?m ?n by(rule order-trans)
      have e:−a−b = −(a+b) by (simp add: diff-minus)
      from a d e have abs(a+b) <= sup ?m ?n
        by (drule-tac abs-leI, auto)
      with g[symmetric] show ?thesis by simp
    qed
  qed 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' $\Longrightarrow$ (x $<$ y) $=$ (x' $<$ y')
by (*simp add*: *less-iff-diff-less-0*[*of* x y] *less-iff-diff-less-0*[*of* x' y'])

lemma *le-eqI*: (x::'a::*pordered-ab-group-add*) $-$ y $=$ x' $-$ y' $\Longrightarrow$ (y $<=$ x) $=$ (y' $<=$ x')
apply (*simp add*: *le-iff-diff-le-0*[*of* y x] *le-iff-diff-le-0*[*of* y' x'])
apply (*simp add*: *neg-le-iff-le*[*symmetric*, *of* $y-x$ 0] *neg-le-iff-le*[*symmetric*, *of* $y'-x'$ 0])
done

lemma *eq-eqI*: (x::'a::*ab-group-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-group-add*) $-$ y $==$ x $+$ ($-y$)
by (*simp add*: *diff-minus*)

lemma *add-minus-cancel*: (a::'a::*ab-group-add*) $+$ ($-a$ $+$ b) $=$ b
by (*simp add*: *add-assoc*[*symmetric*])

lemma *le-add-right-mono*:
assumes
a $<=$ b $+$ (c::'a::*pordered-ab-group-add*)
c $<=$ d
shows a $<=$ b $+$ d
apply (*rule-tac order-trans*[**where** y $=$ b+c])
apply (*simp-all add*: *prems*)
done

lemma *estimate-by-abs*:
a $+$ b $<=$ (c::'a::*lordered-ab-group-add-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

12.7 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 ⟨⟨
  Addsimprocs [ab-group-add-cancel.sum-conv, ab-group-add-cancel.rel-conv];
⟩⟩

end
```

13 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** *intro-locales*

end

class *comm-semiring-0-cancel* = *comm-semiring* + *comm-monoid-add* + *cancel-ab-semigroup-add*
begin

subclass *semiring-0-cancel* **by** *intro-locales*

end

class *zero-neq-one* = *zero* + *one* +
  assumes *zero-neq-one* [*simp*]: $0 \neq 1$
begin

lemma *one-neq-zero* [*simp*]: $1 \neq 0$
  by (*rule not-sym*) (*rule zero-neq-one*)

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult*

begin

subclass *semiring-1* **by** *intro-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** *intro-locales*

subclass *semiring-1* **by** *intro-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** *intro-locales*
subclass *comm-semiring-0-cancel* **by** *intro-locales*
subclass *comm-semiring-1* **by** *intro-locales*

end

class *ring* = *semiring* + *ab-group-add*
begin

subclass *semiring-0-cancel* **by** *intro-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** *intro-locales*
subclass *comm-semiring-0* **by** *intro-locales*

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* **by** *intro-locales*

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*

begin

subclass *ring-1* **by** *intro-locales*
subclass *comm-semiring-1-cancel* **by** *intro-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

Cancellation of equalities with a common factor

lemma *mult-cancel-right* [*simp*, *noatp*]:
$a * c = b * c \longleftrightarrow c = 0 \vee a = b$
proof −
have $(a * c = b * c) = ((a - b) * c = 0)$
by (*simp add*: *ring-distribs right-minus-eq*)
thus *?thesis*
by (*simp add*: *disj-commute right-minus-eq*)

qed

lemma *mult-cancel-left* [*simp*, *noatp*]:
  $c * a = c * b \longleftrightarrow c = 0 \vee a = b$
proof −
  have $(c * a = c * b) = (c * (a - b) = 0)$
    by (*simp add*: *ring-distribs right-minus-eq*)
  thus *?thesis*
    by (*simp add*: *right-minus-eq*)
qed

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

lemma *mult-cancel-right1* [*simp*]:
  $c = b * c \longleftrightarrow c = 0 \vee b = 1$
  by (*insert mult-cancel-right* [*of 1 c b*], *force*)

lemma *mult-cancel-right2* [*simp*]:
  $a * c = c \longleftrightarrow c = 0 \vee a = 1$
  by (*insert mult-cancel-right* [*of a c 1*], *simp*)

lemma *mult-cancel-left1* [*simp*]:
  $c = c * b \longleftrightarrow c = 0 \vee b = 1$
  by (*insert mult-cancel-left* [*of c 1 b*], *force*)

lemma *mult-cancel-left2* [*simp*]:
  $c * a = c \longleftrightarrow c = 0 \vee a = 1$
  by (*insert mult-cancel-left* [*of c a 1*], *simp*)

end

class *idom* = *comm-ring-1* + *no-zero-divisors*
begin

subclass *ring-1-no-zero-divisors* **by** *intro-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 ≠ 0 and b: b ≠ 0
  show a * b ≠ 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

lemma nonzero-imp-inverse-nonzero:
  a ≠ 0 ⟹ inverse a ≠ 0
proof
  assume ianz: inverse a = 0
  assume a ≠ 0
  hence 1 = a * inverse a by simp
  also have ... = 0 by (simp add: ianz)
  finally have 1 = 0 .
  thus False by (simp add: eq-commute)
qed

lemma inverse-zero-imp-zero:
  inverse a = 0 ⟹ a = 0
apply (rule classical)
apply (drule nonzero-imp-inverse-nonzero)
apply auto
done

lemma nonzero-inverse-minus-eq:
  assumes a ≠ 0
  shows inverse (- a) = - inverse a
proof -
  have - a * inverse (- a) = - a * - inverse a
    using assms by simp
  then show ?thesis unfolding mult-cancel-left using assms by simp
qed

lemma nonzero-inverse-inverse-eq:
  assumes a ≠ 0
  shows inverse (inverse a) = a
proof -
  have (inverse (inverse a) * inverse a) * a = a
    using assms by (simp add: nonzero-imp-inverse-nonzero)
  then show ?thesis using assms by (simp add: mult-assoc)
```

qed

lemma *nonzero-inverse-eq-imp-eq*:
assumes *inveq*: *inverse a* = *inverse b*
and *anz*: $a \neq 0$
and *bnz*: $b \neq 0$
shows $a = b$
proof −
have $a * inverse\ b = a * inverse\ a$
by (*simp add*: *inveq*)
hence $(a * inverse\ b) * b = (a * inverse\ a) * b$
by *simp*
then show $a = b$
by (*simp add*: *mult-assoc anz bnz*)
qed

lemma *inverse-1* [*simp*]: *inverse 1* = *1*
proof −
have *inverse 1 * 1* = *1*
by (*rule left-inverse*) (*rule one-neq-zero*)
then show *?thesis* **by** *simp*
qed

lemma *inverse-unique*:
assumes *ab*: $a * b = 1$
shows *inverse a* = *b*
proof −
have $a \neq 0$ **using** *ab* **by** (*cases a* = *0*) *simp-all*
moreover have *inverse a* ∗ (*a* ∗ *b*) = *inverse a* **by** (*simp add*: *ab*)
ultimately show *?thesis* **by** (*simp add*: *mult-assoc* [*symmetric*])
qed

lemma *nonzero-inverse-mult-distrib*:
assumes *anz*: $a \neq 0$
and *bnz*: $b \neq 0$
shows *inverse* (*a* ∗ *b*) = *inverse b* ∗ *inverse a*
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

lemma *division-ring-inverse-add*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ a + inverse\ b = inverse\ a * (a + b) * inverse\ b$

by (*simp add*: *ring-simps mult-assoc*)

lemma *division-ring-inverse-diff*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *inverse a* − *inverse b* = *inverse a* ∗ (*b* − *a*) ∗ *inverse b*
by (*simp add*: *ring-simps mult-assoc*)

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** *intro-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*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
assumes *mult-right-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * c$

class *pordered-semiring* = *mult-mono* + *semiring-0* + *pordered-ab-semigroup-add*

begin

lemma *mult-mono*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq b \Longrightarrow 0 \leq c$
$\Longrightarrow a * c \leq b * d$
apply (*erule mult-right-mono* [*THEN order-trans*], *assumption*)
apply (*erule mult-left-mono*, *assumption*)
done

lemma *mult-mono'*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq a \Longrightarrow 0 \leq c$
$\Longrightarrow a * c \leq b * d$
apply (*rule mult-mono*)
apply (*fast intro*: *order-trans*)+
done

end

class *pordered-cancel-semiring* = *mult-mono* + *pordered-ab-semigroup-add*
+ *semiring* + *comm-monoid-add* + *cancel-ab-semigroup-add*
begin

subclass *semiring-0-cancel* **by** *intro-locales*
subclass *pordered-semiring* **by** *intro-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*: $(0 \leq a \ \& \ b \leq 0) \mid (a \leq 0 \ \& \ 0 \leq b) \Longrightarrow a * b \leq 0$
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** *intro-locales*

subclass *pordered-comm-monoid-add* **by** *intro-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** *intro-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 ≤ c * b ⟹ 0 < c ⟹ a ≤ b*
by (*force simp add*: *mult-strict-left-mono -not-less* [*symmetric*])

lemma *mult-right-le-imp-le*:
*a * c ≤ b * c ⟹ 0 < c ⟹ a ≤ b*
by (*force simp add*: *mult-strict-right-mono not-less* [*symmetric*])

lemma *mult-pos-pos*:
*0 < a ⟹ 0 < b ⟹ 0 < a * b*
by (*drule mult-strict-left-mono* [*of zero b*], *auto*)

lemma *mult-pos-neg*:
*0 < a ⟹ b < 0 ⟹ a * b < 0*
by (*drule mult-strict-left-mono* [*of b zero*], *auto*)

lemma *mult-pos-neg2*:
*0 < a ⟹ b < 0 ⟹ b * a < 0*
by (*drule mult-strict-right-mono* [*of b zero*], *auto*)

lemma *zero-less-mult-pos*:
*0 < a * b ⟹ 0 < a ⟹ 0 < b*
apply (*cases b≤0*)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

lemma *zero-less-mult-pos2*:
*0 < b * a ⟹ 0 < a ⟹ 0 < b*
apply (*cases b≤0*)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg2* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes *a < b* **and** *c < d* **and** *0 < b* **and** *0 ≤ c*
shows *a * c < b * d*
using *assms* **apply** (*cases c=0*)
apply (*simp add*: *mult-pos-pos*)
apply (*erule mult-strict-right-mono* [*THEN less-trans*])
apply (*force simp add*: *le-less*)
apply (*erule mult-strict-left-mono*, *assumption*)
done

This weaker variant has more natural premises

lemma *mult-strict-mono'*:

assumes $a < b$ **and** $c < d$ **and** $0 \leq a$ **and** $0 \leq c$
shows $a * c < b * d$
by (*rule mult-strict-mono*) (*insert assms*, *auto*)

lemma *mult-less-le-imp-less*:
assumes $a < b$ **and** $c \leq d$ **and** $0 \leq a$ **and** $0 < c$
shows $a * c < b * d$
using *assms* **apply** (*subgoal-tac* $a * c < b * c$)
apply (*erule less-le-trans*)
apply (*erule mult-left-mono*)
apply *simp*
apply (*erule mult-strict-right-mono*)
apply *assumption*
done

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
using *assms* **apply** (*subgoal-tac* $a * c \leq b * c$)
apply (*erule le-less-trans*)
apply (*erule mult-strict-left-mono*)
apply *simp*
apply (*erule mult-right-mono*)
apply *simp*
done

lemma *mult-less-imp-less-left*:
assumes *less*: $c * a < c * b$ **and** *nonneg*: $0 \leq c$
shows $a < b$
proof (*rule ccontr*)
assume $\neg\ a < b$
hence $b \leq a$ **by** (*simp add*: *linorder-not-less*)
hence $c * b \leq c * a$ **using** *nonneg* **by** (*rule mult-left-mono*)
with *this* **and** *less* **show** *False*
by (*simp add*: *not-less* [*symmetric*])
qed

lemma *mult-less-imp-less-right*:
assumes *less*: $a * c < b * c$ **and** *nonneg*: $0 \leq c$
shows $a < b$
proof (*rule ccontr*)
assume $\neg\ a < b$
hence $b \leq a$ **by** (*simp add*: *linorder-not-less*)
hence $b * c \leq a * c$ **using** *nonneg* **by** (*rule mult-right-mono*)
with *this* **and** *less* **show** *False*
by (*simp add*: *not-less* [*symmetric*])
qed

end

class *mult-mono1* = *times* + *zero* + *ord* +
  assumes *mult-mono1*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$

class *pordered-comm-semiring* = *comm-semiring-0*
  + *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-semiring*
proof *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** *intro-locales*
subclass *pordered-cancel-semiring* **by** *intro-locales*

end

class *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add*
+
  assumes *mult-strict-left-mono-comm*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *ordered-semiring-strict*
proof *unfold-locales*
  fix a b c :: $'a$
  assume $a < b$ $0 < c$
  thus $c * a < c * b$ **by** (*rule mult-strict-left-mono-comm*)
  thus $a * c < b * c$ **by** (*simp only*: *mult-commute*)
qed

subclass *pordered-cancel-comm-semiring*
proof *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-left-mono* **by** (*cases* $c = 0$) *auto*
qed

end

class *pordered-ring* = *ring* + *pordered-cancel-semiring*
begin

subclass *pordered-ab-group-add* **by** *intro-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* + *uminus* + *ord* + *zero* + *abs* +
assumes *abs-if*: $|a| = (\mathit{if}\ a < 0\ \mathit{then} - a\ \mathit{else}\ a)$

class *sgn-if* = *minus* + *uminus* + *zero* + *one* + *ord* + *sgn* +
  assumes *sgn-if*: *sgn x* = (*if x* = *0 then 0 else if 0* < *x then 1 else* − *1*)

lemma (**in** *sgn-if*) *sgn0*[*simp*]: *sgn 0* = *0*
by(*simp add*:*sgn-if*)

class *ordered-ring* = *ring* + *ordered-semiring*
  + *ordered-ab-group-add* + *abs-if*
begin

subclass *pordered-ring* **by** *intro-locales*

subclass *pordered-ab-group-add-abs*
proof *unfold-locales*
  fix *a b*
  show |*a* + *b*| ≤ |*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** *intro-locales*

lemma *mult-strict-left-mono-neg*:
  b < *a* ⟹ *c* < *0* ⟹ *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* ⟹ *c* < *0* ⟹ *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* ⟹ *b* < *0* ⟹ *0* < *a* ∗ *b*
  by (*drule mult-strict-right-mono-neg*, *auto*)

subclass *ring-no-zero-divisors*

proof *unfold-locales*
fix *a b*
assume $a \neq 0$ **then have** *A*: $a < 0 \vee 0 < a$ **by** (*simp add*: *neq-iff*)
assume $b \neq 0$ **then have** *B*: $b < 0 \vee 0 < b$ **by** (*simp add*: *neq-iff*)
have $a * b < 0 \vee 0 < a * b$
proof (*cases* $a < 0$)
case *True* **note** $A' = this$
show *?thesis* **proof** (*cases* $b < 0$)
case *True* **with** A'
show *?thesis* **by** (*auto dest*: *mult-neg-neg*)
next
case *False* **with** *B* **have** $0 < b$ **by** *auto*
with A' **show** *?thesis* **by** (*auto dest*: *mult-strict-right-mono*)
qed
next
case *False* **with** *A* **have** A': $0 < a$ **by** *auto*
show *?thesis* **proof** (*cases* $b < 0$)
case *True* **with** A'
show *?thesis* **by** (*auto dest*: *mult-strict-right-mono-neg*)
next
case *False* **with** *B* **have** $0 < b$ **by** *auto*
with A' **show** *?thesis* **by** (*auto dest*: *mult-pos-pos*)
qed
qed
then show $a * b \neq 0$ **by** (*simp add*: *neq-iff*)
qed

lemma *zero-less-mult-iff*:
$0 < a * b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
apply (*auto simp add*: *mult-pos-pos mult-neg-neg*)
apply (*simp-all add*: *not-less le-less*)
apply (*erule disjE*) **apply** *assumption* **defer**
apply (*erule disjE*) **defer apply** (*drule sym*) **apply** *simp*
apply (*erule disjE*) **defer apply** (*drule sym*) **apply** *simp*
apply (*erule disjE*) **apply** *assumption* **apply** (*drule sym*) **apply** *simp*
apply (*drule sym*) **apply** *simp*
apply (*blast dest*: *zero-less-mult-pos*)
apply (*blast dest*: *zero-less-mult-pos2*)
done

lemma *zero-le-mult-iff*:
$0 \leq a * b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$
by (*auto simp add*: *eq-commute* [*of* 0] *le-less not-less zero-less-mult-iff*)

lemma *mult-less-0-iff*:
$a * b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$
apply (*insert zero-less-mult-iff* [*of* $-a$ *b*])
apply (*force simp add*: *minus-mult-left*[*symmetric*])
done

lemma *mult-le-0-iff*:
$a * b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$
apply (*insert zero-le-mult-iff* [*of* $-a$ b])
apply (*force simp add*: *minus-mult-left*[*symmetric*])
done

lemma *zero-le-square* [*simp*]: $0 \leq a * a$
by (*simp add*: *zero-le-mult-iff linear*)

lemma *not-square-less-zero* [*simp*]: $\neg (a * a < 0)$
by (*simp add*: *not-less*)

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations $\leq$ and equality.

These "disjunction" versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

lemma *mult-less-cancel-right-disj*:
$a * c < b * c \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
apply (*cases* $c = 0$)
apply (*auto simp add*: *neq-iff mult-strict-right-mono*
mult-strict-right-mono-neg)
apply (*auto simp add*: *not-less*
not-le [*symmetric*, *of* $a*c$]
not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *less-imp-le mult-right-mono*
mult-right-mono-neg)
done

lemma *mult-less-cancel-left-disj*:
$c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
apply (*cases* $c = 0$)
apply (*auto simp add*: *neq-iff mult-strict-left-mono*
mult-strict-left-mono-neg)
apply (*auto simp add*: *not-less*
not-le [*symmetric*, *of* $c*a$]
not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *less-imp-le mult-left-mono*
mult-left-mono-neg)
done

The "conjunction of implication" lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma *mult-less-cancel-right*:
$a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-right-disj* [*of* a c b] **by** *auto*

lemma *mult-less-cancel-left*:
$c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-left-disj* [*of c a b*] **by** *auto*

lemma *mult-le-cancel-right*:
$a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-right-disj*)

lemma *mult-le-cancel-left*:
$c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-left-disj*)

end

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** *intro-locales*
subclass *pordered-cancel-comm-semiring* **by** *intro-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*

lemma *zero-le-one* [*simp*]: $0 \leq 1$
by (*rule zero-less-one* [*THEN less-imp-le*])

lemma *not-one-le-zero* [*simp*]: $\neg\ 1 \leq 0$
by (*simp add*: *not-le*)

lemma *not-one-less-zero* [*simp*]: $\neg\ 1 < 0$
by (*simp add*: *not-less*)

lemma *less-1-mult*:

assumes $1 < m$ **and** $1 < n$
shows $1 < m * n$
using *assms mult-strict-mono* [*of 1 m 1 n*]
by (*simp add*: *less-trans* [*OF zero-less-one*])

end

class *ordered-idom* = *comm-ring-1* +
ordered-comm-semiring-strict + *ordered-ab-group-add* +
abs-if + *sgn-if*

begin

subclass *ordered-ring-strict* **by** *intro-locales*
subclass *pordered-comm-ring* **by** *intro-locales*
subclass *idom* **by** *intro-locales*

subclass *ordered-semidom*
proof *unfold-locales*
have $0 \leq 1 * 1$ **by** (*rule zero-le-square*)
thus $0 < 1$ **by** (*simp add*: *le-less*)
qed

lemma *linorder-neqE-ordered-idom*:
assumes $x \neq y$ **obtains** $x < y \mid y < x$
using *assms* **by** (*rule neqE*)

These cancellation simprules also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*:
$c \leq b * c \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
by (*insert mult-le-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-le-cancel-right2*:
$a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
by (*insert mult-le-cancel-right* [*of a c 1*], *simp*)

lemma *mult-le-cancel-left1*:
$c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
by (*insert mult-le-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-le-cancel-left2*:
$c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
by (*insert mult-le-cancel-left* [*of c a 1*], *simp*)

lemma *mult-less-cancel-right1*:
$c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
by (*insert mult-less-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-less-cancel-right2*:
$a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
by (*insert mult-less-cancel-right* [*of a c 1*], *simp*)

lemma *mult-less-cancel-left1*:
$c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
by (*insert mult-less-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-less-cancel-left2*:
$c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
by (*insert mult-less-cancel-left* [*of c a 1*], *simp*)

end

class *ordered-field* = *field* + *ordered-idom*

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

— FIXME continue localization here

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
$(inverse\ a = 0) = (a = (0::'a::\{division\text{-}ring, division\text{-}by\text{-}zero\}))$
by (*force dest*: *inverse-zero-imp-zero*)

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≠0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-minus-eq*)
qed

lemma *inverse-eq-imp-eq*:
$inverse\ a = inverse\ b \Longrightarrow a = (b::'a::\{division\text{-}ring, division\text{-}by\text{-}zero\})$
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-ring*,*division-by-zero*}))
by (*force dest*!: *inverse-eq-imp-eq*)

lemma *inverse-inverse-eq* [*simp*]:
inverse(*inverse* (*a*::'*a*::{*division-ring*,*division-by-zero*})) = *a*
proof *cases*
assume *a*=*0* **thus** *?thesis* **by** *simp*
next
assume *a*≠*0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [*simp*]:
inverse(*a*∗*b*) = *inverse*(*a*) ∗ *inverse*(*b*::'*a*::{*field*,*division-by-zero*})
proof *cases*
assume *a* ≠ *0* & *b* ≠ *0*
thus *?thesis*
by (*simp add*: *nonzero-inverse-mult-distrib mult-commute*)
next
assume ~ (*a* ≠ *0* & *b* ≠ *0*)
thus *?thesis*
by *force*
qed

There is no slick version using division by zero.

lemma *inverse-add*:
[|*a* ≠ *0*; *b* ≠ *0*|]
==> *inverse a* + *inverse b* = (*a*+*b*) ∗ *inverse a* ∗ *inverse* (*b*::'*a*::*field*)
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*)

13.1 Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

lemma *nonzero-mult-divide-mult-cancel-left*[*simp*,*noatp*]:
assumes [*simp*]: *b*≠*0* **and** [*simp*]: *c*≠*0* **shows** (*c*∗*a*)/(*c*∗*b*) = *a*/(*b*::'*a*::*field*)
proof −
have (*c*∗*a*)/(*c*∗*b*) = *c* ∗ *a* ∗ (*inverse b* ∗ *inverse c*)
by (*simp add*: *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≠0 ==> (c*a) / (c*b) = a / (b::'a::{field,division-by-zero})
apply (cases b = 0)
apply (simp-all add: nonzero-mult-divide-mult-cancel-left)
done

lemma nonzero-mult-divide-mult-cancel-right [noatp]:
  [|b≠0; c≠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≠0 ==> (a*c) / (b*c) = a / (b::'a::{field,division-by-zero})
apply (cases b = 0)
apply (simp-all add: nonzero-mult-divide-mult-cancel-right)
done

lemma divide-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-by-zero})
by (simp add: divide-inverse mult-ac)

lemma divide-divide-eq-left [simp,noatp]:
  (a / b) / (c::'a::{field,division-by-zero}) = a / (b*c)
by (simp add: divide-inverse mult-assoc)

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

13.1.1 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 ≠ *0* ⟹ *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 ≠ *0* ⟹ *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*≠*0*; *b*≠*0* ⟧ ⟹ *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*≠*0*; *b*≠*0* ⟧ ⟹ *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*≠*0*; *c*≠*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*≠*0*; *c*≠*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*)

13.2 Division and Unary Minus

lemma *nonzero-minus-divide-left*: *b* ≠ *0* ==> − (*a*/*b*) = (−*a*) / (*b*::*'a*::*field*)
by (*simp add*: *divide-inverse minus-mult-left*)

lemma *nonzero-minus-divide-right*: *b* ≠ *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* ≠ *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*: $c \neq 0 \Longrightarrow (b/c = (a::'a::field)) = (b = a*c)$
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*: $(c::'a::\{division\text{-}by\text{-}zero,field\}) \sim= 0 \Longrightarrow$
$b = a * c \Longrightarrow b / c = a$
by (*subst divide-eq-eq*, *simp*)

lemma *eq-divide-imp*: $(c::'a::\{division\text{-}by\text{-}zero,field\}) \sim= 0 \Longrightarrow$
$a * c = b \Longrightarrow a = b / c$
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*: $(y::'a::field) \sim= 0 \Longrightarrow z \sim= 0 \Longrightarrow$
$x / y - w / z = (x * z - w * y) / (y * z)$
by (*simp add:field-eq-simps times-divide-eq*)

lemma *frac-eq-eq*: $(y::'a::field) \sim= 0 \Longrightarrow z \sim= 0 \Longrightarrow$
$(x / y = w / z) = (x * z = w * y)$

by (*simp add*:*field-eq-simps times-divide-eq*)

13.3 Ordered Fields

lemma *positive-imp-inverse-positive*:
assumes *a-gt-0*: $0 < a$ **shows** $0 < inverse\ (a::'a::ordered\text{-}field)$
proof −
have $0 < a * inverse\ a$
by (*simp add*: *a-gt-0* [*THEN order-less-imp-not-eq2*] *zero-less-one*)
thus $0 < inverse\ a$
by (*simp add*: *a-gt-0* [*THEN order-less-not-sym*] *zero-less-mult-iff*)
qed

lemma *negative-imp-inverse-negative*:
$a < 0 ==> inverse\ a < (0::'a::ordered\text{-}field)$
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 ~ $b \leq a$
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* ≠ *0*
shows *a* < (*0*::'*a*::*ordered-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* ≤ *inverse a*) = (*0* ≤ (*a*::'*a*::{*ordered-field*,*division-by-zero*}))
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
(*inverse a* ≤ *0*) = (*a* ≤ (*0*::'*a*::{*ordered-field*,*division-by-zero*}))
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *ordered-field-no-lb*: ∀ *x*. ∃ *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 ∃ *y*. *y* < *x* **by** *blast*
qed

lemma *ordered-field-no-ub*: ∀ *x*. ∃ *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 ∃ *y*. *y* > *x* **by** *blast*
qed

13.4 Anti-Monotonicity of *inverse*

lemma *less-imp-inverse-less*:
assumes *less*: *a* < *b* **and** *apos*: *0* < *a*
shows *inverse b* < *inverse* (*a*::'*a*::*ordered-field*)
proof (*rule ccontr*)
assume ~ *inverse b* < *inverse a*
hence *inverse a* ≤ *inverse b*
by (*simp add*: *linorder-not-less*)

hence ~ $(a < b)$
by (*simp add*: *linorder-not-less inverse-le-imp-le* [*OF* - *apos*])
thus *False*
by (*rule notE* [*OF* - *less*])
qed

lemma *inverse-less-imp-less*:
$[|inverse\ a < inverse\ b;\ 0 < a|] ==> b < (a::'a::ordered\text{-}field)$
apply (*simp add*: *order-less-le* [*of inverse a*] *order-less-le* [*of b*])
apply (*force dest*!: *inverse-le-imp-le nonzero-inverse-eq-imp-eq*)
done

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*,*noatp*]:
$[|0 < a;\ 0 < b|] ==> (inverse\ a < inverse\ b) = (b < (a::'a::ordered\text{-}field))$
by (*blast intro*: *less-imp-inverse-less dest*: *inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
$[|a \leq b;\ 0 < a|] ==> inverse\ b \leq inverse\ (a::'a::ordered\text{-}field)$
by (*force simp add*: *order-le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le* [*simp*,*noatp*]:
$[|0 < a;\ 0 < b|] ==> (inverse\ a \leq inverse\ b) = (b \leq (a::'a::ordered\text{-}field))$
by (*blast intro*: *le-imp-inverse-le dest*: *inverse-le-imp-le*)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
$[|inverse\ a \leq inverse\ b;\ b < 0|] ==> b \leq (a::'a::ordered\text{-}field)$
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*:
$[|a \le b; b < 0|] ==> inverse\ b \le inverse\ (a::'a::ordered\text{-}field)$
by (*force simp add*: *order-le-less less-imp-inverse-less-neg*)

lemma *inverse-le-iff-le-neg* [*simp*,*noatp*]:
$[|a < 0; b < 0|] ==> (inverse\ a \le inverse\ b) = (b \le (a::'a::ordered\text{-}field))$
by (*blast intro*: *le-imp-inverse-le-neg dest*: *inverse-le-imp-le-neg*)

13.5 Inverses and the Number One

lemma *one-less-inverse-iff*:
$(1 < inverse\ x) = (0 < x\ \&\ x < (1::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
proof *cases*
assume $0 < x$
with *inverse-less-iff-less* [*OF zero-less-one*, *of x*]
show *?thesis* **by** *simp*
next
assume *notless*: $\sim (0 < x)$
have $\sim (1 < inverse\ x)$
proof
assume $1 < inverse\ x$
also with *notless* **have** $... \le 0$ **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*:
$(1 \le inverse\ x) = (0 < x\ \&\ x \le (1::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}))$
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* ≤ *0* | *1* < (*x*::'*a*::{*ordered-field*,*division-by-zero*}))
by (*simp add*: *linorder-not-le* [*symmetric*] *one-le-inverse-iff*)

lemma *inverse-le-1-iff*:
(*inverse x* ≤ *1*) = (*x* ≤ *0* | *1* ≤ (*x*::'*a*::{*ordered-field*,*division-by-zero*}))
by (*simp add*: *linorder-not-less* [*symmetric*] *one-less-inverse-iff*)

13.6 Simplification of Inequalities Involving Literal Divisors

lemma *pos-le-divide-eq*: *0* < (*c*::'*a*::*ordered-field*) ==> (*a* ≤ *b*/*c*) = (*a*∗*c* ≤ *b*)
proof −
assume *less*: *0*<*c*
hence (*a* ≤ *b*/*c*) = (*a*∗*c* ≤ (*b*/*c*)∗*c*)
by (*simp add*: *mult-le-cancel-right order-less-not-sym* [*OF less*])
also have ... = (*a*∗*c* ≤ *b*)
by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-le-divide-eq*: *c* < (*0*::'*a*::*ordered-field*) ==> (*a* ≤ *b*/*c*) = (*b* ≤ *a*∗*c*)
proof −
assume *less*: *c*<*0*
hence (*a* ≤ *b*/*c*) = ((*b*/*c*)∗*c* ≤ *a*∗*c*)
by (*simp add*: *mult-le-cancel-right order-less-not-sym* [*OF less*])
also have ... = (*b* ≤ *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*: $c < (0::'a::ordered\text{-}field) ==> (b/c \leq a) = (a*c \leq b)$
proof −
assume *less*: $c<0$
hence $(b/c \leq a) = (a*c \leq (b/c)*c)$
by (*simp add*: *mult-le-cancel-right order-less-not-sym* [*OF less*])
also have ... $= (a*c \leq b)$
by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *divide-le-eq*:
$(b/c \leq a) =$
(*if* $0 < c$ *then* $b \leq a*c$
else if $c < 0$ *then* $a*c \leq b$
else $0 \leq (a::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-divide-le-eq neg-divide-le-eq linorder-neq-iff*)
done

lemma *pos-less-divide-eq*:
$0 < (c::'a::ordered\text{-}field) ==> (a < b/c) = (a*c < b)$
proof −
assume *less*: $0<c$
hence $(a < b/c) = (a*c < (b/c)*c)$
by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
also have ... $= (a*c < b)$
by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-less-divide-eq*:
$c < (0::'a::ordered\text{-}field) ==> (a < b/c) = (b < a*c)$
proof −
assume *less*: $c<0$
hence $(a < b/c) = ((b/c)*c < a*c)$
by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
also have ... $= (b < a*c)$
by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *less-divide-eq*:
$(a < b/c) =$
(*if* $0 < c$ *then* $a*c < b$
else if $c < 0$ *then* $b < a*c$
else $a < (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-less-divide-eq neg-less-divide-eq linorder-neq-iff*)
done

lemma *pos-divide-less-eq*:
  $0 < (c::'a::ordered\text{-}field) ==> (b/c < a) = (b < a*c)$
proof −
 assume *less*: $0<c$
 hence $(b/c < a) = ((b/c)*c < a*c)$
  by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
 also have ... $= (b < a*c)$
  by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
 finally show *?thesis* **.**
qed

lemma *neg-divide-less-eq*:
 $c < (0::'a::ordered\text{-}field) ==> (b/c < a) = (a*c < b)$
proof −
 assume *less*: $c<0$
 hence $(b/c < a) = (a*c < (b/c)*c)$
  by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
 also have ... $= (a*c < b)$
  by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
 finally show *?thesis* **.**
qed

lemma *divide-less-eq*:
 $(b/c < a) =$
  (*if* $0 < c$ *then* $b < a*c$
   else if $c < 0$ *then* $a*c < b$
   else $0 < (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-divide-less-eq neg-divide-less-eq linorder-neq-iff*)
done

13.7 Field simplification

Lemmas *field-simps* multiply with denominators in in(equations) if they can be proved to be non-zero (for equations) or positive/negative (for inequations).

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*

13.8 Division and Signs

lemma *zero-less-divide-iff*:
$((0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}) < a/b) = (0 < a \ \& \ 0 < b \mid a < 0 \ \& \ b < 0)$
by (*simp add*: *divide-inverse zero-less-mult-iff*)

lemma *divide-less-0-iff*:
$(a/b < (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})) =$
$(0 < a \ \& \ b < 0 \mid a < 0 \ \& \ 0 < b)$
by (*simp add*: *divide-inverse mult-less-0-iff*)

lemma *zero-le-divide-iff*:
$((0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\}) \leq a/b) =$
$(0 \leq a \ \& \ 0 \leq b \mid a \leq 0 \ \& \ b \leq 0)$
by (*simp add*: *divide-inverse zero-le-mult-iff*)

lemma *divide-le-0-iff*:
$(a/b \leq (0::'a::\{ordered\text{-}field,division\text{-}by\text{-}zero\})) =$
$(0 \leq a \ \& \ b \leq 0 \mid a \leq 0 \ \& \ 0 \leq b)$
by (*simp add*: *divide-inverse mult-le-0-iff*)

lemma *divide-eq-0-iff* [*simp*,*noatp*]:
$(a/b = 0) = (a=0 \mid b=(0::'a::\{field,division\text{-}by\text{-}zero\}))$
by (*simp add*: *divide-inverse*)

lemma *divide-pos-pos*:
$0 < (x::'a::ordered\text{-}field) \Longrightarrow 0 < y \Longrightarrow 0 < x \ / \ y$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-pos*:
$0 <= (x::'a::ordered\text{-}field) \Longrightarrow 0 < y \Longrightarrow 0 <= x \ / \ y$
by(*simp add*:*field-simps*)

lemma *divide-neg-pos*:
$(x::'a::ordered\text{-}field) < 0 \Longrightarrow 0 < y \Longrightarrow x \ / \ y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-pos*:
$(x::'a::ordered\text{-}field) <= 0 \Longrightarrow 0 < y \Longrightarrow x \ / \ y <= 0$
by(*simp add*:*field-simps*)

lemma *divide-pos-neg*:
$0 < (x::'a::ordered\text{-}field) \Longrightarrow y < 0 \Longrightarrow x \ / \ y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-neg*:
0 <= (x::'a::ordered-field) ==> y < 0 ==> x / y <= 0
by(*simp add*:*field-simps*)

lemma *divide-neg-neg*:
(x::'a::ordered-field) < 0 ==> y < 0 ==> 0 < x / y
by(*simp add*:*field-simps*)

lemma *divide-nonpos-neg*:
(x::'a::ordered-field) <= 0 ==> y < 0 ==> 0 <= x / y
by(*simp add*:*field-simps*)

13.9 Cancellation Laws for Division

lemma *divide-cancel-right* [*simp*,*noatp*]:
(a/c = b/c) = (c = 0 | a = (b::'a::{field,division-by-zero}))
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)
done

lemma *divide-cancel-left* [*simp*,*noatp*]:
(c/a = c/b) = (c = 0 | a = (b::'a::{field,division-by-zero}))
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)
done

13.10 Division and the Number One

Simplify expressions equated with 1

lemma *divide-eq-1-iff* [*simp*,*noatp*]:
(a/b = 1) = (b ≠ 0 & a = (b::'a::{field,division-by-zero}))
apply (*cases b=0*, *simp*)
apply (*simp add*: *right-inverse-eq*)
done

lemma *one-eq-divide-iff* [*simp*,*noatp*]:
(1 = a/b) = (b ≠ 0 & a = (b::'a::{field,division-by-zero}))
by (*simp add*: *eq-commute* [*of 1*])

lemma *zero-eq-1-divide-iff* [*simp*,*noatp*]:
((0::'a::{ordered-field,division-by-zero}) = 1/a) = (a = 0)
apply (*cases a=0*, *simp*)
apply (*auto simp add*: *nonzero-eq-divide-eq*)
done

lemma *one-divide-eq-0-iff* [*simp*,*noatp*]:
(1/a = (0::'a::{ordered-field,division-by-zero})) = (a = 0)
apply (*cases a=0*, *simp*)
apply (*insert zero-neq-one* [*THEN not-sym*])

apply (*auto simp add*: *nonzero-divide-eq-eq*)
done

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemmas *zero-less-divide-1-iff* = *zero-less-divide-iff* [*of 1*, *simplified*]
lemmas *divide-less-0-1-iff* = *divide-less-0-iff* [*of 1*, *simplified*]
lemmas *zero-le-divide-1-iff* = *zero-le-divide-iff* [*of 1*, *simplified*]
lemmas *divide-le-0-1-iff* = *divide-le-0-iff* [*of 1*, *simplified*]

declare *zero-less-divide-1-iff* [*simp*]
declare *divide-less-0-1-iff* [*simp*,*noatp*]
declare *zero-le-divide-1-iff* [*simp*]
declare *divide-le-0-1-iff* [*simp*,*noatp*]

13.11 Ordering Rules for Division

lemma *divide-strict-right-mono*:
[|*a* < *b*; *0* < *c*|] ==> *a* / *c* < *b* / (*c*::'*a*::*ordered-field*)
by (*simp add*: *order-less-imp-not-eq2 divide-inverse mult-strict-right-mono positive-imp-inverse-positive*)

lemma *divide-right-mono*:
[|*a* ≤ *b*; *0* ≤ *c*|] ==> *a*/*c* ≤ *b*/(*c*::'*a*::{*ordered-field*,*division-by-zero*})
by (*force simp add*: *divide-strict-right-mono order-le-less*)

lemma *divide-right-mono-neg*: (*a*::'*a*::{*division-by-zero*,*ordered-field*}) <= *b*
==> *c* <= *0* ==> *b* / *c* <= *a* / *c*
apply (*drule divide-right-mono* [*of* - - − *c*])
apply *auto*
done

lemma *divide-strict-right-mono-neg*:
[|*b* < *a*; *c* < *0*|] ==> *a* / *c* < *b* / (*c*::'*a*::*ordered-field*)
apply (*drule divide-strict-right-mono* [*of* - - −*c*], *simp*)
apply (*simp add*: *order-less-imp-not-eq nonzero-minus-divide-right* [*symmetric*])
done

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:
[|*b* < *a*; *0* < *c*; *0* < *a*∗*b*|] ==> *c* / *a* < *c* / (*b*::'*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono*)

lemma *divide-left-mono*:
[|*b* ≤ *a*; *0* ≤ *c*; *0* < *a*∗*b*|] ==> *c* / *a* ≤ *c* / (*b*::'*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-right-mono*)

lemma *divide-left-mono-neg*: (*a*::'*a*::{*division-by-zero*,*ordered-field*}) <= *b*
==> *c* <= *0* ==> *0* < *a* ∗ *b* ==> *c* / *a* <= *c* / *b*
apply (*drule divide-left-mono* [*of* - - − *c*])

apply (*auto simp add*: *mult-commute*)
done

lemma *divide-strict-left-mono-neg*:
[|*a* < *b*; *c* < *0*; *0* < *a*∗*b*|] ==> *c* / *a* < *c* / (*b*::′*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono-neg*)

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*noatp*]:
fixes *a* :: ′*a* :: {*ordered-field*,*division-by-zero*}
shows (*1* ≤ *b* / *a*) = ((*0* < *a* & *a* ≤ *b*) | (*a* < *0* & *b* ≤ *a*))
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1* [*noatp*]:
fixes *a* :: ′*a* :: {*ordered-field*,*division-by-zero*}
shows (*b* / *a* ≤ *1*) = ((*0* < *a* & *b* ≤ *a*) | (*a* < *0* & *a* ≤ *b*) | *a*=*0*)
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1* [*noatp*]:
fixes *a* :: ′*a* :: {*ordered-field*,*division-by-zero*}
shows (*1* < *b* / *a*) = ((*0* < *a* & *a* < *b*) | (*a* < *0* & *b* < *a*))
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1* [*noatp*]:
fixes *a* :: ′*a* :: {*ordered-field*,*division-by-zero*}
shows (*b* / *a* < *1*) = ((*0* < *a* & *b* < *a*) | (*a* < *0* & *a* < *b*) | *a*=*0*)
by (*auto simp add*: *divide-less-eq*)

13.12 Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*,*noatp*]:
fixes *a* :: ′*a* :: {*ordered-field*,*division-by-zero*}
shows *0* < *a* ⟹ (*1* ≤ *b*/*a*) = (*a* ≤ *b*)
by (*auto simp add*: *le-divide-eq*)

lemma *le-divide-eq-1-neg* [*simp*,*noatp*]:
fixes *a* :: ′*a* :: {*ordered-field*,*division-by-zero*}
shows *a* < *0* ⟹ (*1* ≤ *b*/*a*) = (*b* ≤ *a*)
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1-pos* [*simp*,*noatp*]:
fixes *a* :: ′*a* :: {*ordered-field*,*division-by-zero*}
shows *0* < *a* ⟹ (*b*/*a* ≤ *1*) = (*b* ≤ *a*)
by (*auto simp add*: *divide-le-eq*)

lemma *divide-le-eq-1-neg* [*simp*,*noatp*]:
fixes *a* :: ′*a* :: {*ordered-field*,*division-by-zero*}
shows *a* < *0* ⟹ (*b*/*a* ≤ *1*) = (*a* ≤ *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*)

13.13 Reasoning about inequalities with division

lemma *mult-right-le-one-le*: *0 <= (x::'a::ordered-idom) ==> 0 <= y ==> y <= 1*
*==> x * y <= x*
by (*auto simp add*: *mult-compare-simps*)

lemma *mult-left-le-one-le*: *0 <= (x::'a::ordered-idom) ==> 0 <= y ==> y <= 1*
*==> y * x <= x*
by (*auto simp add*: *mult-compare-simps*)

lemma *mult-imp-div-pos-le*: *0 < (y::'a::ordered-field) ==> x <= z * y ==>*
x / y <= z
by (*subst pos-divide-le-eq*, *assumption+*)

lemma *mult-imp-le-div-pos*: *0 < (y::'a::ordered-field) ==> z * y <= x ==>*
z <= x / y

by(*simp add*:*field-simps*)

lemma *mult-imp-div-pos-less*: *0* < (*y*::*'a*::*ordered-field*) ==> *x* < *z* * *y* ==>
x / *y* < *z*
by(*simp add*:*field-simps*)

lemma *mult-imp-less-div-pos*: *0* < (*y*::*'a*::*ordered-field*) ==> *z* * *y* < *x* ==>
z < *x* / *y*
by(*simp add*:*field-simps*)

lemma *frac-le*: (*0*::*'a*::*ordered-field*) <= *x* ==>
x <= *y* ==> *0* < *w* ==> *w* <= *z* ==> *x* / *z* <= *y* / *w*
apply (*rule mult-imp-div-pos-le*)
apply *simp*
apply (*subst times-divide-eq-left*)
apply (*rule mult-imp-le-div-pos*, *assumption*)
apply (*rule mult-mono*)
apply *simp-all*
done

lemma *frac-less*: (*0*::*'a*::*ordered-field*) <= *x* ==>
x < *y* ==> *0* < *w* ==> *w* <= *z* ==> *x* / *z* < *y* / *w*
apply (*rule mult-imp-div-pos-less*)
apply *simp*
apply (*subst times-divide-eq-left*)
apply (*rule mult-imp-less-div-pos*, *assumption*)
apply (*erule mult-less-le-imp-less*)
apply *simp-all*
done

lemma *frac-less2*: (*0*::*'a*::*ordered-field*) < *x* ==>
x <= *y* ==> *0* < *w* ==> *w* < *z* ==> *x* / *z* < *y* / *w*
apply (*rule mult-imp-div-pos-less*)
apply *simp-all*
apply (*subst times-divide-eq-left*)
apply (*rule mult-imp-less-div-pos*, *assumption*)
apply (*erule mult-le-less-imp-less*)
apply *simp-all*
done

It's not obvious whether these should be simprules or not. Their effect is to gather terms into one big fraction, like a*b*c / x*y*z. The rationale for that is unclear, but many proofs seem to need them.

declare *times-divide-eq* [*simp*]

13.14 Ordered Fields are Dense

context *ordered-semidom*
begin

lemma *less-add-one*: $a < a + 1$
proof −
  have $a + 0 < a + 1$
    by (*blast intro*: *zero-less-one add-strict-left-mono*)
  thus *?thesis* **by** *simp*
qed

lemma *zero-less-two*: $0 < 1 + 1$
  by (*blast intro*: *less-trans zero-less-one less-add-one*)

end

lemma *less-half-sum*: $a < b \Longrightarrow a < (a+b) \,/\, (1+1::'a::ordered\text{-}field)$
by (*simp add*: *field-simps zero-less-two*)

lemma *gt-half-sum*: $a < b \Longrightarrow (a+b)/(1+1::'a::ordered\text{-}field) < b$
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

13.15 Absolute Value

context *ordered-idom*
begin

lemma *mult-sgn-abs*: $sgn\ x * abs\ x = x$
  unfolding *abs-if sgn-if* **by** *auto*

end

lemma *abs-one* [*simp*]: $abs\ 1 = (1::'a::ordered\text{-}idom)$
  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** *intro-locales*
subclass *lordered-ab-group-add-join* **by** *intro-locales*

end

lemma *abs-le-mult*: *abs* (*a* * *b*) ≤ (*abs a*) * (*abs* (*b*::'*a*::*lordered-ring*))
proof −
  let *?x* = *pprt a* * *pprt b* − *pprt a* * *nprt b* − *nprt a* * *pprt b* + *nprt a* * *nprt b*
  let *?y* = *pprt a* * *pprt b* + *pprt a* * *nprt b* + *nprt a* * *pprt b* + *nprt a* * *nprt b*
  have *a*: (*abs a*) * (*abs b*) = *?x*
    by (*simp only*: *abs-prts*[*of a*] *abs-prts*[*of b*] *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

13.16 Bounds of products via negative and positive Part

lemma *mult-le-prts*:
assumes
a1 <= (*a*::'*a*::*lordered-ring*)
a <= *a2*
b1 <= *b*
b <= *b2*
shows
a * *b* <= *pprt a2* * *pprt b2* + *pprt a1* * *nprt b2* + *nprt a2* * *pprt b1* + *nprt a1* * *nprt b1*
proof −
have *a* * *b* = (*pprt a* + *nprt a*) * (*pprt b* + *nprt b*)
apply (*subst prts*[*symmetric*])+
apply *simp*
done
then have *a* * *b* = *pprt a* * *pprt b* + *pprt a* * *nprt b* + *nprt a* * *pprt b* + *nprt a* * *nprt b*
by (*simp add*: *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), sim-
plified nprt-neg pprt-neg]
  have le: − (a * b) <= − nprt a1 * pprt b2 + − nprt a2 * nprt b2 + − pprt a1
* pprt b1 + − pprt a2 * nprt b1 by simp
  then have −(− nprt a1 * pprt b2 + − nprt a2 * nprt b2 + − pprt a1 * pprt
b1 + − pprt a2 * nprt b1) <= a * b
    by (simp only: minus-le-iff)
  then show ?thesis by simp
qed

end
```

14 Nat: Natural numbers

```
theory Nat
imports Inductive 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)
begin
```

14.1 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* ≠ *Zero-Rep*

14.2 Type nat

Type definition

inductive *Nat* :: *ind* ⇒ *bool*
where
    Zero-RepI: *Nat Zero-Rep*
  | *Suc-RepI*: *Nat i* ⟹ *Nat* (*Suc-Rep i*)

global

typedef (**open** *Nat*)
  nat = *Collect Nat*
  by (*rule exI*, *rule CollectI*, *rule Nat.Zero-RepI*)

constdefs
  Suc :: *nat* => *nat*
  Suc-def: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat n*)))

local

instantiation *nat* :: *zero*
begin

definition *Zero-nat-def* [*code func del*]:
  0 = *Abs-Nat Zero-Rep*

instance ..

end

lemma *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 CollectD*, *THEN Nat.induct*])
  apply (*iprover elim*: *Abs-Nat-inverse* [*OF CollectI*, *THEN subst*])
  done

lemma *Suc-not-Zero* [*iff*]: *Suc m* ≠ *0*
by (*simp add*: *Zero-nat-def Suc-def*
Abs-Nat-inject Rep-Nat [*THEN CollectD*] *Suc-RepI Zero-RepI*
Suc-Rep-not-Zero-Rep)

lemma *Zero-not-Suc* [*iff*]: *0* ≠ *Suc m*
by (*rule not-sym*, *rule Suc-not-Zero not-sym*)

lemma *inj-Suc*[*simp*]: *inj-on Suc N*
by (*simp add*: *Suc-def inj-on-def Abs-Nat-inject Rep-Nat* [*THEN CollectD*] *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 *n-not-Suc-n*: *n* ≠ *Suc n*
by (*induct n*) *simp-all*

lemma *Suc-n-not-n*: *Suc n* ≠ *n*
by (*rule not-sym*, *rule n-not-Suc-n*)

A special form of induction for reasoning about *m* < *n* and *m* − *n*

lemma *diff-induct*: (!!*x*. *P x 0*) ==> (!!*y*. *P 0* (*Suc y*)) ==>
(!!*x y*. *P x y* ==> *P* (*Suc x*) (*Suc y*)) ==> *P m n*
apply (*rule-tac x* = *m* **in** *spec*)

apply (*induct n*)
prefer *2*
apply (*rule allI*)
apply (*induct-tac x*, *iprover+*)
done

14.3 Arithmetic operators

instantiation *nat* :: {*minus*, *comm-monoid-add*}
begin

primrec *plus-nat*
where
add-0: $0 + n = (n::nat)$
| *add-Suc*: $Suc\ m + n = Suc\ (m + n)$

lemma *add-0-right* [*simp*]: $m + 0 = (m::nat)$
by (*induct m*) *simp-all*

lemma *add-Suc-right* [*simp*]: $m + Suc\ n = Suc\ (m + n)$
by (*induct m*) *simp-all*

lemma *add-Suc-shift* [*code*]: $Suc\ m + n = m + Suc\ n$
by *simp*

primrec *minus-nat*
where
diff-0: $m - 0 = (m::nat)$
| *diff-Suc*: $m - Suc\ n = (case\ m - n\ of\ 0 => 0 \mid Suc\ k => k)$

declare *diff-Suc* [*simp del*, *code del*]

lemma *diff-0-eq-0* [*simp*, *code*]: $0 - n = (0::nat)$
by (*induct n*) (*simp-all add*: *diff-Suc*)

lemma *diff-Suc-Suc* [*simp*, *code*]: $Suc\ m - Suc\ n = m - n$
by (*induct n*) (*simp-all add*: *diff-Suc*)

instance proof
fix *n m q* :: *nat*
show $(n + m) + q = n + (m + q)$ **by** (*induct n*) *simp-all*
show $n + m = m + n$ **by** (*induct n*) *simp-all*
show $0 + n = n$ **by** *simp*
qed

end

instantiation *nat* :: *comm-semiring-1-cancel*
begin

definition
One-nat-def [*simp*]: *1 = Suc 0*

primrec *times-nat*
where
mult-0: *0 * n = (0::nat)*
| *mult-Suc*: *Suc m * n = n + (m * n)*

lemma *mult-0-right* [*simp*]: *(m::nat) * 0 = 0*
by (*induct m*) *simp-all*

lemma *mult-Suc-right* [*simp*]: *m * Suc n = m + (m * n)*
by (*induct m*) (*simp-all add*: *add-left-commute*)

lemma *add-mult-distrib*: *(m + n) * k = (m * k) + ((n * k)::nat)*
by (*induct m*) (*simp-all add*: *add-assoc*)

instance proof
fix *n m q* :: *nat*
show *0 ≠ (1::nat)* **by** *simp*
show *1 * n = n* **by** *simp*
show *n * m = m * n* **by** (*induct n*) *simp-all*
show *(n * m) * q = n * (m * q)* **by** (*induct n*) (*simp-all add*: *add-mult-distrib*)
show *(n + m) * q = n * q + m * q* **by** (*rule add-mult-distrib*)
assume *n + m = n + q* **thus** *m = q* **by** (*induct n*) *simp-all*
qed

end

14.3.1 Addition

lemma *nat-add-assoc*: *(m + n) + k = m + ((n + k)::nat)*
by (*rule add-assoc*)

lemma *nat-add-commute*: *m + n = n + (m::nat)*
by (*rule add-commute*)

lemma *nat-add-left-commute*: *x + (y + z) = y + ((x + z)::nat)*
by (*rule add-left-commute*)

lemma *nat-add-left-cancel* [*simp*]: *(k + m = k + n) = (m = (n::nat))*
by (*rule add-left-cancel*)

lemma *nat-add-right-cancel* [*simp*]: *(m + k = n + k) = (m=(n::nat))*
by (*rule add-right-cancel*)

Reasoning about *m + 0 = 0*, etc.

lemma *add-is-0* [*iff*]:

fixes *m n* :: *nat*
shows $(m + n = 0) = (m = 0 \ \& \ n = 0)$
by (*cases m*) *simp-all*

lemma *add-is-1*:
$(m+n= Suc\ 0) = (m= Suc\ 0 \ \& \ n=0 \mid m=0 \ \& \ n= Suc\ 0)$
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-eq-self-zero*:
fixes *m n* :: *nat*
shows $m + n = m \Longrightarrow n = 0$
by (*induct m*) *simp-all*

lemma *inj-on-add-nat*[*simp*]: *inj-on* (%*n*::*nat*. *n*+*k*) *N*
apply (*induct k*)
apply *simp*
apply(*drule comp-inj-on*[*OF - inj-Suc*])
apply (*simp add*:*o-def*)
done

14.3.2 Difference

lemma *diff-self-eq-0* [*simp*]: (*m*::*nat*) − *m* = *0*
by (*induct m*) *simp-all*

lemma *diff-diff-left*: (*i*::*nat*) − *j* − *k* = *i* − (*j* + *k*)
by (*induct i j rule*: *diff-induct*) *simp-all*

lemma *Suc-diff-diff* [*simp*]: (*Suc m* − *n*) − *Suc k* = *m* − *n* − *k*
by (*simp add*: *diff-diff-left*)

lemma *diff-commute*: (*i*::*nat*) − *j* − *k* = *i* − *k* − *j*
by (*simp add*: *diff-diff-left add-commute*)

lemma *diff-add-inverse*: (*n* + *m*) − *n* = (*m*::*nat*)
by (*induct n*) *simp-all*

lemma *diff-add-inverse2*: (*m* + *n*) − *n* = (*m*::*nat*)
by (*simp add*: *diff-add-inverse add-commute* [*of m n*])

lemma *diff-cancel*: (*k* + *m*) − (*k* + *n*) = *m* − (*n*::*nat*)
by (*induct k*) *simp-all*

lemma *diff-cancel2*: (*m* + *k*) − (*n* + *k*) = *m* − (*n*::*nat*)
by (*simp add*: *diff-cancel add-commute*)

lemma *diff-add-0*: $n - (n + m) = (0::nat)$
by (*induct n*) *simp-all*

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

14.3.3 Multiplication

lemma *nat-mult-assoc*: $(m * n) * k = m * ((n * k)::nat)$
by (*rule mult-assoc*)

lemma *nat-mult-commute*: $m * n = n * (m::nat)$
by (*rule mult-commute*)

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + ((k * n)::nat)$
by (*rule right-distrib*)

lemma *mult-is-0* [*simp*]: $((m::nat) * n = 0) = (m=0 \mid n=0)$
by (*induct m*) *auto*

lemmas *nat-distrib* =
add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

lemma *mult-eq-1-iff* [*simp*]: $(m * n = Suc\ 0) = (m = 1\ \&\ n = 1)$
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-cancel1* [*simp*]: $(k * m = k * n) = (m = n \mid (k = (0::nat)))$
proof −
have $k \neq 0 \Longrightarrow k * m = k * n \Longrightarrow m = n$
proof (*induct n arbitrary*: *m*)
case *0* **then show** $m = 0$ **by** *simp*
next
case (*Suc n*) **then show** $m = Suc\ n$
by (*cases m*) (*simp-all add*: *eq-commute* [*of 0*])

qed
then show *?thesis* **by** *auto*
qed

lemma *mult-cancel2* [*simp*]: $(m * k = n * k) = (m = n \mid (k = (0::nat)))$
by (*simp add*: *mult-commute*)

lemma *Suc-mult-cancel1*: $(Suc\ k * m = Suc\ k * n) = (m = n)$
by (*subst mult-cancel1*) *simp*

14.4 Orders on *nat*

14.4.1 Operation definition

instantiation *nat* :: *linorder*
begin

primrec *less-eq-nat* **where**
$(0::nat) \leq n \longleftrightarrow True$
| $Suc\ m \leq n \longleftrightarrow (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*, *code del*]
lemma [*code*]: $(0::nat) \leq n \longleftrightarrow True$ **by** (*simp add*: *less-eq-nat.simps*)
lemma *le0* [*iff*]: $0 \leq (n::nat)$ **by** (*simp add*: *less-eq-nat.simps*)

definition *less-nat* **where**
less-eq-Suc-le [*code func del*]: $n < m \longleftrightarrow Suc\ n \leq m$

lemma *Suc-le-mono* [*iff*]: $Suc\ n \leq Suc\ m \longleftrightarrow n \leq m$
by (*simp add*: *less-eq-nat.simps*(*2*))

lemma *Suc-le-eq* [*code*]: $Suc\ m \leq n \longleftrightarrow m < n$
unfolding *less-eq-Suc-le* **..**

lemma *le-0-eq* [*iff*]: $(n::nat) \leq 0 \longleftrightarrow n = 0$
by (*induct n*) (*simp-all add*: *less-eq-nat.simps*(*2*))

lemma *not-less0* [*iff*]: $\neg\ n < (0::nat)$
by (*simp add*: *less-eq-Suc-le*)

lemma *less-nat-zero-code* [*code*]: $n < (0::nat) \longleftrightarrow False$
by *simp*

lemma *Suc-less-eq* [*iff*]: $Suc\ m < Suc\ n \longleftrightarrow m < n$
by (*simp add*: *less-eq-Suc-le*)

lemma *less-Suc-eq-le* [*code*]: $m < Suc\ n \longleftrightarrow m \leq n$
by (*simp add*: *less-eq-Suc-le*)

lemma *le-SucI*: $m \leq n \Longrightarrow m \leq Suc\ n$

```
  by (induct m arbitrary: n)
    (simp-all add: less-eq-nat.simps(2) split: nat.splits)

lemma Suc-leD: Suc m ≤ n ⟹ m ≤ n
  by (cases n) (auto intro: le-SucI)

lemma less-SucI: m < n ⟹ m < Suc n
  by (simp add: less-eq-Suc-le) (erule Suc-leD)

lemma Suc-lessD: Suc m < n ⟹ m < n
  by (simp add: less-eq-Suc-le) (erule Suc-leD)

instance
proof
  fix n m :: nat
  have less-imp-le: n < m ⟹ n ≤ m
    unfolding less-eq-Suc-le by (erule Suc-leD)
  have irrefl: ¬ m < m by (induct m) auto
  have strict: n ≤ m ⟹ n ≠ m ⟹ n < m
  proof (induct n arbitrary: m)
    case 0 then show ?case
      by (cases m) (simp-all add: less-eq-Suc-le)
  next
    case (Suc n) then show ?case
      by (cases m) (simp-all add: less-eq-Suc-le)
  qed
  show n < m ⟷ n ≤ m ∧ n ≠ m
    by (auto simp add: irrefl intro: less-imp-le strict)
next
  fix n :: nat show n ≤ n by (induct n) simp-all
next
  fix n m :: nat assume n ≤ m and m ≤ n
  then show n = m
    by (induct n arbitrary: m)
      (simp-all add: less-eq-nat.simps(2) split: nat.splits)
next
  fix n m q :: nat assume n ≤ m and m ≤ q
  then show n ≤ q
  proof (induct n arbitrary: m q)
    case 0 show ?case by simp
  next
    case (Suc n) then show ?case
      by (simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
        simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
        simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits)
  qed
next
  fix n m :: nat show n ≤ m ∨ m ≤ n
    by (induct n arbitrary: m)
```

(*simp-all add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)
qed

end

14.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < Suc\ n$
by (*simp add*: *less-Suc-eq-le*)

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
by (*simp add*: *less-Suc-eq-le*)

14.4.3 Elimination properties

lemma *less-not-refl*: $\sim n < (n::nat)$
by (*rule order-less-irrefl*)

lemma *less-not-refl2*: $n < m \Longrightarrow m \neq (n::nat)$
by (*rule not-sym*) (*rule less-imp-neq*)

lemma *less-not-refl3*: $(s::nat) < t \Longrightarrow s \neq t$
by (*rule less-imp-neq*)

lemma *less-irrefl-nat*: $(n::nat) < n \Longrightarrow R$
by (*rule notE*, *rule less-not-refl*)

lemma *less-zeroE*: $(n::nat) < 0 \Longrightarrow R$
by (*rule notE*) (*rule not-less0*)

lemma *less-Suc-eq*: $(m < Suc\ n) = (m < n \mid m = n)$
unfolding *less-Suc-eq-le le-less* **..**

lemma *less-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*: $m < n \Longrightarrow Suc\ m < Suc\ n$
by *simp*

"Less than" is antisymmetric, sort of

lemma *less-antisym*: $[\![\neg n < m;\ n < Suc\ m]\!] \Longrightarrow m = n$
unfolding *not-less less-Suc-eq-le* **by** (*rule antisym*)

lemma *nat-neq-iff*: $((m::nat) \neq n) = (m < n \mid n < m)$
by (*rule linorder-neq-iff*)

lemma *nat-less-cases*: **assumes** *major*: $(m::nat) < n \Longrightarrow P\ n\ m$

and *eqCase*: $m = n$ ==> $P\ n\ m$ **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

14.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n$ ==> $Suc\ m \neq n$ ==> $Suc\ m < n$
unfolding *less-eq-Suc-le* [*of m*] *le-less* **by** *simp*

lemma *lessE*:
assumes *major*: $i < k$
and *p1*: $k = Suc\ i$ ==> P **and** *p2*: !!j. $i < j$ ==> $k = Suc\ j$ ==> P
shows P
proof −
from *major* **have** $\exists j.\ i \leq j \wedge k = Suc\ j$
unfolding *less-eq-Suc-le* **by** (*induct k*) *simp-all*
then have $(\exists j.\ i < j \wedge k = Suc\ j) \vee k = Suc\ i$
by (*clarsimp simp add*: *less-le*)
with *p1 p2* **show** P **by** *auto*
qed

lemma *less-SucE*: **assumes** *major*: $m < Suc\ n$
and *less*: $m < n$ ==> P **and** *eq*: $m = n$ ==> P **shows** P
apply (*rule major* [*THEN lessE*])
apply (*rule eq*, *blast*)
apply (*rule less*, *blast*)
done

lemma *Suc-lessE*: **assumes** *major*: $Suc\ i < k$
and *minor*: !!j. $i < j$ ==> $k = Suc\ j$ ==> 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 *simp*

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

Can be used with *less-Suc-eq* to get $n = m \vee n < m$

lemma *not-less-eq*: $\neg\ m < n \longleftrightarrow n < Suc\ m$
unfolding *not-less less-Suc-eq-le* **..**

lemma *not-less-eq-eq*: $\neg\ m \leq n \longleftrightarrow Suc\ n \leq m$
unfolding *not-le Suc-le-eq* **..**

Properties of "less than or equal"

lemma *le-imp-less-Suc*: $m \leq n \Longrightarrow m < Suc\ n$
unfolding *less-Suc-eq-le* **.**

lemma *Suc-n-not-le-n*: $\sim\ Suc\ n \leq n$
unfolding *not-le less-Suc-eq-le* **..**

lemma *le-Suc-eq*: $(m \leq Suc\ n) = (m \leq n \mid m = Suc\ n)$
by (*simp add*: *less-Suc-eq-le* [*symmetric*] *less-Suc-eq*)

lemma *le-SucE*: $m \leq Suc\ n \Longrightarrow (m \leq n \Longrightarrow R) \Longrightarrow (m = Suc\ n \Longrightarrow R) \Longrightarrow R$
by (*drule le-Suc-eq* [*THEN iffD1*], *iprover+*)

lemma *Suc-leI*: $m < n \Longrightarrow Suc(m) \leq n$
unfolding *Suc-le-eq* **.**

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: $Suc\ m \leq n \Longrightarrow m < n$
unfolding *Suc-le-eq* **.**

lemma *less-imp-le-nat*: $m < n \Longrightarrow m \leq (n::nat)$
unfolding *less-eq-Suc-le* **by** (*rule Suc-leD*)

For instance, $(Suc\ m < Suc\ n) = (Suc\ m \leq n) = (m < n)$

lemmas *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *less-or-eq-imp-le*: $m < n \mid m = n \Longrightarrow m \leq (n::nat)$
unfolding *le-less* **.**

lemma *le-eq-less-or-eq*: $(m \leq (n::nat)) = (m < n \mid m=n)$
by (*rule le-less*)

Useful with *blast*.

lemma *eq-imp-le*: $(m::nat) = n \Longrightarrow m \leq n$
by *auto*

lemma *le-refl*: $n \leq (n::nat)$

by *simp*

lemma *le-trans*: [| $i \leq j$; $j \leq k$ |] ==> $i \leq (k::nat)$
by (*rule order-trans*)

lemma *le-anti-sym*: [| $m \leq n$; $n \leq m$ |] ==> $m = (n::nat)$
by (*rule antisym*)

lemma *nat-less-le*: $((m::nat) < n) = (m \leq n \;\&\; m \neq n)$
by (*rule less-le*)

lemma *le-neq-implies-less*: $(m::nat) \leq n$ ==> $m \neq n$ ==> $m < n$
unfolding *less-le* **..**

lemma *nat-le-linear*: $(m::nat) \leq n \mid n \leq m$
by (*rule linear*)

lemmas *linorder-neqE-nat* = *linorder-neqE* [**where** $'a = nat$]

lemma *le-less-Suc-eq*: $m \leq n$ ==> $(n < Suc\ m) = (n = m)$
unfolding *less-Suc-eq-le* **by** *auto*

lemma *not-less-less-Suc-eq*: ~ $n < m$ ==> $(n < Suc\ m) = (n = m)$
unfolding *not-less* **by** (*rule le-less-Suc-eq*)

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

These two rules ease the use of primitive recursion. NOTE USE OF ==

lemma *def-nat-rec-0*: (!!n. $f\ n == nat\text{-}rec\ c\ h\ n$) ==> $f\ 0 = c$
by *simp*

lemma *def-nat-rec-Suc*: (!!n. $f\ n == nat\text{-}rec\ c\ h\ n$) ==> $f\ (Suc\ n) = h\ n\ (f\ n)$
by *simp*

lemma *not0-implies-Suc*: $n \neq 0$ ==> $\exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr0-implies-Suc*: $n > 0$ ==> $\exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr-implies-not0*: **fixes** n :: *nat* **shows** $m<n$ ==> $n \neq 0$
by (*cases n*) *simp-all*

lemma *neq0-conv*[*iff*]: **fixes** n :: *nat* **shows** $(n \neq 0) = (0 < n)$
by (*cases n*) *simp-all*

This theorem is useful with *blast*

lemma *gr0I*: $((n::nat) = 0$ ==> *False*$)$ ==> $0 < n$
by (*rule neq0-conv*[*THEN iffD1*], *iprover*)

lemma *gr0-conv-Suc*: $(0 < n) = (\exists m.\ n = Suc\ m)$
by (*fast intro*: *not0-implies-Suc*)

lemma *not-gr0* [*iff*,*noatp*]: $!!n{::}nat.\ (\sim (0 < n)) = (n = 0)$
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*

14.4.5 *min* and *max*

lemma *mono-Suc*: *mono Suc*
by (*rule monoI*) *simp*

lemma *min-0L* [*simp*]: $min\ 0\ n = (0{::}nat)$
by (*rule min-leastL*) *simp*

lemma *min-0R* [*simp*]: $min\ n\ 0 = (0{::}nat)$
by (*rule min-leastR*) *simp*

lemma *min-Suc-Suc* [*simp*]: $min\ (Suc\ m)\ (Suc\ n) = Suc\ (min\ m\ n)$
by (*simp add*: *mono-Suc min-of-mono*)

lemma *min-Suc1*:
$min\ (Suc\ n)\ m = (case\ m\ of\ 0 => 0 \mid Suc\ m' => Suc(min\ n\ m'))$
by (*simp split*: *nat.split*)

lemma *min-Suc2*:
$min\ m\ (Suc\ n) = (case\ m\ of\ 0 => 0 \mid Suc\ m' => Suc(min\ m'\ n))$
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*:
$max\ (Suc\ n)\ m = (case\ m\ of\ 0 => Suc\ n \mid Suc\ m' => Suc(max\ n\ m'))$
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*)

14.4.6 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: *n>0 ==> Suc (n - Suc 0) = n*
by (*simp add*: *diff-Suc split*: *nat.split*)

lemma *nat-add-left-cancel-le* [*simp*]: *(k + m ≤ k + n) = (m≤(n::nat))*
by (*induct k*) *simp-all*

lemma *nat-add-left-cancel-less* [*simp*]: *(k + m < k + n) = (m<(n::nat))*
by (*induct k*) *simp-all*

lemma *add-gr-0* [*iff*]: *!!m::nat. (m + n > 0) = (m>0 | n>0)*
by(*auto dest*:*gr0-implies-Suc*)

strict, in 1st argument

lemma *add-less-mono1*: *i < j ==> i + k < j + (k::nat)*
by (*induct k*) *simp-all*

strict, in both arguments

lemma *add-less-mono*: *[|i < j; k < l|] ==> i + k < j + (l::nat)*
  apply (*rule add-less-mono1* [*THEN less-trans*], *assumption+*)
  apply (*induct j*, *simp-all*)
  done

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

lemma *less-imp-Suc-add*: *m < n ==> (∃k. n = Suc (m + k))*
  apply (*induct n*)
  apply (*simp-all add*: *order-le-less*)
  apply (*blast elim*!: *less-SucE*
          intro!: *add-0-right* [*symmetric*] *add-Suc-right* [*symmetric*])
  done

strict, in 1st argument; proof is by induction on *k > 0*

lemma *mult-less-mono2*: *(i::nat) < j ==> 0<k ==> k * i < k * j*
apply(*auto simp*: *gr0-conv-Suc*)
apply (*induct-tac m*)
apply (*simp-all add*: *add-less-mono*)
done

The naturals form an ordered *comm-semiring-1-cancel*

instance *nat* :: *ordered-semidom*
proof
  fix *i j k* :: *nat*
  show *0 < (1::nat)* **by** *simp*

show $i \leq j$ ==> $k + i \leq k + j$ **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*

14.4.7 Additional theorems about *op* ≤

Complete induction, aka course-of-values induction

lemma *less-induct* [*case-names less*]:
fixes $P :: nat \Rightarrow bool$
assumes *step*: $\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows $P\ a$
proof −
have $\bigwedge z.\ z \leq a \Longrightarrow P\ z$
proof (*induct a*)
case (*0 z*)
have *P 0* **by** (*rule step*) *auto*
thus *?case* **using** *0* **by** *auto*
next
case (*Suc x z*)
then have $z \leq x \vee z = Suc\ x$ **by** (*simp add*: *le-Suc-eq*)
thus *?case*
proof
assume $z \leq x$ **thus** $P\ z$ **by** (*rule Suc*(*1*))
next
assume *z*: $z = Suc\ x$
show $P\ z$
by (*rule step*) (*rule Suc*(*1*), *simp add*: *z le-simps*)
qed
qed
thus *?thesis* **by** *auto*
qed

lemma *nat-less-induct*:
assumes !!*n*. ∀ *m*::*nat*. $m < n$ −−> $P\ m$ ==> $P\ n$ **shows** $P\ n$
using *assms less-induct* **by** *blast*

lemma *measure-induct-rule* [*case-names less*]:
fixes $f :: 'a \Rightarrow nat$
assumes *step*: $\bigwedge x.\ (\bigwedge y.\ f\ y < f\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows $P\ a$
by (*induct m*≡*f a arbitrary*: *a rule*: *less-induct*) (*auto intro*: *step*)

old style induction rules:

lemma *measure-induct*:
fixes $f :: 'a \Rightarrow nat$
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 *full-nat-induct*:
assumes *step*: (!!*n*. (*ALL m*. *Suc m* <= *n* --> *P m*) ==> *P n*)
shows *P n*
by (*rule less-induct*) (*auto intro*: *step simp*:*le-simps*)

An induction rule for estabilishing binary relations

lemma *less-Suc-induct*:
assumes *less*: $i < j$
and *step*: !!*i*. *P i* (*Suc i*)
and *trans*: !!*i j k*. *P i j* ==> *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

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

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. $P(n)$ is true for all $n \in \mathbb{N}$ if

- case "0": given $n = 0$ prove $P(n)$,
- case "smaller": given $n > 0$ and $\neg P(n)$ prove there exists a smaller integer m such that $\neg P(m)$.

A compact version without explicit base case:

lemma *infinite-descent*:
⟦ !!*n*::*nat*. ¬ *P n* ⟹ ∃ *m*<*n*. ¬ *P m* ⟧ ⟹ *P n*
by (*induct n rule*: *less-induct*, *auto*)

lemma *infinite-descent0*[*case-names 0 smaller*]:
⟦ *P 0*; !!*n*. *n*>*0* ⟹ ¬ *P n* ⟹ (∃ *m*::*nat*. *m* < *n* ∧ ¬*P m*) ⟧ ⟹ *P n*
by (*rule infinite-descent*) (*case-tac n*>*0*, *auto*)

Infinite descent using a mapping to $\mathbb{N}$: $P(x)$ is true for all $x \in D$ if there exists a $V : D \to \mathbb{N}$ and

- case "0": given $V(x) = 0$ prove $P(x)$,
- case "smaller": given $V(x) > 0$ and $\neg P(x)$ prove there exists a $y \in D$ such that $V(y) < V(x)$ and $\neg P(y)$.

NB: the proof also shows how to use the previous lemma.

corollary *infinite-descent0-measure* [*case-names 0 smaller*]:
assumes *A0*: !!*x*. *V x* = (*0*::*nat*) ⟹ *P x*
and *A1*: !!*x*. *V x* > *0* ⟹ ¬*P x* ⟹ (∃ *y*. *V y* < *V x* ∧ ¬*P y*)
shows *P x*
proof −
obtain *n* **where** *n* = *V x* **by** *auto*
moreover have ⋀*x*. *V x* = *n* ⟹ *P x*
proof (*induct n rule*: *infinite-descent0*)
case *0* — i.e. *V*(*x*) = 0
with *A0* **show** *P x* **by** *auto*
next — now *n* > 0 and *P*(*x*) does not hold for some *x* with *V*(*x*) = *n*
case (*smaller n*)
then obtain *x* **where** *vxn*: *V x* = *n* **and** *V x* > *0* ∧ ¬ *P x* **by** *auto*
with *A1* **obtain** *y* **where** *V y* < *V x* ∧ ¬ *P y* **by** *auto*
with *vxn* **obtain** *m* **where** *m* = *V y* ∧ *m*<*n* ∧ ¬ *P y* **by** *auto*
then show *?case* **by** *auto*
qed
ultimately show *P x* **by** *auto*
qed

Again, without explicit base case:

lemma *infinite-descent-measure*:
assumes !!*x*. ¬ *P x* ⟹ ∃ *y*. (*V*::*'a*⇒*nat*) *y* < *V x* ∧ ¬ *P y* **shows** *P x*
proof −
from *assms* **obtain** *n* **where** *n* = *V x* **by** *auto*
moreover have !!*x*. *V x* = *n* ⟹ *P x*
proof (*induct n rule*: *infinite-descent*, *auto*)
fix *x* **assume** ¬ *P x*
with *assms* **show** ∃ *m* < *V x*. ∃ *y*. *V y* = *m* ∧ ¬ *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*. $i < j \Longrightarrow f\ i < f\ j$; $i \leq j$ ⟧ $\Longrightarrow f\ i \leq ((f\ j)$::*nat*)
by (*simp add*: *order-le-less*) (*blast*)

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j$ ==> $i + k \leq j + (k$::*nat*)
by (*rule add-right-mono*)

non-strict, in both arguments

lemma *add-le-mono*: [| $i \leq j$; $k \leq l$ |] ==> $i + k \leq j + (l$::*nat*)
by (*rule add-mono*)

lemma *le-add2*: $n \leq ((m + n)$::*nat*)
by (*insert add-right-mono* [*of 0 m n*], *simp*)

lemma *le-add1*: $n \leq ((n + m)$::*nat*)
by (*simp add*: *add-commute*, *rule le-add2*)

lemma *less-add-Suc1*: $i < Suc\ (i + m)$
by (*rule le-less-trans*, *rule le-add1*, *rule lessI*)

lemma *less-add-Suc2*: $i < Suc\ (m + i)$
by (*rule le-less-trans*, *rule le-add2*, *rule lessI*)

lemma *less-iff-Suc-add*: $(m < n) = (\exists k.\ n = Suc\ (m + k))$
by (*iprover intro*!: *less-add-Suc1 less-imp-Suc-add*)

lemma *trans-le-add1*: $(i$::*nat*$) \leq j$ ==> $i \leq j + m$
by (*rule le-trans*, *assumption*, *rule le-add1*)

lemma *trans-le-add2*: $(i$::*nat*$) \leq j$ ==> $i \leq m + j$
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 (*drule add-lessD1*)
apply (*erule less-irrefl* [*THEN notE*])
done

lemma *not-add-less2* [*iff*]: ~ $(j + i < (i::nat))$
by (*simp add*: *add-commute*)

lemma *add-leD1*: $m + k \leq n \Longrightarrow m \leq (n::nat)$
apply (*rule order-trans* [*of* - *m+k*])
apply (*simp-all add*: *le-add1*)
done

lemma *add-leD2*: $m + k \leq n \Longrightarrow k \leq (n::nat)$
apply (*simp add*: *add-commute*)
apply (*erule add-leD1*)
done

lemma *add-leE*: $(m::nat) + k \leq n \Longrightarrow (m \leq n \Longrightarrow k \leq n \Longrightarrow R) \Longrightarrow R$
by (*blast dest*: *add-leD1 add-leD2*)

needs $!!k$ for *add-ac* to work

lemma *less-add-eq-less*: $!!k::nat.\ k < l \Longrightarrow m + l = k + n \Longrightarrow m < n$
by (*force simp del*: *add-Suc-right*
simp add: *less-iff-Suc-add add-Suc-right* [*symmetric*] *add-ac*)

14.4.8 More results about difference

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add-diff-inverse*: ~ $m < n \Longrightarrow n + (m - n) = (m::nat)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *le-add-diff-inverse* [*simp*]: $n \leq m \Longrightarrow n + (m - n) = (m::nat)$
by (*simp add*: *add-diff-inverse linorder-not-less*)

lemma *le-add-diff-inverse2* [*simp*]: $n \leq m \Longrightarrow (m - n) + n = (m::nat)$
by (*simp add*: *add-commute*)

lemma *Suc-diff-le*: $n \leq m \Longrightarrow Suc\ m - n = Suc\ (m - n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-less-Suc*: $m - n < Suc\ m$
apply (*induct m n rule*: *diff-induct*)
apply (*erule-tac* [*3*] *less-SucE*)
apply (*simp-all add*: *less-Suc-eq*)
done

lemma *diff-le-self* [*simp*]: $m - n \leq (m::nat)$
by (*induct m n rule*: *diff-induct*) (*simp-all add*: *le-SucI*)

lemma *le-iff-add*: $(m::nat) \leq n = (\exists k.\ n = m + k)$
by (*auto simp*: *le-add1 dest*!: *le-add-diff-inverse sym* [*of* - *n*])

lemma *less-imp-diff-less*: $(j::nat) < k \Longrightarrow j - n < k$

by (*rule le-less-trans*, *rule diff-le-self*)

lemma *diff-Suc-less* [*simp*]: *0<n ==> n - Suc i < n*
by (*cases n*) (*auto simp add*: *le-simps*)

lemma *diff-add-assoc*: *k ≤ (j::nat) ==> (i + j) - k = i + (j - k)*
by (*induct j k rule*: *diff-induct*) *simp-all*

lemma *diff-add-assoc2*: *k ≤ (j::nat) ==> (j + i) - k = (j - k) + i*
by (*simp add*: *add-commute diff-add-assoc*)

lemma *le-imp-diff-is-add*: *i ≤ (j::nat) ==> (j - i = k) = (j = k + i)*
by (*auto simp add*: *diff-add-inverse2*)

lemma *diff-is-0-eq* [*simp*]: *((m::nat) - n = 0) = (m ≤ n)*
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-is-0-eq'* [*simp*]: *m ≤ n ==> (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 *∃k::nat. 0 < k & i + k = j*
proof
from *assms* **show** *0 < j - i & i + (j - i) = j*
by (*simp add*: *order-less-imp-le*)
qed

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*:
fixes *n m* :: *nat*
shows *n - m + m = max n m*
by (*simp add*: *max-def not-le order-less-imp-le*)

lemma *nat-diff-split*:
P(a - b::nat) = ((a<b --> P 0) & (ALL d. a = b + d --> P d))
— elimination of *-* on *nat*
by (*cases a < b*)
(*auto simp add*: *diff-is-0-eq* [*THEN iffD2*] *diff-add-inverse*
not-less le-less dest!: *sym* [*of a*] *sym* [*of b*] *add-eq-self-zero*)

lemma *nat-diff-split-asm*:
P(a - b::nat) = (~ (a < b & ~ P 0 | (EX d. a = b + d & ~ P d)))
— elimination of *-* on *nat* in assumptions
by (*auto split*: *nat-diff-split*)

14.4.9 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \leq (j::nat) \Longrightarrow i * k \leq j * k$
by (*simp add*: *mult-right-mono*)

lemma *mult-le-mono2*: $i \leq (j::nat) \Longrightarrow k * i \leq k * j$
by (*simp add*: *mult-left-mono*)

$\leq$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq (j::nat) \Longrightarrow k \leq l \Longrightarrow i * k \leq j * l$
by (*simp add*: *mult-mono*)

lemma *mult-less-mono1*: $(i::nat) < j \Longrightarrow 0 < k \Longrightarrow i * k < j * k$
by (*simp add*: *mult-strict-right-mono*)

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $(0 < (m::nat) * n) = (0 < m \ \& \ 0 < n)$
apply (*induct m*)
apply *simp*
apply (*case-tac n*)
apply *simp-all*
done

lemma *one-le-mult-iff* [*simp*]: $(Suc\ 0 \leq m * n) = (1 \leq m \ \& \ 1 \leq n)$
apply (*induct m*)
apply *simp*
apply (*case-tac n*)
apply *simp-all*
done

lemma *mult-less-cancel2* [*simp*]: $((m::nat) * k < n * k) = (0 < k \ \& \ m < n)$
apply (*safe intro*!: *mult-less-mono1*)
apply (*case-tac k*, *auto*)
apply (*simp del*: *le-0-eq add*: *linorder-not-le* [*symmetric*])
apply (*blast intro*: *mult-le-mono1*)
done

lemma *mult-less-cancel1* [*simp*]: $(k * (m::nat) < k * n) = (0 < k \ \& \ m < n)$
by (*simp add*: *mult-commute* [*of k*])

lemma *mult-le-cancel1* [*simp*]: $(k * (m::nat) \leq k * n) = (0 < k \longrightarrow m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *mult-le-cancel2* [*simp*]: $((m::nat) * k \leq n * k) = (0 < k \longrightarrow m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *Suc-mult-less-cancel1*: $(Suc\ k * m < Suc\ k * n) = (m < n)$
by (*subst mult-less-cancel1*) *simp*

lemma *Suc-mult-le-cancel1*: $(Suc\ k * m \leq Suc\ k * n) = (m \leq n)$
by (*subst mult-le-cancel1*) *simp*

lemma *le-square*: $m \leq m * (m::nat)$
by (*cases m*) (*auto intro*: *le-add1*)

lemma *le-cube*: $(m::nat) \leq m * (m * m)$
by (*cases m*) (*auto intro*: *le-add1*)

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $(m::nat) = m * n \Longrightarrow n = 1 \mid m = 0$
apply (*drule sym*)
apply (*rule disjCI*)
apply (*rule nat-less-cases*, *erule-tac* [*2*] -)
apply (*drule-tac* [*2*] *mult-less-mono2*)
apply (*auto*)
done

the lattice order on *nat*

instantiation *nat* :: *distrib-lattice*
begin

definition
$(inf :: nat \Rightarrow nat \Rightarrow nat) = min$

definition
$(sup :: nat \Rightarrow nat \Rightarrow nat) = max$

instance by *intro-classes*
(*auto simp add*: *inf-nat-def sup-nat-def max-def not-le min-def*
intro: *order-less-imp-le antisym elim*!: *order-trans order-less-trans*)

end

14.5 Embedding of the Naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

primrec
$of\text{-}nat :: nat \Rightarrow 'a$
where
of-nat-0: $of\text{-}nat\ 0 = 0$
| *of-nat-Suc*: $of\text{-}nat\ (Suc\ m) = 1 + of\text{-}nat\ m$

lemma *of-nat-1* [*simp*]: $of\text{-}nat\ 1 = 1$
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*)

definition
of-nat-aux :: *nat* ⇒ *'a* ⇒ *'a*
where
[*code func del*]: *of-nat-aux n i* = *of-nat n* + *i*

lemma *of-nat-aux-code* [*code*]:
of-nat-aux 0 i = *i*
of-nat-aux (*Suc n*) *i* = *of-nat-aux n* (*i* + *1*) — tail recursive
by (*simp-all add*: *of-nat-aux-def add-ac*)

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

end

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *semiring-char-0* = *semiring-1* +
assumes *of-nat-eq-iff* [*simp*]: *of-nat m* = *of-nat n* ⟷ *m* = *n*
begin

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*, *noatp*]: *0* = *of-nat n* ⟷ *0* = *n*
by (*rule of-nat-eq-iff* [*of 0*, *simplified*])

lemma *of-nat-eq-0-iff* [*simp*, *noatp*]: *of-nat m* = *0* ⟷ *m* = *0*
by (*rule of-nat-eq-iff* [*of - 0*, *simplified*])

lemma *inj-of-nat*: *inj of-nat*
by (*simp add*: *inj-on-def*)

end

context *ordered-semidom*
begin

lemma *zero-le-imp-of-nat*: *0* ≤ *of-nat m*
apply (*induct m*, *simp-all*)
apply (*erule order-trans*)
apply (*rule ord-le-eq-trans* [*OF - add-commute*])
apply (*rule less-add-one* [*THEN less-imp-le*])
done

lemma *less-imp-of-nat-less*: $m < n \Longrightarrow$ *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* $n \Longrightarrow m < n$
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* $n \longleftrightarrow m < n$
by (*blast intro*: *of-nat-less-imp-less less-imp-of-nat-less*)

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*])

Every *ordered-semidom* has characteristic zero.

subclass *semiring-char-0*
by *unfold-locales* (*simp add*: *eq-iff order-eq-iff*)

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*])

lemma *of-nat-0-less-iff* [*simp*]: $0 <$ *of-nat* $n \longleftrightarrow 0 < n$
by (*rule of-nat-less-iff* [*of 0*, *simplified*])

lemma *of-nat-less-0-iff* [*simp*]: $\neg$ *of-nat* $m < 0$
by (*rule of-nat-less-iff* [*of - 0*, *simplified*])

end

context *ring-1*
begin

lemma *of-nat-diff*: $n \leq m \Longrightarrow$ *of-nat* $(m - n) =$ *of-nat* $m -$ *of-nat* n
by (*simp add*: *compare-rls of-nat-add* [*symmetric*])

end

context *ordered-idom*
begin

lemma *abs-of-nat* [*simp*]: $|$*of-nat* $n| =$ *of-nat* n

unfolding *abs-if* **by** *auto*

end

lemma *of-nat-id* [*simp*]: *of-nat n = n*
by (*induct n*) *auto*

lemma *of-nat-eq-id* [*simp*]: *of-nat = id*
by (*auto simp add*: *expand-fun-eq*)

14.6 The Set of Natural Numbers

context *semiring-1*
begin

definition
Nats :: *'a set* **where**
Nats = range of-nat

notation (*xsymbols*)
Nats (ℕ)

lemma *of-nat-in-Nats* [*simp*]: *of-nat n* ∈ ℕ
by (*simp add*: *Nats-def*)

lemma *Nats-0* [*simp*]: *0* ∈ ℕ
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-0* [*symmetric*])
done

lemma *Nats-1* [*simp*]: *1* ∈ ℕ
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-1* [*symmetric*])
done

lemma *Nats-add* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a + b* ∈ ℕ
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-add* [*symmetric*])
done

lemma *Nats-mult* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a * b* ∈ ℕ
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-mult* [*symmetric*])
done

end

14.7 Further Arithmetic Facts Concerning the Natural Numbers

lemma *subst-equals*:
assumes *1*: $t = s$ **and** *2*: $u = t$
shows $u = s$
using *2 1* **by** (*rule trans*)

use *arith-data.ML*
declaration ⟪ *K ArithData.setup* ⟫

use *Tools/lin-arith.ML*
declaration ⟪ *K LinArith.setup* ⟫

lemmas [*arith-split*] = *nat-diff-split split-min split-max*

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*

Simplification of relational expressions involving subtraction

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*)

lemma *min-diff*: *min* $(m - (i::nat))$ $(n - i) = min\ m\ n - i$
unfolding *min-def* **by** *auto*

lemma *inj-on-diff-nat*:
  assumes *k-le-n*: $\forall n \in N.\ k \leq (n::nat)$
  shows *inj-on* $(\lambda n.\ n - k)$ N
proof (*rule inj-onI*)
  fix $x\ y$
  assume *a*: $x \in N\ y \in N\ x - k = y - k$
  with *k-le-n* **have** $x - k + k = y - k + k$ **by** *auto*
  with *a k-le-n* **show** $x = y$ **by** *auto*
qed

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j$ --> $i - (j - k) = i + (k::nat) - j$
by *arith*

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j$ ==> $m - Suc\ (j - k) = m + k - Suc\ j$
by *arith*

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j$ ==> $Suc\ (j - k) - m = Suc\ j - (k + m)$
by *arith*

Lemmas 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*

lemma *nat-not-singleton*: (∀ *x*. *x* = (*0*::*nat*)) = *False*
by *auto*

lemmas *add-diff-assoc* = *diff-add-assoc* [*symmetric*]
lemmas *add-diff-assoc2* = *diff-add-assoc2*[*symmetric*]

declare *diff-diff-left* [*simp*] *add-diff-assoc* [*simp*] *add-diff-assoc2*[*simp*]

At present we prove no analogue of *not-less-Least* or *Least-Suc*, since there appears to be no need.

14.8 size of a datatype value

class *size* = *type* +
  fixes *size* :: *'a* ⇒ *nat* — see further theory *Wellfounded*

end

15 Power: Exponentiation

theory *Power*
imports *Nat*
begin

class *power* = *type* +
  fixes *power* :: *'a* ⇒ *nat* ⇒ *'a* (**infixr** ^ *80*)

15.1 Powers for Arbitrary Monoids

class *recpower* = *monoid-mult* + *power* +
  assumes *power-0* [*simp*]: *a ^ 0* = *1*
  assumes *power-Suc*: *a ^ Suc n* = *a * (a ^ n)*

lemma *power-0-Suc* [*simp*]: *(0::'a::{recpower,semiring-0}) ^ (Suc n)* = *0*
  by (*simp add*: *power-Suc*)

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*: *0^n* = (*if n=0 then 1 else (0::'a::{recpower,semiring-0})*)
  by (*induct n*) *simp-all*

lemma *power-one* [*simp*]: *1^n* = (*1::'a::recpower*)
  by (*induct n*) (*simp-all add*: *power-Suc*)

lemma *power-one-right* [*simp*]: (*a::'a::recpower*) ^ *1* = *a*
  by (*simp add*: *power-Suc*)

lemma *power-commutes*: (*a::'a::recpower*) ^ *n * a* = *a * a ^ n*
  by (*induct n*) (*simp-all add*: *power-Suc mult-assoc*)

lemma *power-add*: (*a::'a::recpower*) ^ (*m+n*) = (*a^m*) * (*a^n*)
  by (*induct m*) (*simp-all add*: *power-Suc mult-ac*)

lemma *power-mult*: (*a::'a::recpower*) ^ (*m*n*) = (*a^m*) ^ *n*

by (*induct n*) (*simp-all add*: *power-Suc power-add*)

lemma *power-mult-distrib*: ((*a*::'*a*::{*recpower*,*comm-monoid-mult*}) * *b*) ^ *n* = (*a* ^ *n*) * (*b* ^ *n*)
by (*induct n*) (*simp-all add*: *power-Suc mult-ac*)

lemma *zero-less-power*[*simp*]:
0 < (*a*::'*a*::{*ordered-semidom*,*recpower*}) ==> *0* < *a* ^ *n*
apply (*induct n*)
apply (*simp-all add*: *power-Suc zero-less-one mult-pos-pos*)
done

lemma *zero-le-power*[*simp*]:
0 ≤ (*a*::'*a*::{*ordered-semidom*,*recpower*}) ==> *0* ≤ *a* ^ *n*
apply (*simp add*: *order-le-less*)
apply (*erule disjE*)
apply (*simp-all add*: *zero-less-one power-0-left*)
done

lemma *one-le-power*[*simp*]:
1 ≤ (*a*::'*a*::{*ordered-semidom*,*recpower*}) ==> *1* ≤ *a* ^ *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* ≤ (*a*::'*a*::*ordered-semidom*)
by (*simp add*: *order-trans* [*OF zero-le-one order-less-imp-le*])

lemma *power-gt1-lemma*:
assumes *gt1*: *1* < (*a*::'*a*::{*ordered-semidom*,*recpower*})
shows *1* < *a* * *a* ^ *n*
proof −
have *1*1* < *a*1* **using** *gt1* **by** *simp*
also have ... ≤ *a* * *a* ^ *n* **using** *gt1*
by (*simp only*: *mult-mono gt1-imp-ge0 one-le-power order-less-imp-le*
zero-le-one order-refl)
finally show *?thesis* **by** *simp*
qed

lemma *one-less-power*[*simp*]:
⟦*1* < (*a*::'*a*::{*ordered-semidom*,*recpower*}); *0* < *n*⟧ ⟹ *1* < *a* ^ *n*
by (*cases n*, *simp-all add*: *power-gt1-lemma power-Suc*)

lemma *power-gt1*:
1 < (*a*::'*a*::{*ordered-semidom*,*recpower*}) ==> *1* < *a* ^ (*Suc n*)
by (*simp add*: *power-gt1-lemma power-Suc*)

lemma *power-le-imp-le-exp*:
  assumes *gt1*: (*1*::'*a*::{*recpower*,*ordered-semidom*}) < *a*
  shows !!*n*. *a*^*m* ≤ *a*^*n* ==> *m* ≤ *n*
proof (*induct m*)
  case *0*
  show *?case* **by** *simp*
next
  case (*Suc m*)
  show *?case*
  proof (*cases n*)
    case *0*
    from *prems* **have** *a* * *a*^*m* ≤ *1* **by** (*simp add*: *power-Suc*)
    with *gt1* **show** *?thesis*
      by (*force simp only*: *power-gt1-lemma*
          linorder-not-less [*symmetric*])
  next
    case (*Suc n*)
    from *prems* **show** *?thesis*
      by (*force dest*: *mult-left-le-imp-le*
          simp add: *power-Suc order-less-trans* [*OF zero-less-one gt1*])
  qed
qed

Surely we can strengthen this? It holds for *0*<*a*<*1* too.

lemma *power-inject-exp* [*simp*]:
    1 < (*a*::'*a*::{*ordered-semidom*,*recpower*}) ==> (*a*^*m* = *a*^*n*) = (*m*=*n*)
  by (*force simp add*: *order-antisym power-le-imp-le-exp*)

Can relax the first premise to (*0*::'*a*) < *a* in the case of the natural numbers.

lemma *power-less-imp-less-exp*:
    [| (*1*::'*a*::{*recpower*,*ordered-semidom*}) < *a*; *a*^*m* < *a*^*n* |] ==> *m* < *n*
by (*simp add*: *order-less-le* [*of m n*] *order-less-le* [*of a*^*m a*^*n*]
          power-le-imp-le-exp)

lemma *power-mono*:
    [|*a* ≤ *b*; (*0*::'*a*::{*recpower*,*ordered-semidom*}) ≤ *a*|] ==> *a*^*n* ≤ *b*^*n*
apply (*induct n*)
apply (*simp-all add*: *power-Suc*)
apply (*auto intro*: *mult-mono 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 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 ≠ (0::'a::{ring-1-no-zero-divisors,recpower}) ==> a^n ≠ 0
by *force*

lemma *nonzero-power-inverse*:
fixes *a* :: *'a::{division-ring,recpower}*
shows *a ≠ 0 ==> inverse (a ^ n) = (inverse a) ^ 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-ring,division-by-zero,recpower}*
shows *inverse (a ^ n) = (inverse a) ^ n*
apply (*cases a = 0*)
apply (*simp add*: *power-0-left*)
apply (*simp add*: *nonzero-power-inverse*)
done

lemma *power-one-over*: *1 / (a::'a::{field,division-by-zero,recpower})^n =*
(1 / a)^n
apply (*simp add*: *divide-inverse*)
apply (*rule power-inverse*)
done

lemma *nonzero-power-divide*:
b ≠ 0 ==> (a/b) ^ n = ((a::'a::{field,recpower}) ^ n) / (b ^ n)
by (*simp add*: *divide-inverse power-mult-distrib nonzero-power-inverse*)

lemma *power-divide*:
(a/b) ^ n = ((a::'a::{field,division-by-zero,recpower}) ^ n / b ^ n)
apply (*case-tac b=0*, *simp add*: *power-0-left*)
apply (*rule nonzero-power-divide*)
apply *assumption*
done

lemma *power-abs*: *abs(a ^ n) = abs(a::'a::{ordered-idom,recpower}) ^ n*
apply (*induct n*)
apply (*auto simp add*: *power-Suc abs-mult*)
done

lemma *zero-less-power-abs-iff* [*simp*,*noatp*]:
(*0* < (*abs a*) ^ *n*) = (*a* ≠ (*0*::'*a*::{*ordered-idom*,*recpower*}) | *n*=*0*)
proof (*induct n*)
case *0*
show *?case* **by** (*simp 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-idom*,*recpower*}) ≤ (*abs a*) ^ *n*
by (*rule zero-le-power* [*OF abs-ge-zero*])

lemma *power-minus*: (−*a*) ^ *n* = (− *1*) ^ *n* * (*a*::'*a*::{*comm-ring-1*,*recpower*}) ^ *n*
proof −
have −*a* = (− *1*) * *a* **by** (*simp add*: *minus-mult-left* [*symmetric*])
thus *?thesis* **by** (*simp only*: *power-mult-distrib*)
qed

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*:
[|(*0*::'*a*::{*ordered-semidom*,*recpower*}) < *a*; *a* < *1*|]
==> *a* * *a* ^ *n* < *a* ^ *n*
apply (*induct n*)
apply (*auto simp add*: *power-Suc mult-strict-left-mono*)
done

lemma *power-strict-decreasing*:
[|*n* < *N*; *0* < *a*; *a* < (*1*::'*a*::{*ordered-semidom*,*recpower*})|]
==> *a* ^ *N* < *a* ^ *n*
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-Suc power-Suc-less less-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac a* * *a* ^ *m* < *1* * *a* ^ *n*, *simp*)
apply (*rule mult-strict-mono*)
apply (*auto simp add*: *zero-less-one order-less-imp-le*)
done

Proof resembles that of *power-strict-decreasing*

lemma *power-decreasing*:
[|*n* ≤ *N*; *0* ≤ *a*; *a* ≤ (*1*::'*a*::{*ordered-semidom*,*recpower*})|]
==> *a* ^ *N* ≤ *a* ^ *n*
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-Suc le-Suc-eq*)
apply (*rename-tac m*)

apply (*subgoal-tac a * a^m ≤ 1 * a^n, simp*)
apply (*rule mult-mono*)
apply (*auto simp add*: *zero-le-one*)
done

lemma *power-Suc-less-one*:
[| *0 < a; a < (1::'a::{ordered-semidom,recpower})* |] ==> *a ^ Suc n < 1*
apply (*insert power-strict-decreasing* [*of 0 Suc n a*], *simp*)
done

Proof again resembles that of *power-strict-decreasing*

lemma *power-increasing*:
[|*n ≤ N; (1::'a::{ordered-semidom,recpower}) ≤ a*|] ==> *a^n ≤ a^N*
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-Suc le-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac 1 * a^n ≤ a * a^m, simp*)
apply (*rule mult-mono*)
apply (*auto simp add*: *order-trans* [*OF zero-le-one*])
done

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:
*(1::'a::{ordered-semidom,recpower}) < a ==> a^n < a * a^n*
apply (*induct n*)
apply (*auto simp add*: *power-Suc mult-strict-left-mono order-less-trans* [*OF zero-less-one*])
done

lemma *power-strict-increasing*:
[|*n < N; (1::'a::{ordered-semidom,recpower}) < a*|] ==> *a^n < a^N*
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-less-power-Suc power-Suc less-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac 1 * a^n < a * a^m, simp*)
apply (*rule mult-strict-mono*)
apply (*auto simp add*: *order-less-trans* [*OF zero-less-one*] *order-less-imp-le*)
done

lemma *power-increasing-iff* [*simp*]:
1 < (b::'a::{ordered-semidom,recpower}) ==> (b ^ x ≤ b ^ y) = (x ≤ y)
by (*blast intro*: *power-le-imp-le-exp power-increasing order-less-imp-le*)

lemma *power-strict-increasing-iff* [*simp*]:
1 < (b::'a::{ordered-semidom,recpower}) ==> (b ^ x < b ^ y) = (x < y)
by (*blast intro*: *power-less-imp-less-exp power-strict-increasing*)

lemma *power-le-imp-le-base*:

assumes *le*: $a \hat{} Suc\ n \leq b \hat{} Suc\ n$
  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 $b \hat{} Suc\ n < a \hat{} Suc\ n$
    by (*simp only*: *prems power-strict-mono*)
  from *le* **and** *this* **show** *False*
    by (*simp add*: *linorder-not-less* [*symmetric*])
qed

lemma *power-less-imp-less-base*:
  fixes $a\ b :: 'a::\{ordered\text{-}semidom,recpower\}$
  assumes *less*: $a \hat{} n < b \hat{} n$
  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 $b \hat{} n \leq a \hat{} n$ **using** *nonneg* **by** (*rule power-mono*)
  thus $\sim a \hat{} n < b \hat{} n$ **by** (*simp only*: *linorder-not-less*)
qed

lemma *power-inject-base*:
    $[|\ a \hat{} Suc\ n = b \hat{} Suc\ n;\ 0 \leq a;\ 0 \leq b\ |]$
     $==> a = (b::'a::\{ordered\text{-}semidom,recpower\})$
by (*blast intro*: *power-le-imp-le-base order-antisym order-eq-refl sym*)

lemma *power-eq-imp-eq-base*:
  fixes $a\ b :: 'a::\{ordered\text{-}semidom,recpower\}$
  shows $[\![a \hat{} n = b \hat{} n;\ 0 \leq a;\ 0 \leq b;\ 0 < n]\!] \Longrightarrow a = b$
by (*cases n*, *simp-all*, *rule power-inject-base*)

15.2 Exponentiation for the Natural Numbers

instantiation *nat* :: *recpower*
begin

primrec *power-nat* **where**
  $p \hat{} 0 = (1::nat)$
  $|\ p \hat{} (Suc\ n) = (p::nat) * (p \hat{} n)$

instance proof
  fix $z\ n :: nat$
  show $z\hat{}0 = 1$ **by** *simp*
  show $z\hat{}(Suc\ n) = z * (z\hat{}n)$ **by** *simp*
qed

end

lemma *of-nat-power*:
of-nat (*m* ^ *n*) = (*of-nat m*::'*a*::{*semiring-1*,*recpower*}) ^ *n*
by (*induct n*, *simp-all add*: *power-Suc of-nat-mult*)

lemma *nat-one-le-power* [*simp*]: *1* ≤ *i* ==> *Suc 0* ≤ *i*^*n*
by (*insert one-le-power* [*of i n*], *simp*)

lemma *nat-zero-less-power-iff* [*simp*]: (*x*^*n* > *0*) = (*x* > (*0*::*nat*) | *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*^*m* < *i*^*n*
shows *m* < *n*
proof (*cases i* = *1*)
case *True* **with** *less power-one* [**where** *'a* = *nat*] **show** *?thesis* **by** *simp*
next
case *False* **with** *nonneg* **have** *1* < *i* **by** *auto*
from *power-strict-increasing-iff* [*OF this*] *less* **show** *?thesis* **..**
qed

lemma *power-diff*:
assumes *nz*: *a* ~= *0*
shows *n* <= *m* ==> (*a*::'*a*::{*recpower*, *field*}) ^ (*m*−*n*) = (*a*^*m*) / (*a*^*n*)
by (*induct m n rule*: *diff-induct*)
(*simp-all add*: *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

16 Divides: The division operators div, mod and the divides relation dvd

theory *Divides*
imports *Nat Power Product-Type*
uses ~~/*src/Provers/Arith/cancel-div-mod.ML*
begin

16.1 Syntactic division operations

class *div = times +*

fixes *div* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *div 70*)
fixes *mod* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *mod 70*)
begin

definition
dvd :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** *dvd 50*)
where
[*code func del*]: *m dvd n* $\longleftrightarrow$ $(\exists k.\ n = m * k)$

end

16.2 Abstract divisibility in commutative semirings.

class *semiring-div* = *comm-semiring-1-cancel* + *div* +
assumes *mod-div-equality*: *a div b* $*$ *b* + *a mod b* = *a*
and *div-by-0*: *a div 0* = *0*
and *mult-div*: *b* $\neq$ *0* $\Longrightarrow$ *a* $*$ *b div b* = *a*
begin

op div and *op mod*

lemma *div-by-1*: *a div 1* = *a*
using *mult-div* [*of 1 a*] *zero-neq-one* **by** *simp*

lemma *mod-by-1*: *a mod 1* = *0*
proof −
from *mod-div-equality* [*of a one*] *div-by-1* **have** *a* + *a mod 1* = *a* **by** *simp*
then have *a* + *a mod 1* = *a* + *0* **by** *simp*
then show *?thesis* **by** (*rule add-left-imp-eq*)
qed

lemma *mod-by-0*: *a mod 0* = *a*
using *mod-div-equality* [*of a zero*] **by** *simp*

lemma *mult-mod*: *a* $*$ *b mod b* = *0*
proof (*cases b* = *0*)
case *True* **then show** *?thesis* **by** (*simp add*: *mod-by-0*)
next
case *False* **with** *mult-div* **have** *abb*: *a* $*$ *b div b* = *a* **.**
from *mod-div-equality* **have** *a* $*$ *b div b* $*$ *b* + *a* $*$ *b mod b* = *a* $*$ *b* **.**
with *abb* **have** *a* $*$ *b* + *a* $*$ *b mod b* = *a* $*$ *b* + *0* **by** *simp*
then show *?thesis* **by** (*rule add-left-imp-eq*)
qed

lemma *mod-self*: *a mod a* = *0*
using *mult-mod* [*of one*] **by** *simp*

lemma *div-self*: *a* $\neq$ *0* $\Longrightarrow$ *a div a* = *1*
using *mult-div* [*of - one*] **by** *simp*

lemma *div-0*: *0 div a = 0*
proof (*cases a = 0*)
  case *True* **then show** *?thesis* **by** (*simp add*: *div-by-0*)
next
  case *False* **with** *mult-div* **have** *0 * a div a = 0* **.**
  then show *?thesis* **by** *simp*
qed

lemma *mod-0*: *0 mod a = 0*
  using *mod-div-equality* [*of zero a*] *div-0* **by** *simp*

lemma *mod-div-equality2*: *b * (a div b) + a mod b = a*
  unfolding *mult-commute* [*of b*]
  by (*rule mod-div-equality*)

lemma *div-mod-equality*: *((a div b) * b + a mod b) + c = a + c*
  by (*simp add*: *mod-div-equality*)

lemma *div-mod-equality2*: *(b * (a div b) + a mod b) + c = a + c*
  by (*simp add*: *mod-div-equality2*)

The *op dvd* relation

lemma *dvdI* [*intro?*]: *a = b * c ⟹ b dvd a*
  unfolding *dvd-def* **..**

lemma *dvdE* [*elim?*]: *b dvd a ⟹ (⋀c. a = b * c ⟹ P) ⟹ P*
  unfolding *dvd-def* **by** *blast*

lemma *dvd-def-mod* [*code func*]: *a dvd b ⟷ b mod a = 0*
proof
  assume *b mod a = 0*
  with *mod-div-equality* [*of b a*] **have** *b div a * a = b* **by** *simp*
  then have *b = a * (b div a)* **unfolding** *mult-commute* **..**
  then have *∃c. b = a * c* **..**
  then show *a dvd b* **unfolding** *dvd-def* **.**
next
  assume *a dvd b*
  then have *∃c. b = a * c* **unfolding** *dvd-def* **.**
  then obtain *c* **where** *b = a * c* **..**
  then have *b mod a = a * c mod a* **by** *simp*
  then have *b mod a = c * a mod a* **by** (*simp add*: *mult-commute*)
  then show *b mod a = 0* **by** (*simp add*: *mult-mod*)
qed

lemma *dvd-refl*: *a dvd a*
  unfolding *dvd-def-mod mod-self* **..**

lemma *dvd-trans*:
  assumes *a dvd b* **and** *b dvd c*

shows *a dvd c*
proof −
from *assms* **obtain** *v* **where** *b = a * v* **unfolding** *dvd-def* **by** *auto*
moreover from *assms* **obtain** *w* **where** *c = b * w* **unfolding** *dvd-def* **by** *auto*
ultimately have *c = a * (v * w)* **by** (*simp add*: *mult-assoc*)
then show *?thesis* **unfolding** *dvd-def* **..**
qed

lemma *zero-dvd-iff* [*noatp*]: *0 dvd a ⟷ a = 0*
unfolding *dvd-def* **by** *simp*

lemma *dvd-0*: *a dvd 0*
unfolding *dvd-def* **proof**
show *0 = a * 0* **by** *simp*
qed

lemma *one-dvd*: *1 dvd a*
unfolding *dvd-def* **by** *simp*

lemma *dvd-mult*: *a dvd c ⟹ a dvd (b * c)*
unfolding *dvd-def* **by** (*blast intro*: *mult-left-commute*)

lemma *dvd-mult2*: *a dvd b ⟹ a dvd (b * c)*
apply (*subst mult-commute*)
apply (*erule dvd-mult*)
done

lemma *dvd-triv-right*: *a dvd b * a*
by (*rule dvd-mult*) (*rule dvd-refl*)

lemma *dvd-triv-left*: *a dvd a * b*
by (*rule dvd-mult2*) (*rule dvd-refl*)

lemma *mult-dvd-mono*: *a dvd c ⟹ b dvd d ⟹ a * b dvd c * d*
apply (*unfold dvd-def*, *clarify*)
apply (*rule-tac x = k * ka* **in** *exI*)
apply (*simp add*: *mult-ac*)
done

lemma *dvd-mult-left*: *a * b dvd c ⟹ a dvd c*
by (*simp add*: *dvd-def mult-assoc*, *blast*)

lemma *dvd-mult-right*: *a * b dvd c ⟹ b dvd c*
unfolding *mult-ac* [*of a*] **by** (*rule dvd-mult-left*)

end

16.3 Division on *nat*

We define *op div* and *op mod* on *nat* by means of a characteristic relation with two input arguments m, n and two output arguments q(uotient) and r(emainder).

definition *divmod-rel* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
divmod-rel m n q r ⟷ *m* = *q* ∗ *n* + *r* ∧ (*if n* > *0 then* 0 ≤ *r* ∧ *r* < *n else q* = *0*)

divmod-rel is total:

lemma *divmod-rel-ex*:
obtains *q r* **where** *divmod-rel m n q r*
proof (*cases n* = *0*)
case *True* **with** *that* **show** *thesis*
by (*auto simp add*: *divmod-rel-def*)
next
case *False*
have ∃ *q r*. *m* = *q* ∗ *n* + *r* ∧ *r* < *n*
proof (*induct m*)
case *0* **with** ‹*n* ≠ *0*›
have (*0*::*nat*) = *0* ∗ *n* + *0* ∧ *0* < *n* **by** *simp*
then show *?case* **by** *blast*
next
case (*Suc m*) **then obtain** *q′ r′*
where *m*: *m* = *q′* ∗ *n* + *r′* **and** *n*: *r′* < *n* **by** *auto*
then show *?case* **proof** (*cases Suc r′* < *n*)
case *True*
from *m n* **have** *Suc m* = *q′* ∗ *n* + *Suc r′* **by** *simp*
with *True* **show** *?thesis* **by** *blast*
next
case *False* **then have** *n* ≤ *Suc r′* **by** *auto*
moreover from *n* **have** *Suc r′* ≤ *n* **by** *auto*
ultimately have *n* = *Suc r′* **by** *auto*
with *m* **have** *Suc m* = *Suc q′* ∗ *n* + *0* **by** *simp*
with ‹*n* ≠ *0*› **show** *?thesis* **by** *blast*
qed
qed
with *that* **show** *thesis*
using ‹*n* ≠ *0*› **by** (*auto simp add*: *divmod-rel-def*)
qed

divmod-rel is injective:

lemma *divmod-rel-unique-div*:
assumes *divmod-rel m n q r*
and *divmod-rel m n q′ r′*
shows *q* = *q′*
proof (*cases n* = *0*)
case *True* **with** *assms* **show** *?thesis*

```
    by (simp add: divmod-rel-def)
next
  case False
  have aux: ⋀q r q' r'. q' * n + r' = q * n + r ⟹ r < n ⟹ q' ≤ (q::nat)
  apply (rule leI)
  apply (subst less-iff-Suc-add)
  apply (auto simp add: add-mult-distrib)
  done
  from ‹n ≠ 0› assms show ?thesis
    by (auto simp add: divmod-rel-def
      intro: order-antisym dest: aux sym)
qed

lemma divmod-rel-unique-mod:
  assumes divmod-rel m n q r
    and divmod-rel m n q' r'
  shows r = r'
proof −
  from assms have q = q' by (rule divmod-rel-unique-div)
  with assms show ?thesis by (simp add: divmod-rel-def)
qed
```

We instantiate divisibility on the natural numbers by means of *divmod-rel*:

```
instantiation nat :: semiring-div
begin

definition divmod :: nat ⇒ nat ⇒ nat × nat where
  [code func del]: divmod m n = (THE (q, r). divmod-rel m n q r)

definition div-nat where
  m div n = fst (divmod m n)

definition mod-nat where
  m mod n = snd (divmod m n)

lemma divmod-div-mod:
  divmod m n = (m div n, m mod n)
  unfolding div-nat-def mod-nat-def by simp

lemma divmod-eq:
  assumes divmod-rel m n q r
  shows divmod m n = (q, r)
  using assms by (auto simp add: divmod-def
    dest: divmod-rel-unique-div divmod-rel-unique-mod)

lemma div-eq:
  assumes divmod-rel m n q r
  shows m div n = q
  using assms by (auto dest: divmod-eq simp add: div-nat-def)
```

lemma *mod-eq*:
assumes *divmod-rel m n q r*
shows *m mod n = r*
using *assms* **by** (*auto dest*: *divmod-eq simp add*: *mod-nat-def*)

lemma *divmod-rel*: *divmod-rel m n* (*m div n*) (*m mod n*)
proof −
from *divmod-rel-ex*
obtain *q r* **where** *rel*: *divmod-rel m n q r* **.**
moreover with *div-eq mod-eq* **have** *m div n = q* **and** *m mod n = r*
by *simp-all*
ultimately show *?thesis* **by** *simp*
qed

lemma *divmod-zero*:
divmod m 0 = (*0*, *m*)
proof −
from *divmod-rel* [*of m 0*] **show** *?thesis*
unfolding *divmod-div-mod divmod-rel-def* **by** *simp*
qed

lemma *divmod-base*:
assumes *m < n*
shows *divmod m n* = (*0*, *m*)
proof −
from *divmod-rel* [*of m n*] **show** *?thesis*
unfolding *divmod-div-mod divmod-rel-def*
using *assms* **by** (*cases m div n = 0*)
(*auto simp add*: *gr0-conv-Suc* [*of m div n*])
qed

lemma *divmod-step*:
assumes *0 < n* **and** *n ≤ m*
shows *divmod m n* = (*Suc* ((*m − n*) *div n*), (*m − n*) *mod n*)
proof −
from *divmod-rel* **have** *divmod-m-n*: *divmod-rel m n* (*m div n*) (*m mod n*) **.**
with *assms* **have** *m-div-n*: *m div n ≥ 1*
by (*cases m div n*) (*auto simp add*: *divmod-rel-def*)
from *assms divmod-m-n* **have** *divmod-rel* (*m − n*) *n* (*m div n − 1*) (*m mod n*)
by (*cases m div n*) (*auto simp add*: *divmod-rel-def*)
with *divmod-eq* **have** *divmod* (*m − n*) *n* = (*m div n − 1*, *m mod n*) **by** *simp*
moreover from *divmod-div-mod* **have** *divmod* (*m − n*) *n* = ((*m − n*) *div n*,
(*m − n*) *mod n*) **.**
ultimately have *m div n = Suc* ((*m − n*) *div n*)
and *m mod n* = (*m − n*) *mod n* **using** *m-div-n* **by** *simp-all*
then show *?thesis* **using** *divmod-div-mod* **by** *simp*
qed

The "recursion" equations for *op div* and *op mod*

lemma *div-less* [*simp*]:
fixes *m n* :: *nat*
assumes $m < n$
shows *m div n = 0*
using *assms divmod-base divmod-div-mod* **by** *simp*

lemma *le-div-geq*:
fixes *m n* :: *nat*
assumes $0 < n$ **and** $n \leq m$
shows *m div n = Suc ((m − n) div n)*
using *assms divmod-step divmod-div-mod* **by** *simp*

lemma *mod-less* [*simp*]:
fixes *m n* :: *nat*
assumes $m < n$
shows *m mod n = m*
using *assms divmod-base divmod-div-mod* **by** *simp*

lemma *le-mod-geq*:
fixes *m n* :: *nat*
assumes $n \leq m$
shows *m mod n = (m − n) mod n*
using *assms divmod-step divmod-div-mod* **by** (*cases n = 0*) *simp-all*

instance proof
fix *m n* :: *nat* **show** *m div n * n + m mod n = m*
using *divmod-rel* [*of m n*] **by** (*simp add*: *divmod-rel-def*)
next
fix *n* :: *nat* **show** *n div 0 = 0*
using *divmod-zero divmod-div-mod* [*of n 0*] **by** *simp*
next
fix *m n* :: *nat* **assume** $n \neq 0$ **then show** *m * n div n = m*
by (*induct m*) (*simp-all add*: *le-div-geq*)
qed

end

Simproc for cancelling *op div* and *op mod*

lemmas *mod-div-equality* = *semiring-div-class.times-div-mod-plus-zero-one.mod-div-equality* [*of m*::*nat n*, *standard*]
lemmas *mod-div-equality2* = *mod-div-equality2* [*of n*::*nat m*, *standard*]
lemmas *div-mod-equality* = *div-mod-equality* [*of m*::*nat n k*, *standard*]
lemmas *div-mod-equality2* = *div-mod-equality2* [*of m*::*nat n k*, *standard*]

ML ⟪
structure CancelDivModData =
struct

val div-name = @{*const-name div*};

val mod-name = @{const-name mod};
val mk-binop = HOLogic.mk-binop;
val mk-sum = ArithData.mk-sum;
val dest-sum = ArithData.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 ArithData.prove-conv all-tac (ArithData.simp-all-tac simps) end;

end;

structure CancelDivMod = CancelDivModFun(CancelDivModData);

val cancel-div-mod-proc = Simplifier.simproc @{theory}
cancel-div-mod [(m::nat) + n] (K CancelDivMod.proc);

Addsimprocs[cancel-div-mod-proc];
⟩⟩

code generator setup

lemma *divmod-if* [*code*]: *divmod m n = (if n = 0 ∨ m < n then (0, m) else*
let (q, r) = divmod (m − n) n in (Suc q, r))
by (*simp add*: *divmod-zero divmod-base divmod-step*)
(*simp add*: *divmod-div-mod*)

code-modulename *SML*
Divides Nat

code-modulename *OCaml*
Divides Nat

code-modulename *Haskell*
Divides Nat

16.3.1 Quotient

lemmas *DIVISION-BY-ZERO-DIV* [*simp*] = *div-by-0* [*of a::nat, standard*]
lemmas *div-0* [*simp*] = *semiring-div-class.div-0* [*of n::nat, standard*]

lemma *div-geq*: *0 < n ⟹ ¬ m < n ⟹ m div n = Suc ((m − n) div n)*
by (*simp add*: *le-div-geq linorder-not-less*)

lemma *div-if*: *0 < n ⟹ m div n = (if m < n then 0 else Suc ((m − n) div n))*

by (*simp add*: *div-geq*)

lemma *div-mult-self-is-m* [*simp*]: *0*<*n* ==> (*m*∗*n*) *div n* = (*m*::*nat*)
by (*rule mult-div*) *simp*

lemma *div-mult-self1-is-m* [*simp*]: *0*<*n* ==> (*n*∗*m*) *div n* = (*m*::*nat*)
by (*simp add*: *mult-commute*)

16.3.2 Remainder

lemmas *DIVISION-BY-ZERO-MOD* [*simp*] = *mod-by-0* [*of a*::*nat*, *standard*]
lemmas *mod-0* [*simp*] = *semiring-div-class.mod-0* [*of n*::*nat*, *standard*]

lemma *mod-less-divisor* [*simp*]:
fixes *m n* :: *nat*
assumes *n* > *0*
shows *m mod n* < (*n*::*nat*)
using *assms divmod-rel* **unfolding** *divmod-rel-def* **by** *auto*

lemma *mod-less-eq-dividend* [*simp*]:
fixes *m n* :: *nat*
shows *m mod n* ≤ *m*
proof (*rule add-leD2*)
from *mod-div-equality* **have** *m div n* ∗ *n* + *m mod n* = *m* **.**
then show *m div n* ∗ *n* + *m mod n* ≤ *m* **by** *auto*
qed

lemma *mod-geq*: ¬ *m* < (*n*::*nat*) ⟹ *m mod n* = (*m* − *n*) *mod n*
by (*simp add*: *le-mod-geq linorder-not-less*)

lemma *mod-if*: *m mod* (*n*::*nat*) = (*if m* < *n then m else* (*m* − *n*) *mod n*)
by (*simp add*: *le-mod-geq*)

lemma *mod-1* [*simp*]: *m mod Suc 0* = *0*
by (*induct m*) (*simp-all add*: *mod-geq*)

lemmas *mod-self* [*simp*] = *semiring-div-class.mod-self* [*of n*::*nat*, *standard*]

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 le-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*)

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-le-divisor*[*simp*]: $0 < n \Longrightarrow m\ mod\ n \leq (n::nat)$
apply (*drule mod-less-divisor* [**where** $m = m$])
apply *simp*
done

16.3.3 Quotient and Remainder

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

lemma *divmod-rel-mult1-eq*:

```
  [| divmod-rel b c q r; c > 0 |]
   ==> divmod-rel (a*b) c (a*q + a*r div c) (a*r mod c)
by (auto simp add: split-ifs mult-ac divmod-rel-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: divmod-rel [THEN divmod-rel-mult1-eq, THEN div-eq])
done

lemma mod-mult1-eq: (a*b) mod c = a*(b mod c) mod (c::nat)
apply (cases c = 0, simp)
apply (blast intro: divmod-rel [THEN divmod-rel-mult1-eq, THEN mod-eq])
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 divmod-rel-add1-eq:
  [| divmod-rel a c aq ar; divmod-rel b c bq br;  c > 0 |]
   ==> divmod-rel (a + b) c (aq + bq + (ar+br) div c) ((ar + br) mod c)
by (auto simp add: split-ifs mult-ac divmod-rel-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: divmod-rel-add1-eq [THEN div-eq] divmod-rel)
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: divmod-rel-add1-eq [THEN mod-eq] divmod-rel)
done

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 *divmod-rel-mult2-eq*: [| *divmod-rel a b q r*; *0 < b*; *0 < c* |]
==> *divmod-rel a* (*b*∗*c*) (*q div c*) (*b*∗(*q mod c*) + *r*)
by (*auto simp add*: *mult-ac divmod-rel-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*: *divmod-rel* [*THEN divmod-rel-mult2-eq*, *THEN div-eq*])
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 divmod-rel* [*THEN divmod-rel-mult2-eq*, *THEN mod-eq*])
done

16.3.4 Cancellation of Common Factors in Division

lemma *div-mult-mult-lemma*:
[| (*0*::*nat*) < *b*; *0 < c* |] ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
by (*auto simp add*: *div-mult2-eq*)

lemma *div-mult-mult1* [*simp*]: (*0*::*nat*) < *c* ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
apply (*cases b = 0*)
apply (*auto simp add*: *linorder-neq-iff* [*of b*] *div-mult-mult-lemma*)
done

lemma *div-mult-mult2* [*simp*]: (*0*::*nat*) < *c* ==> (*a*∗*c*) *div* (*b*∗*c*) = *a div b*
apply (*drule div-mult-mult1*)
apply (*auto simp add*: *mult-commute*)
done

16.3.5 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: *m div Suc 0* = *m*
by (*induct m*) (*simp-all add*: *div-geq*)

lemmas *div-self* [*simp*] = *semiring-div-class.div-self* [*of n*::*nat*, *standard*]

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*

16.3.6 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*)

lemmas *dvd-refl* [*simp*] = *semiring-div-class.dvd-refl* [*of m::nat*, *standard*]
lemmas *dvd-trans* [*trans*] = *semiring-div-class.dvd-trans* [*of m::nat n p*, *standard*]

lemma *dvd-anti-sym*: [| *m dvd n*; *n dvd m* |] ==> *m = (n::nat)*
unfolding *dvd-def*
by (*force dest*: *mult-eq-self-implies-10 simp add*: *mult-assoc mult-eq-1-iff*)

op dvd is a partial order

interpretation *dvd*: *order* [*op dvd* λ*n m* :: *nat*. *n dvd m* ∧ *n* ≠ *m*]
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

lemmas *dvd-eq-mod-eq-0* = *dvd-def-mod* [*of k*::*nat n*, *standard*]

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 *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 nat-zero-less-power-iff [simp]: (x^n > 0) = (x > (0::nat) | n=0)
 by (induct n) auto

lemma power-le-dvd [rule-format]: k^j dvd n --> i≤j --> k^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^m dvd i^n;  (1::nat) < i|] ==> m ≤ n
 apply (rule power-le-imp-le-exp, assumption)
 apply (erule dvd-imp-le, simp)
 done

lemma mod-eq-0-iff: (m mod d = 0) = (∃q::nat. m = d*q)
 by (auto simp add: dvd-eq-mod-eq-0 [symmetric] dvd-def)

lemmas mod-eq-0D [dest!] = mod-eq-0-iff [THEN iffD1]


lemma mod-eqD: (m mod d = r) ==> ∃q::nat. m = r + q*d
 apply (cut-tac 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 ⟶ P 0) ∧ (k ≠ 0 ⟶ (!i. !j<k. n = k*i + j ⟶ P i)))
 (is ?P = ?Q is - = (- ∧ (- ⟶ ?R)))
proof
 assume P: ?P
 show ?Q
 proof (cases)
  assume k = 0
  with P show ?Q by(simp add:DIVISION-BY-ZERO-DIV)
 next
  assume not0: k ≠ 0
  thus ?Q
  proof (simp, intro allI impI)
   fix i j
   assume n: n = k*i + j and j: j < k
   show P i
   proof (cases)
    assume i = 0
```

```
      with n j P show P i by simp
    next
      assume i ≠ 0
      with not0 n j P show P i by(simp add:add-ac)
    qed
  qed
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:
  assumes 0 < n
  shows n * q ≤ m ∧ m < n * Suc q ⟷ q = ((m::nat) div n) (is ?lhs ⟷ ?rhs)
proof
  assume ?rhs
  with mult-div-cancel have nq: n * q = m - (m mod n) by simp
  then have A: n * q ≤ m by simp
  have n - (m mod n) > 0 using mod-less-divisor assms by auto
  then have m < m + (n - (m mod n)) by simp
  then have m < n + (m - (m mod n)) by simp
  with nq have m < n + n * q by simp
  then have B: m < n * Suc q by simp
  from A B show ?lhs ..
next
  assume P: ?lhs
  then have divmod-rel m n q (m - n * q)
    unfolding divmod-rel-def by (auto simp add: mult-ac)
  then show ?rhs using divmod-rel by (rule divmod-rel-unique-div)
qed

theorem split-div':
  P ((m::nat) div n) = ((n = 0 ∧ 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*:
$P(n\ mod\ k :: nat) =$
$((k = 0 \longrightarrow P\ n) \wedge (k \neq 0 \longrightarrow (!i.\ !j{<}k.\ n = k{*}i + j \longrightarrow P\ j)))$
(**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

16.3.7 An "induction" law for modulus arithmetic.

lemma *mod-induct-0*:
assumes *step*: $\forall i<p.\ P\ i \longrightarrow P\ ((Suc\ i)\ mod\ p)$
and *base*: $P\ i$ **and** *i*: $i<p$
shows $P\ 0$
proof (*rule ccontr*)
assume *contra*: $\neg(P\ 0)$
from *i* **have** *p*: $0<p$ **by** *simp*
have $\forall k.\ 0<k \longrightarrow \neg\ P\ (p-k)$ (**is** $\forall k.\ ?A\ k$)
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 $\neg(Suc\ n < p)$
hence $p - Suc\ n = 0$
by *simp*
with *y contra* **show** *False*
by *simp*
qed
hence *n2*: $Suc\ (p - Suc\ n) = p-n$ **by** *arith*
from *p* **have** $p - Suc\ n < p$ **by** *arith*
with *y step* **have** *z*: $P\ ((Suc\ (p - Suc\ n))\ mod\ p)$
by *blast*
show *False*
proof (*cases n=0*)
case *True*
with *z n2 contra* **show** *?thesis* **by** *simp*
next
case *False*
with *p* **have** $p-n < p$ **by** *arith*
with *z n2 False ih* **show** *?thesis* **by** *simp*
qed
qed
qed
qed
moreover

```
  from i obtain k where 0<k ∧ i+k=p
    by (blast dest: less-imp-add-positive)
  hence 0<k ∧ i=p−k by auto
  moreover
  note base
  ultimately
  show False by blast
qed

lemma mod-induct:
  assumes step: ∀ i<p. P i ⟶ P ((Suc i) mod p)
  and base: P i and i: i<p and j: j<p
  shows P j
proof −
  have ∀ j<p. P j
  proof
    fix j
    show j<p ⟶ P j (is ?A j)
    proof (induct j)
      from step base i show ?A 0
        by (auto elim: mod-induct-0)
    next
      fix k
      assume ih: ?A k
      show ?A (Suc k)
      proof
        assume suc: Suc k < p
        hence k: k<p by simp
        with ih have P k ..
        with step k have P (Suc k mod p)
          by blast
        moreover
        from suc have Suc k mod p = Suc k
          by simp
        ultimately
        show P (Suc k) by simp
      qed
    qed
  qed
  with j show ?thesis by blast
qed

end
```

17 Relation: Relations

```
theory Relation
imports Product-Type
```

begin

17.1 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*)

abbreviation
reflexive :: $('a * 'a)$ *set* => *bool* **where** — reflexivity over a type
reflexive == *refl UNIV*

definition
sym :: $('a * 'a)$ *set* => *bool* **where** — symmetry predicate
sym r == *ALL x y*. (*x*,*y*): *r* −−> (*y*,*x*): *r*

definition
antisym :: $('a * 'a)$ *set* => *bool* **where** — antisymmetry predicate
antisym r == *ALL x y*. (*x*,*y*):*r* −−> (*y*,*x*):*r* −−> *x*=*y*

definition
trans :: $('a * 'a)$ *set* => *bool* **where** — transitivity predicate
trans r == (*ALL x y z*. (*x*,*y*):*r* −−> (*y*,*z*):*r* −−> (*x*,*z*):*r*)

definition
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*}

17.2 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*)

17.3 Diagonal: identity over a set

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

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

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

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

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

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

17.4 Composition of two relations

lemma *rel-compI* [*intro*]:
$(a, b) : s \Longrightarrow (b, c) : r \Longrightarrow (a, c) : r\ O\ s$
by (*unfold rel-comp-def*) *blast*

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s \Longrightarrow$
$(!!x\ y\ z.\ xz = (x, z) \Longrightarrow (x, y) : s \Longrightarrow (y, z) : r \Longrightarrow P) \Longrightarrow P$
by (*unfold rel-comp-def*) (*iprover elim*!: *CollectE splitE exE conjE*)

lemma *rel-compEpair*:
$(a, c) : r\ O\ s \Longrightarrow (!!y.\ (a, y) : s \Longrightarrow (y, c) : r \Longrightarrow P) \Longrightarrow P$
by (*iprover elim*: *rel-compE Pair-inject ssubst*)

lemma *R-O-Id* [*simp*]: $R\ O\ Id = R$
by *fast*

lemma *Id-O-R* [*simp*]: $Id\ O\ R = R$
by *fast*

lemma *rel-comp-empty1*[*simp*]: $\{\}\ O\ R = \{\}$
by *blast*

lemma *rel-comp-empty2*[*simp*]: $R\ O\ \{\} = \{\}$
by *blast*

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
by *blast*

lemma *trans-O-subset*: $trans\ r \Longrightarrow r\ O\ r \subseteq r$
by (*unfold trans-def*) *blast*

lemma *rel-comp-mono*: $r' \subseteq r \Longrightarrow s' \subseteq s \Longrightarrow (r'\ O\ s') \subseteq (r\ O\ s)$
by *blast*

lemma *rel-comp-subset-Sigma*:
$s \subseteq A \times B \Longrightarrow r \subseteq B \times C \Longrightarrow (r\ O\ s) \subseteq A \times C$
by *blast*

17.5 Reflexivity

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

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

lemma *reflD1*: $refl\ A\ r \Longrightarrow (x, y) : r \Longrightarrow x : A$
by (*unfold refl-def*) *blast*

lemma *reflD2*: $refl\ A\ r \Longrightarrow (x, y) : r \Longrightarrow y : A$
by (*unfold refl-def*) *blast*

lemma *refl-Int*: $refl\ A\ r \Longrightarrow refl\ B\ s \Longrightarrow refl\ (A \cap B)\ (r \cap s)$
by (*unfold refl-def*) *blast*

lemma *refl-Un*: $refl\ A\ r \Longrightarrow refl\ B\ s \Longrightarrow refl\ (A \cup B)\ (r \cup s)$
by (*unfold refl-def*) *blast*

lemma *refl-INTER*:
$ALL\ x{:}S.\ refl\ (A\ x)\ (r\ x) \Longrightarrow refl\ (INTER\ S\ A)\ (INTER\ S\ r)$
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-empty*[*simp*]: $refl\ \{\}\ \{\}$
by(*simp add*:*refl-def*)

lemma *refl-diag*: $refl\ A\ (diag\ A)$
by (*rule reflI* [*OF diag-subset-Times diagI*])

17.6 Antisymmetry

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

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

lemma *antisym-subset*: $r \subseteq s \Longrightarrow antisym\ s \Longrightarrow antisym\ r$
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*

17.7 Symmetry

lemma *symI*: $(!!a\ b.\ (a, b) : r \Longrightarrow (b, a) : r) \Longrightarrow sym\ r$
by (*unfold sym-def*) *iprover*

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

lemma *sym-Int*: $sym\ r \Longrightarrow sym\ s \Longrightarrow sym\ (r \cap s)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-Un*: $sym\ r \Longrightarrow sym\ s \Longrightarrow sym\ (r \cup s)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-INTER*: $ALL\ x{:}S.\ sym\ (r\ x) \Longrightarrow sym\ (INTER\ S\ r)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-UNION*: $ALL\ x{:}S.\ sym\ (r\ x) \Longrightarrow sym\ (UNION\ S\ r)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-diag* [*simp*]: *sym* (*diag A*)
by (*rule symI*) *clarify*

17.8 Transitivity

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

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

lemma *trans-Int*: $trans\ r \Longrightarrow trans\ s \Longrightarrow trans\ (r \cap s)$
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-INTER*: $ALL\ x{:}S.\ trans\ (r\ x) \Longrightarrow trans\ (INTER\ S\ r)$
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-diag* [*simp*]: *trans* (*diag A*)
by (*fast intro*: *transI elim*: *transD*)

17.9 Converse

lemma *converse-iff* [*iff*]: $((a,b): r\hat{}-1) = ((b,a) : r)$
by (*simp add*: *converse-def*)

lemma *converseI*[*sym*]: $(a,\ b) : r \Longrightarrow (b,\ a) : r\hat{}-1$
by (*simp add*: *converse-def*)

lemma *converseD*[*sym*]: $(a,b) : r\hat{}-1 \Longrightarrow (b,\ a) : r$
by (*simp add*: *converse-def*)

lemma *converseE* [*elim*!]:
$yx : r\hat{}-1 \Longrightarrow (!!x\ y.\ yx = (y,\ x) \Longrightarrow (x,\ y) : r \Longrightarrow P) \Longrightarrow P$
— More general than *converseD*, as it "splits" the member of the relation.
by (*unfold converse-def*) (*iprover elim*!: *CollectE splitE bexE*)

lemma *converse-converse* [*simp*]: $(r\hat{}-1)\hat{}-1 = r$
by (*unfold converse-def*) *blast*

lemma *converse-rel-comp*: $(r\ O\ s)\hat{}-1 = s\hat{}-1\ O\ r\hat{}-1$
by *blast*

lemma *converse-Int*: $(r \cap s)\hat{}-1 = r\hat{}-1 \cap s\hat{}-1$
by *blast*

lemma *converse-Un*: $(r \cup s)\hat{}-1 = r\hat{}-1 \cup s\hat{}-1$
by *blast*

lemma *converse-INTER*: $(INTER\ S\ r)\hat{}-1 = (INT\ x{:}S.\ (r\ x)\hat{}-1)$
by *fast*

lemma *converse-UNION*: $(UNION\ S\ r)\hat{}-1 = (UN\ x{:}S.\ (r\ x)\hat{}-1)$
by *blast*

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

lemma *converse-diag* [*simp*]: $(diag\ A)\hat{}-1 = diag\ A$
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*^−*1* = *r*)
by (*unfold sym-def*) *fast*

lemma *sym-Un-converse*: *sym* (*r* ∪ *r*^−*1*)
by (*unfold sym-def*) *blast*

lemma *sym-Int-converse*: *sym* (*r* ∩ *r*^−*1*)
by (*unfold sym-def*) *blast*

17.10 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*) = (⋃ *A*∈*S*. *Domain A*)
by *blast*

lemma *Domain-converse*[*simp*]: $Domain(r\hat{}-1) = Range\ r$
by(*auto simp*:*Range-def*)

lemma *Domain-mono*: $r \subseteq s \Longrightarrow Domain\ r \subseteq Domain\ s$
by *blast*

lemma *fst-eq-Domain*: $fst\ `\ R = Domain\ R$
by (*auto intro*!:*image-eqI*)

17.11 Range

lemma *Range-iff*: $(a : Range\ r) = (EX\ y.\ (y,\ a) : r)$
by (*simp add*: *Domain-def Range-def*)

lemma *RangeI* [*intro*]: $(a,\ b) : r \Longrightarrow b : Range\ r$
by (*unfold Range-def*) (*iprover intro*!: *converseI DomainI*)

lemma *RangeE* [*elim*!]: $b : Range\ r \Longrightarrow (!!x.\ (x,\ b) : r \Longrightarrow P) \Longrightarrow P$
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 *Range-converse*[*simp*]: $Range(r\hat{}-1) = Domain\ r$
by *blast*

lemma *snd-eq-Range*: $snd\ `\ R = Range\ R$
by (*auto intro*!:*image-eqI*)

17.12 Field

lemma *mono-Field*: $r \subseteq s \Longrightarrow Field\ r \subseteq Field\ s$
by(*auto simp*:*Field-def Domain-def Range-def*)

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

lemma *Field-insert*[*simp*]: $Field\ (insert\ (a,b)\ r) = \{a,b\} \cup Field\ r$
by(*auto simp*:*Field-def*)

lemma *Field-Un*[*simp*]: $Field\ (r \cup s) = Field\ r \cup Field\ s$
by(*auto simp*:*Field-def*)

lemma *Field-Union*[*simp*]: $Field\ (\bigcup R) = \bigcup (Field\ `\ R)$
by(*auto simp*:*Field-def*)

lemma *Field-converse*[*simp*]: $Field(r\hat{}-1) = Field\ r$
by(*auto simp*:*Field-def*)

17.13 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 ∩ B*
by *blast*

lemma *Image-Int-subset*: *R `` (A ∩ B) ⊆ R `` A ∩ R `` B*
by *blast*

lemma *Image-Int-eq*:
single-valued (converse R) ==> R `` (A ∩ B) = R `` A ∩ R `` B
by (*simp add*: *single-valued-def*, *blast*)

lemma *Image-Un*: *R `` (A ∪ B) = R `` A ∪ R `` B*
by *blast*

lemma *Un-Image*: *(R ∪ S) `` A = R `` A ∪ S `` A*
by *blast*

lemma *Image-subset*: *r ⊆ A × B ==> r``C ⊆ B*
by (*iprover intro*!: *subsetI elim*!: *ImageE dest*!: *subsetD SigmaD2*)

lemma *Image-eq-UN*: *r``B = (⋃y∈ B. r``{y})*
— NOT suitable for rewriting
by *blast*

lemma *Image-mono*: *r' ⊆ r ==> A' ⊆ A ==> (r' `` A') ⊆ (r `` A)*
by *blast*

lemma *Image-UN*: *(r `` (UNION A B)) = (⋃x∈A. r `` (B x))*
by *blast*

lemma *Image-INT-subset*: *(r `` INTER A B) ⊆ (⋂x∈A. r `` (B x))*
by *blast*

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
[|single-valued (r⁻¹); A≠{}|] ==> r `` INTER A B = (⋂x∈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 ⊆ B) = (A ⊆ - ((r^-1) `` (-B)))*
by *blast*

17.14 Single valued relations

lemma *single-valuedI*:
ALL x y. (x,y):r --> (ALL z. (x,z):r --> y=z) ==> single-valued r
by (*unfold single-valued-def*)

lemma *single-valuedD*:
single-valued r ==> (x, y) : r ==> (x, z) : r ==> y = z
by (*simp add*: *single-valued-def*)

lemma *single-valued-rel-comp*:
single-valued r ==> single-valued s ==> single-valued (r O s)
by (*unfold single-valued-def*) *blast*

lemma *single-valued-subset*:
r ⊆ s ==> single-valued s ==> single-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*

17.15 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*

17.16 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

17.17 Version of *lfp-induct* for binary relations

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

end

18 Predicate: Predicates

theory *Predicate*
imports *Inductive Relation*
begin

18.1 Equality and Subsets

lemma *pred-equals-eq*: $((\lambda x.\ x \in R) = (\lambda x.\ x \in S)) = (R = S)$
by (*simp add*: *mem-def*)

lemma *pred-equals-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) = (\lambda x\ y.\ (x,\ y) \in S)) = (R = S)$
by (*simp add*: *expand-fun-eq mem-def*)

lemma *pred-subset-eq*: $((\lambda x.\ x \in R) <= (\lambda x.\ x \in S)) = (R <= S)$
by (*simp add*: *mem-def*)

lemma *pred-subset-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) <= (\lambda x\ y.\ (x,\ y) \in S)) = (R <= S)$
by *fast*

18.2 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*)

18.3 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*)

18.4 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*!]: $(\sim B\ x \Longrightarrow A\ x) \Longrightarrow sup\ A\ B\ x$
by *auto*

lemma *sup2CI* [*intro*!]: $(\sim B\ x\ y \Longrightarrow A\ x\ y) \Longrightarrow sup\ A\ B\ x\ y$
by *auto*

lemma *sup1E* [*elim*!]: $sup\ A\ B\ x \Longrightarrow (A\ x \Longrightarrow P) \Longrightarrow (B\ x \Longrightarrow P) \Longrightarrow P$
by *simp iprover*

lemma *sup2E* [*elim*!]: $sup\ A\ B\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow P) \Longrightarrow (B\ x\ y \Longrightarrow P) \Longrightarrow P$
by *simp iprover*

18.5 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*!]: $A\ x \Longrightarrow B\ x \Longrightarrow inf\ A\ B\ x$
by *simp*

lemma *inf2I* [*intro*!]: $A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow inf\ A\ B\ x\ y$
by *simp*

lemma *inf1D1*: $inf\ A\ B\ x \Longrightarrow A\ x$
by *simp*

lemma *inf2D1*: $inf\ A\ B\ x\ y \Longrightarrow A\ x\ y$
by *simp*

lemma *inf1D2*: $inf\ A\ B\ x \Longrightarrow B\ x$
by *simp*

lemma *inf2D2*: $inf\ A\ B\ x\ y \Longrightarrow B\ x\ y$
by *simp*

lemma *inf1E* [*elim*!]: $inf\ A\ B\ x \Longrightarrow (A\ x \Longrightarrow B\ x \Longrightarrow P) \Longrightarrow P$
by *simp*

lemma *inf2E* [*elim*!]: $inf\ A\ B\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow P) \Longrightarrow P$
by *simp*

18.6 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*]: $a : A \Longrightarrow B\ a\ b \Longrightarrow (SUP\ x{:}A.\ B\ x)\ b$
by *auto*

lemma *SUP2-I* [*intro*]: $a : A \Longrightarrow B\ a\ b\ c \Longrightarrow (SUP\ x{:}A.\ B\ x)\ b\ c$
by *auto*

lemma *SUP1-E* [*elim*!]: $(SUP\ x{:}A.\ B\ x)\ b \Longrightarrow (!!x.\ x : A \Longrightarrow B\ x\ b \Longrightarrow R) \Longrightarrow R$
by *auto*

lemma *SUP2-E* [*elim*!]: $(SUP\ x{:}A.\ B\ x)\ b\ c \Longrightarrow (!!x.\ x : A \Longrightarrow B\ x\ b\ c \Longrightarrow R) \Longrightarrow R$
by *auto*

lemma *SUP-UN-eq*: $(SUP\ i.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (UN\ i.\ r\ i))$
by (*simp add*: *expand-fun-eq*)

lemma *SUP-UN-eq2*: $(SUP\ i.\ (\lambda x\ y.\ (x, y) \in r\ i)) = (\lambda x\ y.\ (x, y) \in (UN\ i.\ r\ i))$
by (*simp add*: *expand-fun-eq*)

18.7 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*]: $(INF\ x{:}A.\ B\ x)\ b ==> (B\ a\ b ==> R) ==> (a \sim: A ==> R) ==> R$
by *auto*

lemma *INF2-E* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b\ c ==> (B\ a\ b\ c ==> R) ==> (a \sim: A ==> R) ==> R$
by *auto*

lemma *INF-INT-eq*: $(INF\ i.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (INT\ i.\ r\ i))$
by (*simp add*: *expand-fun-eq*)

lemma *INF-INT-eq2*: $(INF\ i.\ (\lambda x\ y.\ (x, y) \in r\ i)) = (\lambda x\ y.\ (x, y) \in (INT\ i.\ r\ i))$
by (*simp add*: *expand-fun-eq*)

18.8 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*]:
$((\lambda x\ y.\ (x,\ y) \in r)\ OO\ (\lambda x\ y.\ (x,\ y) \in s)) = (\lambda x\ y.\ (x,\ y) \in r\ O\ s)$
by (*auto simp add*: *expand-fun-eq elim*: *pred-compE*)

18.9 Converse

inductive
conversep :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b \Rightarrow 'a \Rightarrow bool$
$((\text{-}\hat{}{-}{-}1)\ [1000]\ 1000)$
for r :: $'a \Rightarrow 'b \Rightarrow bool$
where
conversepI: $r\ a\ b \Longrightarrow r\hat{}{-}{-}1\ b\ a$

notation (*xsymbols*)
conversep $((\text{-}^{-1-1})\ [1000]\ 1000)$

lemma *conversepD*:
assumes *ab*: $r\hat{}{-}{-}1\ a\ b$
shows $r\ b\ a$ **using** *ab*
by *cases simp*

lemma *conversep-iff* [*iff*]: $r\hat{}{-}{-}1\ a\ b = r\ b\ a$
by (*iprover intro*: *conversepI dest*: *conversepD*)

lemma *conversep-converse-eq* [*pred-set-conv*]:
$(\lambda x\ y.\ (x,\ y) \in r)\hat{}{-}{-}1 = (\lambda x\ y.\ (x,\ y) \in r\hat{}{-}1)$
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-conversep* [*simp*]: $(r\hat{}{-}{-}1)\hat{}{-}{-}1 = r$
by (*iprover intro*: *order-antisym conversepI dest*: *conversepD*)

lemma *converse-pred-comp*: $(r\ OO\ s)\hat{}{-}{-}1 = s\hat{}{-}{-}1\ OO\ r\hat{}{-}{-}1$
by (*iprover intro*: *order-antisym conversepI pred-compI*
elim: *pred-compE dest*: *conversepD*)

lemma *converse-meet*: $(inf\ r\ s)\hat{}{-}{-}1 = inf\ r\hat{}{-}{-}1\ s\hat{}{-}{-}1$
by (*simp add*: *inf-fun-eq inf-bool-eq*)
(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *converse-join*: $(sup\ r\ s)\hat{}{-}{-}1 = sup\ r\hat{}{-}{-}1\ s\hat{}{-}{-}1$
by (*simp add*: *sup-fun-eq sup-bool-eq*)
(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *conversep-noteq* [*simp*]: $(op\ \tilde{}{=})\hat{}{-}{-}1 = op\ \tilde{}{=}$
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-eq* [*simp*]: $(op\ =)\hat{}{-}{-}1 = op\ =$

by (*auto simp add*: *expand-fun-eq*)

18.10 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 predicate1I*)

18.11 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* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Range r*)
by (*blast intro*!: *Orderings.order-antisym predicate1I*)

18.12 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* (λ*x y*. (*x*, *y*) ∈ *r*) *f* = (λ*x y*. (*x*, *y*) ∈ *inv-image r f*)
by (*simp add*: *inv-image-def inv-imagep-def*)

lemma *in-inv-imagep* [*simp*]: *inv-imagep r f x y* = *r* (*f x*) (*f y*)
by (*simp add*: *inv-imagep-def*)

18.13 The Powerset operator

definition *Powp* :: (*'a* ⇒ *bool*) ⇒ *'a set* ⇒ *bool* **where**
Powp A == λ*B*. ∀ *x* ∈ *B*. *A x*

lemma *Powp-Pow-eq* [*pred-set-conv*]: *Powp* (λ*x*. *x* ∈ *A*) = (λ*x*. *x* ∈ *Pow A*)
by (*auto simp add*: *Powp-def expand-fun-eq*)

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred pred-subset-eq*]

18.14 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

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

theory *Transitive-Closure*
imports *Predicate*
uses ~~/*src*/*Provers*/*trancl.ML*
begin

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

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

inductive-set
rtrancl :: ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ((-^*) [*1000*] *999*)
for *r* :: ('*a* × '*a*) *set*
where
rtrancl-refl [*intro*!, *Pure.intro*!, *simp*]: (*a*, *a*) : *r*^*
| *rtrancl-into-rtrancl* [*Pure.intro*]: (*a*, *b*) : *r*^* ==> (*b*, *c*) : *r* ==> (*a*, *c*) : *r*^*

inductive-set
trancl :: ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ((-^+) [*1000*] *999*)
for *r* :: ('*a* × '*a*) *set*
where
r-into-trancl [*intro*, *Pure.intro*]: (*a*, *b*) : *r* ==> (*a*, *b*) : *r*^+
| *trancl-into-trancl* [*Pure.intro*]: (*a*, *b*) : *r*^+ ==> (*b*, *c*) : *r* ==> (*a*, *c*) : *r*^+

notation
rtranclp ((-^**) [*1000*] *1000*) **and**
tranclp ((-^++) [*1000*] *1000*)

abbreviation

reflclp :: ('*a* => '*a* => *bool*) => '*a* => '*a* => *bool* ((-^==) [*1000*] *1000*)
where
r^== == *sup r op* =

abbreviation
reflcl :: ('*a* × '*a*) *set* => ('*a* × '*a*) *set* ((-^=) [*1000*] *999*) **where**
r^= == *r* ∪ *Id*

notation (*xsymbols*)
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-++) [*1000*] *1000*) **and**
reflclp ((-==) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-+) [*1000*] *999*) **and**
reflcl ((-=) [*1000*] *999*)

notation (*HTML* **output**)
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-++) [*1000*] *1000*) **and**
reflclp ((-==) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-+) [*1000*] *999*) **and**
reflcl ((-=) [*1000*] *999*)

19.1 Reflexive closure

lemma *reflexive-reflcl*[*simp*]: *reflexive*(*r*^=)
by(*simp add*:*refl-def*)

lemma *antisym-reflcl*[*simp*]: *antisym*(*r*^=) = *antisym r*
by(*simp add*:*antisym-def*)

lemma *trans-reflclI*[*simp*]: *trans r* ⟹ *trans*(*r*^=)
unfolding *trans-def* **by** *blast*

19.2 Reflexive-transitive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: (*sup* (λ*x y*. (*x*, *y*) ∈ *r*) *op* =) = (λ*x y*. (*x*, *y*) ∈ *r Un Id*)
by (*simp add*: *expand-fun-eq*)

lemma *r-into-rtrancl* [*intro*]: !!*p*. *p* ∈ *r* ==> *p* ∈ *r*^*
— *rtrancl* of *r* contains *r*
apply (*simp only*: *split-tupled-all*)
apply (*erule rtrancl-refl* [*THEN rtrancl-into-rtrancl*])
done

lemma *r-into-rtranclp* [*intro*]: *r x y* ==> *r*^** *x y*
— *rtrancl* of *r* contains *r*
by (*erule rtranclp.rtrancl-refl* [*THEN rtranclp.rtrancl-into-rtrancl*])

lemma *rtranclp-mono*: $r \leq s \Longrightarrow r\hat{}** \leq s\hat{}**$
— monotonicity of *rtrancl*
apply (*rule predicate2I*)
apply (*erule rtranclp.induct*)
apply (*rule-tac* [*2*] *rtranclp.rtrancl-into-rtrancl*, *blast*+)
done

lemmas *rtrancl-mono* = *rtranclp-mono* [*to-set*]

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

lemmas *rtrancl-induct* [*induct set*: *rtrancl*] = *rtranclp-induct* [*to-set*]

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

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

lemma *reflexive-rtrancl*: *reflexive* $(r\hat{}*)$
by (*unfold refl-def*) *fast*

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* $(r\hat{}*)$
proof (*rule transI*)
fix *x y z*
assume $(x, y) \in r^*$
assume $(y, z) \in r^*$
then show $(x, z) \in r^*$
proof *induct*
case *base*
show $(x, y) \in r^*$ **by** *fact*
next
case (*step u v*)
from ‹$(x, u) \in r^*$› **and** ‹$(u, v) \in r$›
show $(x, v) \in r^*$ **..**
qed
qed

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

lemma *rtranclp-trans*:
assumes *xy*: *r*^** *x y*
and *yz*: *r*^** *y z*
shows *r*^** *x z* **using** *yz xy*
by *induct iprover+*

lemma *rtranclE* [*cases set*: *rtrancl*]:
assumes *major*: (*a*::'*a*, *b*) : *r*^*
obtains
(*base*) *a* = *b*
| (*step*) *y* **where** (*a*, *y*) : *r*^* **and** (*y*, *b*) : *r*
— elimination of *rtrancl* – by induction on a special formula
apply (*subgoal-tac* (*a*::'*a*) = *b* | (*EX y*. (*a*,*y*) : *r*^* & (*y*,*b*) : *r*))
apply (*rule-tac* [*2*] *major* [*THEN rtrancl-induct*])
prefer *2* **apply** *blast*
prefer *2* **apply** *blast*
apply (*erule asm-rl exE disjE conjE base step*)+
done

lemma *rtrancl-Int-subset*: [| *Id* ⊆ *s*; *r O* (*r*^* ∩ *s*) ⊆ *s*|] ==> *r*^* ⊆ *s*
apply (*rule subsetI*)
apply (*rule-tac p*=*x* **in** *PairE*, *clarify*)
apply (*erule rtrancl-induct*, *auto*)
done

lemma *converse-rtranclp-into-rtranclp*:
r a b ⟹ *r*** *b c* ⟹ *r*** *a c*
by (*rule rtranclp-trans*) *iprover+*

lemmas *converse-rtrancl-into-rtrancl* = *converse-rtranclp-into-rtranclp* [*to-set*]

More r^* equations and inclusions.

lemma *rtranclp-idemp* [*simp*]: (*r*^**)^** = *r*^**
apply (*auto intro*!: *order-antisym*)
apply (*erule rtranclp-induct*)
apply (*rule rtranclp.rtrancl-refl*)
apply (*blast intro*: *rtranclp-trans*)
done

lemmas *rtrancl-idemp* [*simp*] = *rtranclp-idemp* [*to-set*]

lemma *rtrancl-idemp-self-comp* [*simp*]: *R*^* *O R*^* = *R*^*
apply (*rule set-ext*)
apply (*simp only*: *split-tupled-all*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *rtrancl-subset-rtrancl*: $r \subseteq s\hat{}* \Longrightarrow r\hat{}* \subseteq s\hat{}*$
apply (*drule rtrancl-mono*)
apply *simp*
done

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

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: $(sup\ (R\hat{}**)\ (S\hat{}**))\hat{}** = (sup\ R\ S)\hat{}**$
by (*blast intro*!: *rtranclp-subset intro*: *rtranclp-mono* [*THEN predicate2D*])

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

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

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflcl* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - Id)\hat{}* = r\hat{}*$
apply (*rule sym*)
apply (*rule rtrancl-subset*, *blast*, *clarify*)
apply (*rename-tac a b*)
apply (*case-tac a = b*)
apply *blast*
apply (*blast intro*!: *r-into-rtrancl*)
done

lemma *rtranclp-r-diff-Id*: $(inf\ r\ op\ \tilde{}=)\hat{}** = r\hat{}**$
apply (*rule sym*)
apply (*rule rtranclp-subset*)
apply *blast*+
done

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

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

theorem *rtranclp-converseI*:
assumes *r^** y x*
shows *(r^--1)^** x y*
using *assms*
by *induct* (*iprover intro*: *rtranclp-trans conversepI*)+

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

lemma *rtrancl-converse*: *(r^-1)^* = (r^*)^-1*
by (*fast dest*!: *rtrancl-converseD intro*!: *rtrancl-converseI*)

lemma *sym-rtrancl*: *sym r ==> sym (r^*)*
by (*simp only*: *sym-conv-converse-eq rtrancl-converse* [*symmetric*])

theorem *converse-rtranclp-induct*[*consumes 1*]:
assumes *major*: *r^** a b*
and *cases*: *P b* !!*y z*. [| *r y z*; *r^** *z b*; *P z* |] ==> *P y*
shows *P a*
using *rtranclp-converseI* [*OF major*]
by *induct* (*iprover intro*: *cases dest*!: *conversepD rtranclp-converseD*)+

lemmas *converse-rtrancl-induct* = *converse-rtranclp-induct* [*to-set*]

lemmas *converse-rtranclp-induct2* =
converse-rtranclp-induct [*of* - (*ax*,*ay*) (*bx*,*by*), *split-rule*,
consumes 1, *case-names refl step*]

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

lemma *converse-rtranclpE*:
assumes *major*: *r^** x z*
and *cases*: *x=z ==> P*
!!*y*. [| *r x y*; *r^** *y z* |] ==> *P*
shows *P*
apply (*subgoal-tac x = z | (EX y. r x y & r^** y z)*)
apply (*rule-tac* [*2*] *major* [*THEN converse-rtranclp-induct*])
prefer *2* **apply** *iprover*
prefer *2* **apply** *iprover*
apply (*erule asm-rl exE disjE conjE cases*)+
done

lemmas *converse-rtranclE* = *converse-rtranclpE* [*to-set*]

lemmas *converse-rtranclpE2* = *converse-rtranclpE* [*of* - (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemmas *converse-rtranclE2* = *converse-rtranclE* [*of* (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemma *r-comp-rtrancl-eq*: $r\ O\ r\hat{}* = r\hat{}*\ O\ r$
by (*blast elim*: *rtranclE converse-rtranclE*
intro: *rtrancl-into-rtrancl converse-rtrancl-into-rtrancl*)

lemma *rtrancl-unfold*: $r\hat{}* = Id\ Un\ r\ O\ r\hat{}*$
by (*auto intro*: *rtrancl-into-rtrancl elim*: *rtranclE*)

19.3 Transitive closure

lemma *trancl-mono*: $!!p.\ p \in r\hat{}+ ==> r \subseteq s ==> p \in s\hat{}+$
apply (*simp add*: *split-tupled-all*)
apply (*erule trancl.induct*)
apply (*iprover dest*: *subsetD*)+
done

lemma *r-into-trancl'*: $!!p.\ p : r ==> p : r\hat{}+$
by (*simp only*: *split-tupled-all*) (*erule r-into-trancl*)

Conversions between *trancl* and *rtrancl*.

lemma *tranclp-into-rtranclp*: $r\hat{}++\ a\ b ==> r\hat{}**\ a\ b$
by (*erule tranclp.induct*) *iprover*+

lemmas *trancl-into-rtrancl* = *tranclp-into-rtranclp* [*to-set*]

lemma *rtranclp-into-tranclp1*: **assumes** *r*: $r\hat{}**\ a\ b$
shows $!!c.\ r\ b\ c ==> r\hat{}++\ a\ c$ **using** *r*
by *induct iprover*+

lemmas *rtrancl-into-trancl1* = *rtranclp-into-tranclp1* [*to-set*]

lemma *rtranclp-into-tranclp2*: $[|\ r\ a\ b;\ r\hat{}**\ b\ c\ |] ==> r\hat{}++\ a\ c$
— intro rule from *r* and *rtrancl*
apply (*erule rtranclp.cases*)
apply *iprover*
apply (*rule rtranclp-trans* [*THEN rtranclp-into-tranclp1*])
apply (*simp* | *rule r-into-rtranclp*)+
done

lemmas *rtrancl-into-trancl2* = *rtranclp-into-tranclp2* [*to-set*]

Nice induction rule for *trancl*

lemma *tranclp-induct* [*consumes 1*, *case-names base step*, *induct pred*: *tranclp*]:
assumes $r\hat{}++\ a\ b$
and *cases*: $!!y.\ r\ a\ y ==> P\ y$
$!!y\ z.\ r\hat{}++\ a\ y ==> r\ y\ z ==> P\ y ==> P\ z$
shows $P\ b$
proof −
from ‹$r\hat{}++\ a\ b$› **have** $a = a --> P\ b$
by (*induct* %*x y*. $x = a --> P\ y$) (*iprover intro*: *cases*)+

```
  then show ?thesis by iprover
qed

lemmas trancl-induct [induct set: trancl] = tranclp-induct [to-set]

lemmas tranclp-induct2 =
  tranclp-induct [of - (ax,ay) (bx,by), split-rule,
    consumes 1, case-names base step]

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

lemma tranclp-trans-induct:
  assumes major: r^++ x y
    and cases: !!x y. r x y ==> P x y
      !!x y z. [| r^++ x y; P x y; r^++ y z; P y z |] ==> P x z
  shows P x y
  — Another induction rule for trancl, incorporating transitivity
  by (iprover intro: major [THEN tranclp-induct] cases)

lemmas trancl-trans-induct = tranclp-trans-induct [to-set]

lemma tranclE [cases set: trancl]:
  assumes (a, b) : r^+
  obtains
    (base) (a, b) : r
  | (step) c where (a, c) : r^+ and (c, b) : r
  using assms by cases simp-all

lemma trancl-Int-subset: [| r ⊆ s; r O (r^+ ∩ s) ⊆ s|] ==> r^+ ⊆ s
  apply (rule subsetI)
  apply (rule-tac p = x in PairE)
  apply clarify
  apply (erule trancl-induct)
   apply auto
  done

lemma trancl-unfold: r^+ = r Un r O r^+
  by (auto intro: trancl-into-trancl elim: tranclE)
```

Transitivity of r^+

```
lemma trans-trancl [simp]: trans (r^+)
proof (rule transI)
  fix x y z
  assume (x, y) ∈ r^+
  assume (y, z) ∈ r^+
  then show (x, z) ∈ r^+
  proof induct
```

```
    case (base u)
    from ‹(x, y) ∈ r^+› and ‹(y, u) ∈ r›
    show (x, u) ∈ r^+ ..
  next
    case (step u v)
    from ‹(x, u) ∈ r^+› and ‹(u, v) ∈ r›
    show (x, v) ∈ r^+ ..
  qed
qed

lemmas trancl-trans = trans-trancl [THEN transD, standard]

lemma tranclp-trans:
  assumes xy: r^++ x y
  and yz: r^++ y z
  shows r^++ x z using yz xy
  by induct iprover+

lemma trancl-id [simp]: trans r ⟹ r^+ = r
  apply auto
  apply (erule trancl-induct)
   apply assumption
  apply (unfold trans-def)
  apply blast
  done

lemma rtranclp-tranclp-tranclp:
  assumes r^** x y
  shows !!z. r^++ y z ==> r^++ x z using assms
  by induct (iprover intro: tranclp-trans)+

lemmas rtrancl-trancl-trancl = rtranclp-tranclp-tranclp [to-set]

lemma tranclp-into-tranclp2: r a b ==> r^++ b c ==> r^++ 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)^+ = r^+ ∪ {(a, b). (a, y) ∈ r^* ∧ (x, b) ∈ r^*}
  — 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*: $(r\hat{}{+}{+})\hat{}{-}{-}1\ x\ y \Longrightarrow (r\hat{}{-}{-}1)\hat{}{+}{+}\ x\ y$
apply (*drule conversepD*)
apply (*erule tranclp-induct*)
apply (*iprover intro*: *conversepI tranclp-trans*)+
done

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

lemma *tranclp-converseD*: $(r\hat{}{-}{-}1)\hat{}{+}{+}\ x\ y \Longrightarrow (r\hat{}{+}{+})\hat{}{-}{-}1\ x\ y$
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* $\Longrightarrow$ *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*: $!!y.\ r\ y\ b \Longrightarrow P(y)$
$!!y\ z.[|\ r\ y\ z;\ \ r\hat{}{+}{+}\ z\ b;\ \ P(z)\ |] \Longrightarrow P(y)$
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*: $R\hat{}{+}{+}\ x\ y \Longrightarrow EX\ z.\ R\ x\ z \wedge R\hat{}{*}{*}\ z\ y$
apply (*erule converse-tranclp-induct*)
apply *auto*
apply (*blast intro*: *rtranclp-trans*)
done

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *tranclD2*:

$(x,\ y) \in R^+ \Longrightarrow \exists\, z.\ (x,\ z) \in R^* \wedge (z,\ y) \in R$
by (*blast elim*: *tranclE intro*: *trancl-into-rtrancl*)

lemma *irrefl-tranclI*: $r\hat{}-1 \cap r\hat{}* = \{\} ==> (x,\ x) \notin r\hat{}+$
by (*blast elim*: *tranclE dest*: *trancl-into-rtrancl*)

lemma *irrefl-trancl-rD*: $!!X.\ ALL\ x.\ (x,\ x) \notin r\hat{}+ ==> (x,\ y) \in r ==> x \neq y$
by (*blast dest*: *r-into-trancl*)

lemma *trancl-subset-Sigma-aux*:
$(a,\ b) \in r\hat{}* ==> r \subseteq A \times A ==> a = b \vee a \in A$
by (*induct rule*: *rtrancl-induct*) *auto*

lemma *trancl-subset-Sigma*: $r \subseteq A \times A ==> r\hat{}+ \subseteq A \times A$
apply (*rule subsetI*)
apply (*simp only*: *split-tupled-all*)
apply (*erule tranclE*)
apply (*blast dest*!: *trancl-into-rtrancl trancl-subset-Sigma-aux*)+
done

lemma *reflcl-tranclp* [*simp*]: $(r\hat{}++)\hat{}== = r\hat{}**$
apply (*safe intro*!: *order-antisym*)
apply (*erule tranclp-into-rtranclp*)
apply (*blast elim*: *rtranclp.cases dest*: *rtranclp-into-tranclp1*)
done

lemmas *reflcl-trancl* [*simp*] = *reflcl-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r\hat{}=)\hat{}+ = r\hat{}*$
apply *safe*
apply (*drule trancl-into-rtrancl*, *simp*)
apply (*erule rtranclE*, *safe*)
apply (*rule r-into-trancl*, *simp*)
apply (*rule rtrancl-into-trancl1*)
apply (*erule rtrancl-reflcl* [*THEN equalityD2*, *THEN subsetD*], *fast*)
done

lemma *trancl-empty* [*simp*]: $\{\}\hat{}+ = \{\}$
by (*auto elim*: *trancl-induct*)

lemma *rtrancl-empty* [*simp*]: $\{\}\hat{}* = Id$
by (*rule subst* [*OF reflcl-trancl*]) *simp*

lemma *rtranclpD*: $R\hat{}**\ a\ b ==> a = b \vee a \neq b \wedge R\hat{}++\ a\ b$
by (*force simp add*: *reflcl-tranclp* [*symmetric*] *simp del*: *reflcl-tranclp*)

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*:

$(x,y) \in R^* = (x{=}y \vee x{\neq}y \wedge (x,y) \in R^+)$
by (*fast elim*: *trancl-into-rtrancl dest*: *rtranclD*)

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: *Domain* $(R\hat{}*) = UNIV$
by *blast*

lemma *Range-rtrancl* [*simp*]: *Range* $(R\hat{}*) = UNIV$
by *blast*

lemma *rtrancl-Un-subset*: $(R\hat{}* \cup S\hat{}*) \subseteq (R\ Un\ S)\hat{}*$
by (*rule rtrancl-Un-rtrancl* [*THEN subst*]) *fast*

lemma *in-rtrancl-UnI*: $x \in R\hat{}* \vee x \in S\hat{}* ==> x \in (R \cup S)\hat{}*$
by (*blast intro*: *subsetD* [*OF rtrancl-Un-subset*])

lemma *trancl-domain* [*simp*]: *Domain* $(r\hat{}+) = Domain\ r$
by (*unfold Domain-def*) (*blast dest*: *tranclD*)

lemma *trancl-range* [*simp*]: *Range* $(r\hat{}+) = Range\ r$
unfolding *Range-def* **by**(*simp add*: *trancl-converse* [*symmetric*])

lemma *Not-Domain-rtrancl*:
$x \sim$: *Domain* $R ==> ((x,\ y) : R\hat{}*) = (x = y)$
apply *auto*
apply (*erule rev-mp*)
apply (*erule rtrancl-induct*)
apply *auto*
done

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
$[\![$ *single-valued* r; $(x,y) \in r\hat{}*$; $(x,z) \in r\hat{}*]\!]$
$\Longrightarrow (y,z) \in r\hat{}* \vee (z,y) \in r\hat{}*$
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*: $(a,\ b) \in R ==> (b,\ c) \in R ==> (a,\ c) \in R\hat{}+$
by (*fast intro*: *trancl-trans*)

lemma *trancl-into-trancl* [*rule-format*]:
$(a,\ b) \in r^+ ==> (b,\ c) \in r --> (a,c) \in r^+$
apply (*erule trancl-induct*)
apply (*fast intro*: *r-r-into-trancl*)
apply (*fast intro*: *r-r-into-trancl trancl-trans*)

done

lemma *tranclp-rtranclp-tranclp*:
r^{++} *a b* ==> r^{**} *b c* ==> r^{++} *a c*
apply (*drule tranclpD*)
apply (*elim exE conjE*)
apply (*drule rtranclp-trans*, *assumption*)
apply (*drule rtranclp-into-tranclp2*, *assumption*, *assumption*)
done

lemmas *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl.trancl-into-trancl trancl-into-trancl2
rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

19.4 Setup of transitivity reasoner

ML ⟪

```
structure Trancl_Tac = Trancl_Tac_Fun (
  struct
    val r_into_trancl = @{thm trancl.r_into_trancl};
    val trancl_trans  = @{thm trancl_trans};
    val rtrancl_refl = @{thm rtrancl.rtrancl_refl};
    val r_into_rtrancl = @{thm r_into_rtrancl};
    val trancl_into_rtrancl = @{thm trancl_into_rtrancl};
    val rtrancl_trancl_trancl = @{thm rtrancl_trancl_trancl};
    val trancl_rtrancl_trancl = @{thm trancl_rtrancl_trancl};
    val rtrancl_trans = @{thm rtrancl_trans};

  fun decomp (Trueprop $ t) =
    let fun dec (Const (op :, _) $ (Const (Pair, _) $ a $ b) $ rel ) =
        let fun decr (Const (Transitive_Closure.rtrancl, _ ) $ r) = (r,r*)
              | decr (Const (Transitive_Closure.trancl, _ ) $ r)  = (r,r+)
              | decr r = (r,r);
            val (rel,r) = decr (Envir.beta_eta_contract rel);
        in SOME (a,b,rel,r) end
      | dec _ =  NONE
```

```
    in dec t end;

  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, rel, r) end
      | dec - =  NONE
    in dec t end;

  end);
⟫

declaration ⟪ fn - =>
  Simplifier.map-ss (fn ss => ss
    addSolver (mk-solver Trancl (fn - => Trancl-Tac.trancl-tac))
    addSolver (mk-solver Rtrancl (fn - => Trancl-Tac.rtrancl-tac))
    addSolver (mk-solver Tranclp (fn - => Tranclp-Tac.trancl-tac))
    addSolver (mk-solver Rtranclp (fn - => Tranclp-Tac.rtrancl-tac)))
⟫



method-setup trancl =
  ⟪ 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

20 Finite-Set: Finite sets

theory *Finite-Set*
imports *Divides Transitive-Closure*
begin

20.1 Definition and basic properties

inductive *finite* :: *'a set* => *bool*
  where
    emptyI [*simp*, *intro*!]: *finite* {}
  | *insertI* [*simp*, *intro*!]: *finite A* ==> *finite* (*insert a A*)

lemma *ex-new-if-finite*: — does not depend on def of finite at all
  assumes ¬ *finite* (*UNIV* :: *'a set*) **and** *finite A*
  shows ∃ *a*::*'a*. *a* ∉ *A*
proof −
  from *prems* **have** *A* ≠ *UNIV* **by** *blast*
  thus *?thesis* **by** *blast*
qed

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
  finite F ==>
    P {} ==> (!!*x F*. *finite F* ==> *x* ∉ *F* ==> *P F* ==> *P* (*insert x F*)) ==>
P F
  — Discharging *x* ∉ *F* entails extra work.
proof −
  assume *P* {} **and**
    insert: !!*x F*. *finite F* ==> *x* ∉ *F* ==> *P F* ==> *P* (*insert x F*)
  assume *finite F*
  thus *P F*
  proof *induct*
    show *P* {} **by** *fact*
    fix *x F* **assume** *F*: *finite F* **and** *P*: *P F*
    show *P* (*insert x F*)
    proof *cases*
      assume *x* ∈ *F*
      hence *insert x F* = *F* **by** (*rule insert-absorb*)
      with *P* **show** *?thesis* **by** (*simp only*:)
    next
      assume *x* ∉ *F*
      from *F this P* **show** *?thesis* **by** (*rule insert*)
    qed
  qed

qed

lemma *finite-ne-induct*[*case-names singleton insert*, *consumes 2*]:
assumes *fin*: *finite F* **shows** $F \neq \{\} \Longrightarrow$
$[\![\bigwedge x.\ P\{x\};$
$\quad \bigwedge x\ F.\ [\![\textit{finite } F;\ F \neq \{\};\ x \notin F;\ P\ F\ ]\!] \Longrightarrow P\ (\textit{insert } x\ F)\ ]\!]$
$\Longrightarrow P\ F$
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 $F \neq \{\}$
thus *?thesis* **using** *insert* **by** *blast*
qed
qed

lemma *finite-subset-induct* [*consumes 2*, *case-names empty insert*]:
assumes *finite F* **and** $F \subseteq A$
and *empty*: $P\ \{\}$
and *insert*: !!*a F*. *finite F* ==> $a \in A$ ==> $a \notin F$ ==> $P\ F$ ==> P (*insert a F*)
shows $P\ F$
proof −
from ‹*finite F*› **and** ‹$F \subseteq A$›
show *?thesis*
proof *induct*
show $P\ \{\}$ **by** *fact*
next
fix *x F*
assume *finite F* **and** $x \notin F$ **and**
P: $F \subseteq A$ ==> $P\ F$ **and** *i*: *insert x F* $\subseteq A$
show P (*insert x F*)
proof (*rule insert*)
from *i* **show** $x \in A$ **by** *blast*
from *i* **have** $F \subseteq A$ **by** *blast*
with *P* **show** $P\ F$ **.**
show *finite F* **by** *fact*
show $x \notin F$ **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*} ⟹ *finite A*
proof (*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
let *?B* = *f* ' {*i*. *i* < *n*}
have *finB*: *finite ?B* **by**(*rule Suc.hyps*[*OF refl*])
show *?case*
proof *cases*
assume ∃*k*<*n*. *f n* = *f k*
hence *A* = *?B* **using** *Suc.prems* **by**(*auto simp*:*less-Suc-eq*)
thus *?thesis* **using** *finB* **by** *simp*
next
assume ¬(∃ *k*<*n*. *f n* = *f k*)
hence *A* = *insert* (*f n*) *?B* **using** *Suc.prems* **by**(*auto simp*:*less-Suc-eq*)
thus *?thesis* **using** *finB* **by** *simp*
qed
qed

lemma *finite-conv-nat-seg-image*:
finite A = (∃ (*n*::*nat*) *f*. *A* = *f* ' {*i*::*nat*. *i*<*n*})
by(*blast intro*: *nat-seg-image-imp-finite dest*: *finite-imp-nat-seg-image-inj-on*)

20.1.1 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 ∈ y ==> P y ==> P (y -
{x})
      have c ⊆ b ==> P (b - c)
        using c
      proof induct
        case empty
        from P1 show ?case by simp
      next
        case (insert x F)
        have P (b - F - {x})
        proof (rule P2)
          from - b show finite (b - F) by (rule finite-subset) blast
          from insert show x ∈ b - F by simp
          from insert show P (b - F) by simp
        qed
       also have b - F - {x} = b - insert x F by (rule Diff-insert [symmetric])
        finally show ?case .
      qed
    }
    then show ?thesis by this (simp-all add: assms)
  qed
  then show ?thesis by simp
qed

lemma finite-Diff [simp]: finite 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*} ⊆ {*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 $(\forall x{\in}A.\ \mathit{finite}\ (B\ x)) \wedge \mathit{finite}\ \{x \in A.\ B\ x \neq \{\}\}$?
We'd need to prove $\mathit{finite}\ C \Longrightarrow \forall A\ B.\ \mathit{UNION}\ A\ B \subseteq C \longrightarrow \mathit{finite}\ \{x \in A.\ B\ x \neq \{\}\}$ 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* $\exists k.\ k<n\ \&\ f\ k = (x,y)$)
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

20.2 Class *finite*

setup ⟪ *Sign.add-path finite* ⟫ — FIXME: name tweaking
class *finite* = *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 subset-UNIV finite-UNIV finite-subset*)+

lemma *UNIV-unit* [*noatp*]:
UNIV = {()} **by** *auto*

instance *unit* :: *finite*
by *default* (*simp add*: *UNIV-unit*)

lemma *UNIV-bool* [*noatp*]:
UNIV = {*False*, *True*} **by** *auto*

instance *bool* :: *finite*
by *default* (*simp add*: *UNIV-bool*)

instance * :: (*finite*, *finite*) *finite*
by *default* (*simp only*: *UNIV-Times-UNIV* [*symmetric*] *finite-cartesian-product finite*)

instance + :: (*finite*, *finite*) *finite*
by *default* (*simp only*: *UNIV-Plus-UNIV* [*symmetric*] *finite-Plus finite*)

lemma *inj-graph*: *inj* (%*f*. {(*x*, *y*). *y* = *f x*})
by (*rule inj-onI*, *auto simp add*: *expand-set-eq expand-fun-eq*)

instance *fun* :: (*finite*, *finite*) *finite*
proof
  show *finite* (*UNIV* :: ('*a* => '*b*) *set*)
  proof (*rule finite-imageD*)
    let *?graph* = %*f*::'*a* => '*b*. {(*x*, *y*). *y* = *f x*}
    have *range ?graph* ⊆ *Pow UNIV* **by** *simp*
    moreover have *finite* (*Pow* (*UNIV* :: ('*a* * '*b*) *set*))
      by (*simp only*: *finite-Pow-iff finite*)
    ultimately show *finite* (*range ?graph*)
      by (*rule finite-subset*)
    show *inj ?graph* **by** (*rule inj-graph*)
  qed
qed

20.3 A fold functional for finite sets

The intended behaviour is *fold f g z* {x_1, ..., x_n} = *f* (*g* x_1) (... (*f* (*g* x_n) *z*)...) 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* ∉ *A*; *foldSet f g z A y* ⟧
    ⟹ *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*

20.3.1 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 ∉ A;
inj-on h {i. i < Suc m}|]
==> A = h ‘ {i. i < m}
apply (*auto simp add*: *image-less-Suc inj-on-def*)
apply (*blast intro*: *less-trans*)
done

lemma *insert-inj-onE*:
assumes *aA*: *insert a A = h‘{i::nat. i<n}* **and** *anot*: *a ∉ A*
and *inj-on*: *inj-on h {i::nat. i<n}*
shows *∃hm m. inj-on hm {i::nat. i<m} & A = hm ‘ {i. i<m} & m < n*
proof (*cases n*)
case *0* **thus** *?thesis* **using** *aA* **by** *auto*
next
case (*Suc m*)
have *nSuc*: *n = Suc m* **by** *fact*
have *mlessn*: *m<n* **by** (*simp add*: *nSuc*)
from *aA* **obtain** *k* **where** *hkeq*: *h k = a* **and** *klessn*: *k<n* **by** (*blast elim*!: *equalityE*)
let *?hm = swap k m h*
have *inj-hm*: *inj-on ?hm {i. i < n}* **using** *klessn mlessn*
by (*simp add*: *inj-on-swap-iff inj-on*)
show *?thesis*
proof (*intro exI conjI*)
show *inj-on ?hm {i. i < m}* **using** *inj-hm*
by (*auto simp add*: *nSuc less-Suc-eq intro*: *subset-inj-on*)
show *m<n* **by** (*rule mlessn*)
show *A = ?hm ‘ {i. i < m}*
proof (*rule insert-image-inj-on-eq*)
show *inj-on* (*swap k m h*) *{i. i < Suc m}* **using** *inj-hm nSuc* **by** *simp*
show *?hm m ∉ A* **by** (*simp add*: *swap-def hkeq anot*)
show *insert* (*?hm m*) *A = ?hm ‘ {i. i < Suc m}*
using *aA hkeq nSuc klessn*
by (*auto simp add*: *swap-def image-less-Suc fun-upd-image*
less-Suc-eq inj-on-image-set-diff [*OF inj-on*])
qed
qed
qed

context *ab-semigroup-mult*

begin

lemma *foldSet-determ-aux*:
 !!$A\ x\ x'\ h.$ ⟦ $A = h`\{i::nat.\ i<n\}$; *inj-on h* $\{i.\ i<n\}$;
 foldSet times g z A x; *foldSet times g z A* x' ⟧
 ⟹ $x' = x$
proof (*induct n rule*: *less-induct*)
 case (*less n*)
 have *IH*: !!$m\ h\ A\ x\ x'.$
 ⟦$m<n$; $A = h\ `\ \{i.\ i<m\}$; *inj-on h* $\{i.\ i<m\}$;
 foldSet times g z A x; *foldSet times g z A* x'⟧ ⟹ $x' = x$ **by** *fact*
 have *Afoldx*: *foldSet times g z A x* **and** *Afoldx'*: *foldSet times g z A* x'
 and *A*: $A = h`\{i.\ i<n\}$ **and** *injh*: *inj-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*: $x = g\ b * u$
 and *notinB*: $b \notin B$ **and** *Bu*: *foldSet times 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'*: $x' = g\ c * v$
 and *notinC*: $c \notin C$ **and** *Cv*: *foldSet times 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 times g z ?D d
      using finite-imp-foldSet by iprover
    moreover have cinB: c ∈ B using B by auto
    ultimately have foldSet times g z B (g c * d)
      by(rule Diff1-foldSet)
    then have g c * d = u by (rule IH [OF lessB Beq inj-onB Bu])
    then have u = g c * d ..
    moreover have v = g b * d
    proof (rule sym, rule IH [OF lessC Ceq inj-onC Cv])
      show foldSet times g z C (g b * d) using C notinB Dfoldd
        by fastsimp
    qed
    ultimately show ?thesis using x x'
      by (simp add: mult-left-commute)
  qed
 qed
qed
qed
```

```
lemma foldSet-determ:
  foldSet times g z A x ==> foldSet times 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 fold-equality: foldSet times g z A y ==> fold times g z A = y
  by (unfold fold-def) (blast intro: foldSet-determ)
```

The base case for *fold*:

```
lemma (in -) fold-empty [simp]: fold f g z {} = z
  by (unfold fold-def) blast
```

```
lemma fold-insert-aux: x ∉ A ==>
    (foldSet times g z (insert x A) v) =
    (EX y. foldSet times g z A y & v = g x * y)
  apply auto
  apply (rule-tac A1 = A and f1 = times in finite-imp-foldSet [THEN exE])
   apply (fastsimp dest: foldSet-imp-finite)
  apply (blast intro: foldSet-determ)
  done
```

The recursion equation for *fold*:

```
lemma fold-insert [simp]:
    finite A ==> x ∉ A ==> fold times g z (insert x A) = g x * fold times 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 fold-rec:
assumes fin: finite A and a: a:A
shows fold times g z A = g a * fold times g z (A - {a})
proof-
  have A: A = insert a (A - {a}) using a by blast
  hence fold times g z A = fold times g z (insert a (A - {a})) by simp
  also have ... = g a * fold times g z (A - {a})
    by(rule fold-insert) (simp add:fin)+
  finally show ?thesis .
qed

end
```

A simplified version for idempotent functions:

```
context ab-semigroup-idem-mult
begin

lemma fold-insert-idem:
assumes finA: finite A
shows fold times g z (insert a A) = g a * fold times 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 times g z (insert a A) = fold times g z (insert a B) using A by simp
    also have ... = g a * fold times g z B
      using finB disj by simp
    also have ... = g a * fold times g z A
      using A finB disj
        by (simp add: mult-idem mult-assoc [symmetric])
    finally show ?thesis .
  qed
next
  assume a ∉ A
  with finA show ?thesis by simp
qed

lemma foldI-conv-id:
  finite A ⟹ fold times g z A = fold times id z (g ' A)
by(erule finite-induct)(simp-all add: fold-insert-idem del: fold-insert)
```

end

20.3.2 Lemmas about *fold*

context *ab-semigroup-mult*
begin

lemma *fold-commute*:
*finite A ==> (!!z. x * (fold times g z A) = fold times g (x * z) A)*
apply (*induct set*: *finite*)
apply *simp*
apply (*simp add*: *mult-left-commute* [*of x*])
done

lemma *fold-nest-Un-Int*:
finite A ==> finite B
==> fold times g (fold times g z B) A = fold times g (fold times 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 *fold-nest-Un-disjoint*:
finite A ==> finite B ==> A Int B = {}
==> fold times g z (A Un B) = fold times g (fold times g z B) A
by (*simp add*: *fold-nest-Un-Int*)

lemma *fold-reindex*:
assumes *fin*: *finite A*
shows *inj-on h A ⟹ fold times g z (h ' A) = fold times (g ∘ h) z A*
using *fin* **apply** *induct*
apply *simp*
apply *simp*
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 *fold-fusion*:
includes *ab-semigroup-mult g*
assumes *fin*: *finite A*
and *hyp*: ⋀*x y. h (g x y) = times x (h y)*
shows *h (fold g j w A) = fold times j (h w) A*
using *fin hyp* **by** (*induct set*: *finite*) *simp-all*

lemma *fold-cong*:
finite A ⟹ (!!x. x:A ==> g x = h x) ==> fold times g z A = fold times h z A
apply (*subgoal-tac ALL C. C <= A --> (ALL x:C. g x = h x) --> fold times g z C = fold times h z C*)

```
 apply simp
apply (erule finite-induct, simp)
apply (simp add: subset-insert-iff, clarify)
apply (subgoal-tac finite C)
 prefer 2 apply (blast dest: finite-subset [COMP swap-prems-rl])
apply (subgoal-tac C = insert x (C - {x}))
 prefer 2 apply blast
apply (erule ssubst)
apply (drule spec)
apply (erule (1) notE impE)
apply (simp add: Ball-def del: insert-Diff-single)
done

end

context comm-monoid-mult
begin

lemma fold-Un-Int:
  finite A ==> finite B ==>
    fold times g 1 A * fold times g 1 B =
    fold times g 1 (A Un B) * fold times g 1 (A Int B)
  by (induct set: finite)
    (auto simp add: mult-ac insert-absorb Int-insert-left)

corollary fold-Un-disjoint:
  finite A ==> finite B ==> A Int B = {} ==>
    fold times g 1 (A Un B) = fold times g 1 A * fold times g 1 B
  by (simp add: fold-Un-Int)

lemma fold-UN-disjoint:
  [| finite I; ALL i:I. finite (A i);
     ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {} |]
  ==> fold times g 1 (UNION I A) =
      fold times (%i. fold times g 1 (A i)) 1 I
  apply (induct set: finite, simp, atomize)
  apply (subgoal-tac ALL i:F. x ≠ i)
   prefer 2 apply blast
  apply (subgoal-tac A x Int UNION F A = {})
   prefer 2 apply blast
  apply (simp add: fold-Un-disjoint)
  done

lemma fold-Sigma: finite A ==> ALL x:A. finite (B x) ==>
  fold times (%x. fold times (g x) 1 (B x)) 1 A =
  fold times (split g) 1 (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 *fold-distrib*: *finite A* $\Longrightarrow$
fold times (%*x*. *g x* $*$ *h x*) *1 A* = *fold times g 1 A* $*$ *fold times h 1 A*
by (*erule finite-induct*) (*simp-all add*: *mult-ac*)

end

20.4 Generalized summation over a set

interpretation *comm-monoid-add*: *comm-monoid-mult* [*0*::'*a*::*comm-monoid-add op* +]
by *unfold-locales* (*auto intro*: *add-assoc add-commute*)

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 ($\sum$ - [*1000*] *999*) **where**
$\sum A$ == *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*
$\sum i{\in}A.\ b$ == *setsum* (%*i*. *b*) *A*

Instead of $\sum x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\sum x|P.\ e$.

syntax
-qsetsum :: *pttrn* ⇒ *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}
∑ 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 comm-monoid-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: comm-monoid-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: comm-monoid-add.fold-cong*)

lemma *setsum-cong2*: ⟦⋀x. x ∈ A ⟹ f x = g x⟧ ⟹ *setsum f A = setsum g A*
by (*rule setsum-cong[OF refl], auto*)

lemma *setsum-reindex-cong:*
[|inj-on f A; B = f ' A; !!a. a:A ⟹ g a = h (f a)|]
==> setsum h B = setsum g A

by (*simp add*: *setsum-reindex cong*: *setsum-cong*)

lemma *setsum-0*[*simp*]: *setsum* (%*i*. *0*) *A* = *0*
apply (*clarsimp simp*: *setsum-def*)
apply (*erule finite-induct*, *auto*)
done

lemma *setsum-0'*: *ALL a*:*A*. *f a* = *0* ==> *setsum f A* = *0*
by(*simp add*:*setsum-cong*)

lemma *setsum-Un-Int*: *finite A* ==> *finite B* ==>
setsum g (*A Un B*) + *setsum g* (*A Int B*) = *setsum g A* + *setsum g B*
— The reversed orientation looks more natural, but LOOPS as a simprule!
by(*simp add*: *setsum-def comm-monoid-add.fold-Un-Int* [*symmetric*])

lemma *setsum-Un-disjoint*: *finite A* ==> *finite B*
==> *A Int B* = {} ==> *setsum g* (*A Un B*) = *setsum g A* + *setsum g B*
by (*subst setsum-Un-Int* [*symmetric*], *auto*)

lemma *setsum-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. *i* ≠ *j* --> *A i Int A j* = {}) ==>
setsum f (*UNION I A*) = (∑ *i*∈*I*. *setsum f* (*A i*))
by(*simp add*: *setsum-def comm-monoid-add.fold-UN-disjoint cong*: *setsum-cong*)

No need to assume that C is finite. If infinite, the rhs is directly 0, and $\bigcup C$ is also infinite, hence the lhs is also 0.

lemma *setsum-Union-disjoint*:
[| (*ALL A*:*C*. *finite A*);
(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* --> *A Int B* = {}) |]
==> *setsum f* (*Union C*) = *setsum* (*setsum f*) *C*
apply (*cases finite C*)
prefer *2* **apply** (*force dest*: *finite-UnionD simp add*: *setsum-def*)
apply (*frule setsum-UN-disjoint* [*of C id f*])
apply (*unfold Union-def id-def*, *assumption*+)
done

lemma *setsum-Sigma*: *finite A* ==> *ALL x*:*A*. *finite* (*B x*) ==>
(∑ *x*∈*A*. (∑ *y*∈*B x*. *f x y*)) = (∑ (*x*,*y*)∈(*SIGMA x*:*A*. *B x*). *f x y*)
by(*simp add*:*setsum-def comm-monoid-add.fold-Sigma split-def cong*:*setsum-cong*)

Here we can eliminate the finiteness assumptions, by cases.

lemma *setsum-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*: *setsum-Sigma*)

apply (*cases A={}*, *simp*)
apply (*simp*)
apply (*auto simp add*: *setsum-def*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *setsum-addf*: *setsum* (%*x*. *f x* + *g x*) *A* = (*setsum f A* + *setsum g A*)
by(*simp add*:*setsum-def comm-monoid-add.fold-distrib*)

20.4.1 Properties in more restricted classes of structures

lemma *setsum-SucD*: *setsum f A* = *Suc n* ==> *EX a*:*A*. *0* < *f a*
apply (*case-tac finite A*)
prefer *2* **apply** (*simp add*: *setsum-def*)
apply (*erule rev-mp*)
apply (*erule finite-induct*, *auto*)
done

lemma *setsum-eq-0-iff* [*simp*]:
finite F ==> (*setsum f F* = *0*) = (*ALL a*:*F*. *f a* = (*0*::*nat*))
by (*induct set*: *finite*) *auto*

lemma *setsum-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-group-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* ⟹
(*setsum f* (*A* − {*a*}) :: (*'a*::*ab-group-add*)) =
(*if a*:*A then setsum f A* − *f a else setsum f A*)
by (*erule finite-induct*) (*auto simp add*: *insert-Diff-if*)

lemma *setsum-diff1'*[*rule-format*]: *finite A* ⟹ *a* ∈ *A* ⟶ (∑ *x* ∈ *A*. *f x*) = *f a* + (∑ *x* ∈ (*A* − {*a*}). *f x*)
apply (*erule finite-induct*[**where** *F*=*A* **and** *P*=% *A*. (*a* ∈ *A* ⟶ (∑ *x* ∈ *A*. *f*

$x) = f\ a + (\sum\ x \in (A - \{a\}).\ f\ x))]$)
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* $(A -$ *insert x F*$)$ = *setsum f A* − *setsum f* (*insert x F*)
by *simp*
thus *setsum f* $(A -$ *insert x F*$)$ = *setsum f A* − *setsum f* (*insert x F*) **by** *simp*
qed

lemma *setsum-diff*:
assumes *le*: *finite A B* $\subseteq A$
shows *setsum f* $(A - B)$ = *setsum f A* − ((*setsum f B*)::($'a$::*ab-group-add*))
proof −
from *le* **have** *finiteB*: *finite B* **using** *finite-subset* **by** *auto*
show *?thesis* **using** *finiteB le*
proof *induct*
case *empty*
thus *?case* **by** *auto*
next
case (*insert x F*)
thus *?case* **using** *le finiteB*
by (*simp add*: *Diff-insert*[**where** *a*=*x* **and** *B*=*F*] *setsum-diff1 insert-absorb*)
qed
qed

lemma *setsum-mono*:
assumes *le*: $\bigwedge i.\ i{\in}K \Longrightarrow f\ (i{::}'a) \leq ((g\ i){::}('b{::}\{\mathit{comm\text{-}monoid\text{-}add},\ \mathit{pordered\text{-}ab\text{-}semigroup\text{-}add}\}))$
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 $f :: 'a \Rightarrow 'b{::}\{\mathit{pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add}, \mathit{comm\text{-}monoid\text{-}add}\}$
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 $(\%x.\ -\ (f\ x){::}'a{::}\mathit{ab\text{-}group\text{-}add})\ A = -$ *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 $(\%x.\ ((f\ x){::}'a{::}\mathit{ab\text{-}group\text{-}add})\ -\ g\ x)\ A =$
setsum f A $-$ *setsum g A*
proof (*cases finite A*)
case *True* **thus** *?thesis* **by** (*simp add*: *diff-minus setsum-addf setsum-negf*)
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-nonneg*:

assumes *nn*: $\forall x{\in}A.\ (0{::}'a{::}\{$*pordered-ab-semigroup-add*,*comm-monoid-add*$\}) \leq f\ x$
shows $0 \leq$ *setsum f A*
proof (*cases finite A*)
case *True* **thus** *?thesis* **using** *nn*
proof *induct*
case *empty* **then show** *?case* **by** *simp*
next
case (*insert x F*)
then have $0 + 0 \leq f\ x +$ *setsum f F* **by** (*blast intro*: *add-mono*)
with *insert* **show** *?case* **by** *simp*
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-nonpos*:
assumes *np*: $\forall x{\in}A.\ f\ x \leq (0{::}'a{::}\{$*pordered-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* $\leq 0 + 0$ **by** (*blast intro*: *add-mono*)
with *insert* **show** *?case* **by** *simp*
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-mono2*:
fixes $f :: 'a \Rightarrow 'b ::$ {*pordered-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 $\ldots =$ *setsum f* $(A \cup (B{-}A))$ **using** *fin finite-subset*[*OF sub fin*]
by (*simp add*:*setsum-Un-disjoint del*:*Un-Diff-cancel*)
also have $A \cup (B{-}A) = B$ **using** *sub* **by** *blast*
finally show *?thesis* **.**
qed

lemma *setsum-mono3*: *finite B ==> A <= B ==>*
ALL x: *B − A.*
0 <= ((*f x*)::*'a*::{*comm-monoid-add*,*pordered-ab-semigroup-add*}) *==>*
setsum f A <= setsum f B

```
  apply (subgoal-tac setsum f B = setsum f A + setsum f (B - A))
  apply (erule ssubst)
  apply (subgoal-tac setsum f A + 0 <= setsum f A + setsum f (B - A))
  apply simp
  apply (rule add-left-mono)
  apply (erule setsum-nonneg)
  apply (subst setsum-Un-disjoint [THEN sym])
  apply (erule finite-subset, assumption)
  apply (rule finite-subset)
  prefer 2
  apply assumption
  apply auto
  apply (rule setsum-cong)
  apply auto
done

lemma setsum-right-distrib:
  fixes f :: 'a => ('b::semiring-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 ... = (∑ (y,x)∈B × A. f x y)
    (is - = ?t)
    apply (simp add: swap-product)
    done
  finally show ?s = ?t .
qed

lemma setsum-product:
  fixes f :: 'a => ('b::semiring-0)
  shows setsum f A * setsum g B = (∑ i∈A. ∑ j∈B. f i * g j)
  by (simp add: setsum-right-distrib setsum-left-distrib) (rule setsum-commute)
```

20.5 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*$\prod$-∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*$\prod$-∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
PROD i:*A*. *b* == *setprod* (%*i*. *b*) *A*
$\prod$ *i*∈*A*. *b* == *setprod* (%*i*. *b*) *A*

Instead of $\prod x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\prod x|P.\ e$.

syntax
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3PROD* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*$\prod$ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*$\prod$ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations
PROD x|*P*. *t* => *setprod* (%*x*. *t*) {*x*. *P*}
$\prod x|P.\ t$ => *setprod* (%*x*. *t*) {*x*. *P*}

lemma *setprod-empty* [*simp*]: *setprod f* {} = *1*
by (*auto simp add*: *setprod-def*)

lemma *setprod-insert* [*simp*]: [| *finite A*; *a* ∉ *A* |] ==>
setprod f (*insert a A*) = *f a* ∗ *setprod f A*
by (*simp add*: *setprod-def*)

lemma *setprod-infinite* [*simp*]: ~ *finite A* ==> *setprod f A* = *1*
by (*simp add*: *setprod-def*)

lemma *setprod-reindex*:
inj-on f B ==> *setprod h* (*f* ‘ *B*) = *setprod* (*h* ∘ *f*) *B*
by(*auto simp*: *setprod-def fold-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*:

$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow f\ x = g\ x) \Longrightarrow setprod\ f\ A = setprod\ g\ B$
by(*fastsimp simp*: *setprod-def intro*: *fold-cong*)

lemma *strong-setprod-cong*:
$A = B \Longrightarrow (!!x.\ x{:}B =simp=> f\ x = g\ x) \Longrightarrow setprod\ f\ A = setprod\ g\ B$
by(*fastsimp simp*: *simp-implies-def setprod-def intro*: *fold-cong*)

lemma *setprod-reindex-cong*: *inj-on f A* ==>
$B = f\ `\ A \Longrightarrow g = h \circ f \Longrightarrow setprod\ h\ B = setprod\ g\ A$
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'*: $ALL\ a{:}F.\ f\ a = 1 \Longrightarrow setprod\ f\ F = 1$
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*
$\Longrightarrow setprod\ g\ (A\ Un\ B) * setprod\ g\ (A\ Int\ B) = setprod\ g\ A * setprod\ g\ B$
by(*simp add*: *setprod-def fold-Un-Int*[*symmetric*])

lemma *setprod-Un-disjoint*: *finite A* ==> *finite B*
$\Longrightarrow A\ Int\ B = \{\} \Longrightarrow setprod\ g\ (A\ Un\ B) = setprod\ g\ A * setprod\ g\ B$
by (*subst setprod-Un-Int* [*symmetric*], *auto*)

lemma *setprod-UN-disjoint*:
finite I ==> $(ALL\ i{:}I.\ finite\ (A\ i)) \Longrightarrow$
$(ALL\ i{:}I.\ ALL\ j{:}I.\ i \neq j \longrightarrow A\ i\ Int\ A\ j = \{\}) \Longrightarrow$
$setprod\ f\ (UNION\ I\ A) = setprod\ (\%i.\ setprod\ f\ (A\ i))\ I$
by(*simp add*: *setprod-def fold-UN-disjoint cong*: *setprod-cong*)

lemma *setprod-Union-disjoint*:
[| $(ALL\ A{:}C.\ finite\ A)$;
$(ALL\ A{:}C.\ ALL\ B{:}C.\ A \neq B \longrightarrow A\ Int\ B = \{\})$ |]
$\Longrightarrow setprod\ f\ (Union\ C) = setprod\ (setprod\ f)\ C$
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* ==> $ALL\ x{:}A.\ finite\ (B\ x) \Longrightarrow$
$(\prod x{\in}A.\ (\prod y{\in}\ B\ x.\ f\ x\ y)) =$

$(\prod (x,y)\in(SIGMA\ x{:}A.\ B\ x).\ f\ x\ y)$
by(*simp add*:*setprod-def fold-Sigma split-def cong*:*setprod-cong*)

Here we can eliminate the finiteness assumptions, by cases.

lemma *setprod-cartesian-product*:
$(\prod x\in A.\ (\prod y\in B.\ f\ x\ y)) = (\prod (x,y)\in(A <*> B).\ f\ x\ y)$
apply (*cases finite A*)
apply (*cases finite B*)
apply (*simp add*: *setprod-Sigma*)
apply (*cases A={}*, *simp*)
apply (*simp add*: *setprod-1*)
apply (*auto simp add*: *setprod-def*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *setprod-timesf*:
setprod (%*x*. *f x* * *g x*) *A* = (*setprod f A* * *setprod g A*)
by(*simp add*:*setprod-def fold-distrib*)

20.5.1 Properties in more restricted classes of structures

lemma *setprod-eq-1-iff* [*simp*]:
finite F ==> (*setprod f F* = *1*) = (*ALL a*:*F*. *f a* = (*1*::*nat*))
by (*induct set*: *finite*) *auto*

lemma *setprod-zero*:
finite A ==> *EX x*: *A*. *f x* = (*0*::'*a*::*comm-semiring-1*) ==> *setprod f A* = *0*
apply (*induct set*: *finite*, *force*, *clarsimp*)
apply (*erule disjE*, *auto*)
done

lemma *setprod-nonneg* [*rule-format*]:
(*ALL x*: *A*. (*0*::'*a*::*ordered-idom*) ≤ *f x*) --> *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-mono*, *assumption*+)
apply (*auto simp add*: *setprod-def*)
done

lemma *setprod-pos* [*rule-format*]: (*ALL x*: *A*. (*0*::'*a*::*ordered-idom*) < *f x*)
--> *0* < *setprod f A*
apply (*case-tac finite A*)
apply (*induct set*: *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-semiring-1*) * *y* = *0* --> *x* = *0* | *y* = *0*) ==>
finite A ==> (*ALL x*: *A. f x* ≠ (*0*::'*a*)) --> *setprod f A* ≠ *0*
apply (*erule finite-induct*, *auto*)
done

lemma *setprod-zero-eq*:
(*ALL x y.* (*x*::'*a*::*comm-semiring-1*) * *y* = *0* --> *x* = *0* | *y* = *0*) ==>
finite A ==> (*setprod f A* = (*0*::'*a*)) = (*EX x*: *A. f x* = *0*)
apply (*insert setprod-zero* [*of A f*] *setprod-nonzero* [*of A f*], *blast*)
done

lemma *setprod-nonzero-field*:
finite A ==> (*ALL x*: *A. f x* ≠ (*0*::'*a*::*idom*)) ==> *setprod f A* ≠ *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* ≠ *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* ≠ *0* ==>
(*setprod f* (*A* − {*a*}) :: '*a* :: {*field*}) =
(*if a*:*A then setprod f A* / *f a else setprod f A*)
by (*erule finite-induct*) (*auto simp add*: *insert-Diff-if*)

lemma *setprod-inversef*: *finite A* ==>
ALL x: *A. f x* ≠ (*0*::'*a*::{*field*,*division-by-zero*}) ==>
setprod (*inverse* ∘ *f*) *A* = *inverse* (*setprod f A*)
apply (*erule finite-induct*)
apply (*simp*, *simp*)
done

lemma *setprod-dividef*:
[|*finite A*;
∀ *x* ∈ *A. g x* ≠ (*0*::'*a*::{*field*,*division-by-zero*})|]
==> *setprod* (%*x. f x* / *g x*) *A* = *setprod f A* / *setprod g A*
apply (*subgoal-tac*
setprod (%*x. f x* / *g x*) *A* = *setprod* (%*x. f x* * (*inverse* ∘ *g*) *x*) *A*)

apply (*erule ssubst*)
apply (*subst divide-inverse*)
apply (*subst setprod-timesf*)
apply (*subst setprod-inversef*, *assumption+*, *rule refl*)
apply (*rule setprod-cong*, *rule refl*)
apply (*subst divide-inverse*, *auto*)
done

20.6 Finite cardinality

This definition, although traditional, is ugly to work with: *card A == LEAST n. EX f. A = {f i | i. i < n}*. But now that we have *setsum* things are easy:

definition
card :: *'a set* ⇒ *nat*
where
card A = *setsum* (λ*x*. *1*) *A*

lemma *card-empty* [*simp*]: *card* {} = *0*
by (*simp add*: *card-def*)

lemma *card-infinite* [*simp*]: ~ *finite A* ==> *card A* = *0*
by (*simp add*: *card-def*)

lemma *card-eq-setsum*: *card A* = *setsum* (%*x*. *1*) *A*
by (*simp add*: *card-def*)

lemma *card-insert-disjoint* [*simp*]:
finite A ==> *x* ∉ *A* ==> *card* (*insert x A*) = *Suc*(*card A*)
by(*simp add*: *card-def*)

lemma *card-insert-if*:
finite A ==> *card* (*insert x A*) = (*if x*:*A then card A else Suc*(*card*(*A*)))
by (*simp add*: *insert-absorb*)

lemma *card-0-eq* [*simp*,*noatp*]: *finite A* ==> (*card A* = *0*) = (*A* = {})
apply *auto*
apply (*drule-tac a* = *x* **in** *mk-disjoint-insert*, *clarify*, *auto*)
done

lemma *card-eq-0-iff*: (*card A* = *0*) = (*A* = {} | ~ *finite A*)
by *auto*

lemma *card-Suc-Diff1*: *finite A* ==> *x*: *A* ==> *Suc* (*card* (*A* − {*x*})) = *card A*
apply(*rule-tac t* = *A* **in** *insert-Diff* [*THEN subst*], *assumption*)
apply(*simp del*:*insert-Diff-single*)
done

lemma *card-Diff-singleton*:
finite A ==> x: A ==> card (A − {x}) = card A − 1
by (*simp add*: *card-Suc-Diff1* [*symmetric*])

lemma *card-Diff-singleton-if*:
finite A ==> card (A−{x}) = (if x : A then card A − 1 else card A)
by (*simp add*: *card-Diff-singleton*)

lemma *card-Diff-insert*[*simp*]:
assumes *finite A* **and** *a*:*A* **and** *a* ~: *B*
shows *card*(*A − insert a B*) = *card*(*A − B*) − *1*
proof −
have *A − insert a B* = (*A − B*) − {*a*} **using** *assms* **by** *blast*
then show *?thesis* **using** *assms* **by**(*simp add*:*card-Diff-singleton*)
qed

lemma *card-insert*: *finite A ==> card* (*insert x A*) = *Suc* (*card* (*A* − {*x*}))
by (*simp add*: *card-insert-if card-Suc-Diff1 del*:*card-Diff-insert*)

lemma *card-insert-le*: *finite A ==> card A <= card* (*insert x A*)
by (*simp add*: *card-insert-if*)

lemma *card-mono*: ⟦ *finite B*; *A* ⊆ *B* ⟧ ⟹ *card A* ≤ *card B*
by (*simp add*: *card-def setsum-mono2*)

lemma *card-seteq*: *finite B ==>* (!!*A*. *A <= B ==> card B <= card A ==> A* = *B*)
apply (*induct set*: *finite*, *simp*, *clarify*)
apply (*subgoal-tac finite A* & *A* − {*x*} <= *F*)
prefer *2* **apply** (*blast intro*: *finite-subset*, *atomize*)
apply (*drule-tac x* = *A* − {*x*} **in** *spec*)
apply (*simp add*: *card-Diff-singleton-if split add*: *split-if-asm*)
apply (*case-tac card A*, *auto*)
done

lemma *psubset-card-mono*: *finite B ==> A < B ==> card A < card B*
apply (*simp add*: *psubset-eq linorder-not-le* [*symmetric*])
apply (*blast dest*: *card-seteq*)
done

lemma *card-Un-Int*: *finite A ==> finite B*
==> card A + card B = card (*A Un B*) + *card* (*A Int B*)
by(*simp add*:*card-def setsum-Un-Int*)

lemma *card-Un-disjoint*: *finite A ==> finite B*
==> A Int B = {} *==> card* (*A Un B*) = *card A* + *card B*
by (*simp add*: *card-Un-Int*)

lemma *card-Diff-subset*:

finite B ==> B <= A ==> card (A − B) = card A − card B
by(*simp add*:*card-def setsum-diff-nat*)

lemma *card-Diff1-less*: *finite A ==> x*: *A ==> card (A − {x}) < card A*
apply (*rule Suc-less-SucD*)
apply (*simp add*: *card-Suc-Diff1 del*:*card-Diff-insert*)
done

lemma *card-Diff2-less*:
finite A ==> x: *A ==> y*: *A ==> card (A − {x} − {y}) < card A*
apply (*case-tac x = y*)
apply (*simp add*: *card-Diff1-less del*:*card-Diff-insert*)
apply (*rule less-trans*)
prefer *2* **apply** (*auto intro*!: *card-Diff1-less simp del*:*card-Diff-insert*)
done

lemma *card-Diff1-le*: *finite A ==> card (A − {x}) <= card A*
apply (*case-tac x* : *A*)
apply (*simp-all add*: *card-Diff1-less less-imp-le*)
done

lemma *card-psubset*: *finite B ==> A ⊆ B ==> card A < card B ==> A < B*
by (*erule psubsetI*, *blast*)

lemma *insert-partition*:
⟦ *x ∉ F*; ∀ *c1* ∈ *insert x F*. ∀ *c2* ∈ *insert x F*. *c1 ≠ c2* ⟶ *c1* ∩ *c2* = {} ⟧
⟹ *x* ∩ ⋃ *F* = {}
by *auto*

main cardinality theorem

lemma *card-partition* [*rule-format*]:
finite C ==>
finite (⋃ *C*) −−>
(∀ *c*∈*C*. *card c* = *k*) −−>
(∀ *c1* ∈ *C*. ∀ *c2* ∈ *C*. *c1 ≠ c2* −−> *c1* ∩ *c2* = {}) −−>
k ∗ *card*(*C*) = *card* (⋃ *C*)
apply (*erule finite-induct*, *simp*)
apply (*simp add*: *card-insert-disjoint card-Un-disjoint insert-partition*
finite-subset [*of* - ⋃ (*insert x F*)])
done

The form of a finite set of given cardinality

lemma *card-eq-SucD*:
assumes *card A = Suc k*
shows ∃ *b B*. *A = insert b B* & *b ∉ B* & *card B = k* & (*k=0* ⟶ *B*={})
proof −
have *fin*: *finite A* **using** *assms* **by** (*auto intro*: *ccontr*)
moreover have *card A ≠ 0* **using** *assms* **by** *auto*
ultimately obtain *b* **where** *b*: *b ∈ A* **by** *auto*

```
  show ?thesis
  proof (intro exI conjI)
    show A = insert b (A−{b}) using b by blast
    show b ∉ A − {b} by blast
    show card (A − {b}) = k and k = 0 ⟶ A − {b} = {}
      using assms b fin by(fastsimp dest:mk-disjoint-insert)+
  qed
qed

lemma card-Suc-eq:
  (card A = Suc k) =
   (∃ b B. A = insert b B & b ∉ B & card B = k & (k=0 ⟶ B={}))
apply(rule iffI)
 apply(erule card-eq-SucD)
apply(auto)
apply(subst card-insert)
 apply(auto intro:ccontr)
done

lemma setsum-constant [simp]: (∑ x ∈ 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 ==> (∏ x∈ 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
```

20.6.1 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
```

20.6.2 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
proof (induct rule: finite-induct)
  case empty then show ?case by simp
next
  interpret ab-semigroup-mult [op Un]
    by unfold-locales auto
  case insert
  then show ?case by simp
qed

lemma card-image-le: finite A ==> card (f ' A) <= card A
  apply (induct set: finite)
   apply simp
  apply (simp add: le-SucI finite-imageI card-insert-if)
  done

lemma card-image: inj-on f A ==> card (f ' A) = card A
by(simp add:card-def setsum-reindex o-def del:setsum-constant)

lemma endo-inj-surj: finite A ==> f ' A ⊆ A ==> inj-on f A ==> f ' A = A
by (simp add: card-seteq card-image)

lemma eq-card-imp-inj-on:
  [| finite A; card(f ' A) = card A |] ==> inj-on f A
apply (induct rule:finite-induct)
apply simp
apply(frule card-image-le[where f = f])
apply(simp add:card-insert-if split:if-splits)
done

lemma inj-on-iff-eq-card:
  finite A ==> inj-on f A = (card(f ' A) = card A)
by(blast intro: card-image eq-card-imp-inj-on)
```

lemma *card-inj-on-le*:
[|*inj-on f A*; *f ' A* ⊆ *B*; *finite B* |] ==> *card A* ≤ *card B*
apply (*subgoal-tac finite A*)
apply (*force intro*: *card-mono simp add*: *card-image* [*symmetric*])
apply (*blast intro*: *finite-imageD dest*: *finite-subset*)
done

lemma *card-bij-eq*:
[|*inj-on f A*; *f ' A* ⊆ *B*; *inj-on g B*; *g ' B* ⊆ *A*;
finite A; *finite B* |] ==> *card A* = *card B*
by (*auto intro*: *le-anti-sym card-inj-on-le*)

20.6.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
⟦ *finite A*; *ALL a*:*A*. *finite* (*B a*) ⟧
⟹ *card* (*SIGMA x*: *A*. *B x*) = (∑ *a*∈*A*. *card* (*B a*))
by(*simp add*:*card-def setsum-Sigma del*:*setsum-constant*)

lemma *card-cartesian-product*: *card* (*A* <*> *B*) = *card*(*A*) * *card*(*B*)
apply (*cases finite A*)
apply (*cases finite B*)
apply (*auto simp add*: *card-eq-0-iff*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *card-cartesian-product-singleton*: *card*({*x*} <*> *A*) = *card*(*A*)
by (*simp add*: *card-cartesian-product*)

20.6.4 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

20.6.5 Relating injectivity and surjectivity

lemma *finite-surj-inj*: *finite*(*A*) $\Longrightarrow$ *A* <= *f*‘*A* $\Longrightarrow$ *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*) $\Longrightarrow$ *surj f* $\Longrightarrow$ *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*) $\Longrightarrow$ *inj f* $\Longrightarrow$ *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

20.7 A fold functional for non-empty sets

Does not require start value.

inductive
  fold1Set :: (*'a* => *'a* => *'a*) => *'a set* => *'a* => *bool*
  for *f* :: *'a* => *'a* => *'a*
where
  fold1Set-insertI [*intro*]:
   ⟦ *foldSet f id a A x*; *a* ∉ *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* ≠ {}
  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* ≠ {} ⟧ ⟹ *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*.

context *ab-semigroup-mult*
begin

lemma *foldSet-insert-swap*:
assumes *fold*: *foldSet times id b A y*
shows *b* ∉ *A* ⟹ *foldSet times id z* (*insert b A*) (*z* ∗ *y*)
using *fold*
proof (*induct rule*: *foldSet.induct*)
case *emptyI* **thus** *?case* **by** (*force simp add*: *fold-insert-aux mult-commute*)
next
case (*insertI x A y*)
have *foldSet times* (λ*u*. *u*) *z* (*insert x* (*insert b A*)) (*x* ∗ (*z* ∗ *y*))
using *insertI* **by** *force* — how does *id* get unfolded?
thus *?case* **by** (*simp add*: *insert-commute mult-ac*)
qed

lemma *foldSet-permute-diff*:
assumes *fold*: *foldSet times id b A x*
shows !!*a*. ⟦*a* ∈ *A*; *b* ∉ *A*⟧ ⟹ *foldSet times 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* ∨ *a* ∈ *A* **using** *insertI* **by** *simp*
thus *?case*
proof
assume *a* = *x*
with *insertI* **show** *?thesis*
by (*simp add*: *id-def* [*symmetric*], *blast intro*: *foldSet-insert-swap*)

```
  next
    assume ainA: a ∈ A
    hence foldSet times id a (insert x (insert b (A − {a}))) (x ∗ 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 fold1-eq-fold:
    [|finite A; a ∉ A|] ==> fold1 times (insert a A) = fold times 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 - times id])
apply (rule sym, clarify)
apply (case-tac Aa=A)
 apply (best intro: the-equality foldSet-determ)
apply (subgoal-tac foldSet times 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 ≠ {}) = (∃x B. A = insert x B & x ∉ B)
apply safe
apply simp
apply (drule-tac x=x in spec)
apply (drule-tac x=A−{x} in spec, auto)
done

lemma fold1-insert:
  assumes nonempty: A ≠ {} and A: finite A x ∉ A
  shows fold1 times (insert x A) = x ∗ fold1 times A
proof −
  from nonempty obtain a A' where A = insert a A' & a ~: A'
    by (auto simp add: nonempty-iff)
  with A show ?thesis
    by (simp add: insert-commute [of x] fold1-eq-fold eq-commute)
qed

end

context ab-semigroup-idem-mult
begin

lemma fold1-insert-idem [simp]:
```

```
  assumes nonempty: A ≠ {} and A: finite A
  shows fold1 times (insert x A) = x * fold1 times 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: mult-idem)
    next
      assume A' ≠ {}
      with prems show ?thesis
        by (simp add: fold1-insert mult-assoc [symmetric] mult-idem)
    qed
  next
    assume a ≠ x
    with prems show ?thesis
      by (simp add: insert-commute fold1-eq-fold fold-insert-idem)
  qed
qed

lemma hom-fold1-commute:
assumes hom: !!x y. h (x * y) = h x * h y
and N: finite N N ≠ {} shows h (fold1 times N) = fold1 times (h ' N)
using N proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case (insert n N)
  then have h (fold1 times (insert n N)) = h (n * fold1 times N) by simp
  also have ... = h n * h (fold1 times N) by(rule hom)
  also have h (fold1 times N) = fold1 times (h ' N) by(rule insert)
  also have times (h n) ... = fold1 times (insert (h n) (h ' N))
    using insert by(simp)
  also have insert (h n) (h ' N) = h ' insert n N by simp
  finally show ?case .
qed

end
```

Now the recursion rules for definitions:

```
lemma fold1-singleton-def: g = fold1 f ⟹ g {a} = a
by(simp add:fold1-singleton)

lemma (in ab-semigroup-mult) fold1-insert-def:
  ⟦ g = fold1 times; finite A; x ∉ A; A ≠ {} ⟧ ⟹ g (insert x A) = x * g A
by (simp add:fold1-insert)
```

lemma (**in** *ab-semigroup-idem-mult*) *fold1-insert-idem-def*:
⟦ g = *fold1 times*; *finite* A; $A \neq \{\}$ ⟧ ⟹ g (*insert* x A) = $x * g\ A$
by *simp*

20.7.1 Determinacy for *fold1Set*

Not actually used!!

context *ab-semigroup-mult*
begin

lemma *foldSet-permute*:
[|*foldSet times id b* (*insert a A*) x; $a \notin A$; $b \notin A$|]
==> *foldSet times id a* (*insert b A*) x
apply (*cases* a=b)
apply (*auto dest*: *foldSet-permute-diff*)
done

lemma *fold1Set-determ*:
fold1Set times A x ==> *fold1Set times A y* ==> $y = x$
proof (*clarify elim*!: *fold1Set.cases*)
fix A x B y a b
assume *Ax*: *foldSet times id a A x*
assume *By*: *foldSet times 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≠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 times id a* (*insert b ?D*) y
by (*auto intro*: *foldSet-permute simp add*: *insert-absorb*)
moreover
have *foldSet times id a* (*insert b ?D*) x
by (*simp add*: A [*symmetric*] *Ax*)
ultimately show *?thesis* **by** (*blast intro*: *foldSet-determ*)
qed
qed

lemma *fold1Set-equality*: *fold1Set times A y* ==> *fold1 times A* = y

by (*unfold fold1-def*) (*blast intro*: *fold1Set-determ*)

end

declare
empty-foldSetE [*rule del*] *foldSet.intros* [*rule del*]
empty-fold1SetE [*rule del*] *insert-fold1SetE* [*rule del*]
— No more proofs involve these relations.

20.7.2 Lemmas about *fold1*

context *ab-semigroup-mult*
begin

lemma *fold1-Un*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹ *A Int B* = {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
using *A* **by** (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert mult-assoc*)

lemma *fold1-in*:
assumes *A*: *finite* (*A*) *A* ≠ {} **and** *elem*: ⋀*x y*. *x* ∗ *y* ∈ {*x*,*y*}
shows *fold1 times A* ∈ *A*
using *A*
proof (*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **using** *elem* **by** (*force simp add*:*fold1-insert*)
qed

end

lemma (**in** *ab-semigroup-idem-mult*) *fold1-Un2*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
using *A*
proof(*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **by** (*simp add*: *mult-assoc*)
qed

20.7.3 Fold1 in lattices with *inf* and *sup*

As an application of *fold1* we define infimum and supremum in (not necessarily complete!) lattices over (non-empty) sets by means of *fold1*.

context *lower-semilattice*

begin

lemma *ab-semigroup-idem-mult-inf*:
ab-semigroup-idem-mult inf
apply *unfold-locales*
apply (*rule inf-assoc*)
apply (*rule inf-commute*)
apply (*rule inf-idem*)
done

lemma *below-fold1-iff*:
assumes *finite A* $A \neq \{\}$
shows $x \leq$ *fold1 inf A* $\longleftrightarrow (\forall a{\in}A.\ x \leq a)$
proof −
interpret *ab-semigroup-idem-mult* [*inf*]
by (*rule ab-semigroup-idem-mult-inf*)
show *?thesis* **using** *assms* **by** (*induct rule*: *finite-ne-induct*) *simp-all*
qed

lemma *fold1-belowI*:
assumes *finite A*
and $a \in A$
shows *fold1 inf A* $\leq a$
proof −
from *assms* **have** $A \neq \{\}$ **by** *auto*
from ‹*finite A*› ‹$A \neq \{\}$› ‹$a \in A$› **show** *?thesis*
proof (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
interpret *ab-semigroup-idem-mult* [*inf*]
by (*rule ab-semigroup-idem-mult-inf*)
case (*insert x F*)
from *insert*(*5*) **have** $a = x \vee a \in F$ **by** *simp*
thus *?case*
proof
assume $a = x$ **thus** *?thesis* **using** *insert*
by (*simp add*: *mult-ac-idem*)
next
assume $a \in F$
hence *bel*: *fold1 inf F* $\leq a$ **by** (*rule insert*)
have *inf* (*fold1 inf* (*insert x F*)) $a =$ *inf x* (*inf* (*fold1 inf F*) a)
using *insert* **by** (*simp add*: *mult-ac-idem*)
also have *inf* (*fold1 inf F*) $a =$ *fold1 inf F*
using *bel* **by** (*auto intro*: *antisym*)
also have *inf x* ... = *fold1 inf* (*insert x F*)
using *insert* **by** (*simp add*: *mult-ac-idem*)
finally have *aux*: *inf* (*fold1 inf* (*insert x F*)) $a =$ *fold1 inf* (*insert x F*) **.**
moreover have *inf* (*fold1 inf* (*insert x F*)) $a \leq a$ **by** *simp*
ultimately show *?thesis* **by** *simp*

qed
qed
qed

end

lemma (**in** *upper-semilattice*) *ab-semigroup-idem-mult-sup*:
ab-semigroup-idem-mult sup
by (*rule lower-semilattice.ab-semigroup-idem-mult-inf*)
(*rule dual-lattice*)

context *lattice*
begin

definition
Inf-fin :: *'a set* ⇒ *'a* ($\bigsqcap_{fin}$- [*900*] *900*)
where
Inf-fin = *fold1 inf*

definition
Sup-fin :: *'a set* ⇒ *'a* ($\bigsqcup_{fin}$- [*900*] *900*)
where
Sup-fin = *fold1 sup*

lemma *Inf-le-Sup* [*simp*]: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ $\bigsqcap_{fin} A \leq \bigsqcup_{fin} A$
apply(*unfold Sup-fin-def Inf-fin-def*)
apply(*subgoal-tac EX a*. *a*:*A*)
prefer *2* **apply** *blast*
apply(*erule exE*)
apply(*rule order-trans*)
apply(*erule* (*1*) *fold1-belowI*)
apply(*erule* (*1*) *lower-semilattice.fold1-belowI* [*OF dual-lattice*])
done

lemma *sup-Inf-absorb* [*simp*]:
finite A ⟹ *a* ∈ *A* ⟹ *sup a* ($\bigsqcap_{fin} A$) = *a*
apply(*subst sup-commute*)
apply(*simp add*: *Inf-fin-def sup-absorb2 fold1-belowI*)
done

lemma *inf-Sup-absorb* [*simp*]:
finite A ⟹ *a* ∈ *A* ⟹ *inf a* ($\bigsqcup_{fin} A$) = *a*
by (*simp add*: *Sup-fin-def inf-absorb1*
lower-semilattice.fold1-belowI [*OF dual-lattice*])

end

context *distrib-lattice*
begin

lemma *sup-Inf1-distrib*:
assumes *finite A*
and $A \neq \{\}$
shows *sup x* $(\bigsqcap_{fin} A) = \bigsqcap_{fin}\{sup\ x\ a | a.\ a \in A\}$
proof −
interpret *ab-semigroup-idem-mult* [*inf*]
by (*rule ab-semigroup-idem-mult-inf*)
from *assms* **show** *?thesis*
by (*simp add*: *Inf-fin-def image-def*
hom-fold1-commute [**where** *h*=*sup x*, *OF sup-inf-distrib1*])
(*rule arg-cong* [**where** *f*=*fold1 inf*], *blast*)
qed

lemma *sup-Inf2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows *sup* $(\bigsqcap_{fin} A)$ $(\bigsqcap_{fin} B) = \bigsqcap_{fin}\{sup\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by (*simp add*: *sup-Inf1-distrib* [*OF B*] *fold1-singleton-def* [*OF Inf-fin-def*])
next
interpret *ab-semigroup-idem-mult* [*inf*]
by (*rule ab-semigroup-idem-mult-inf*)
case (*insert x A*)
have *finB*: *finite* $\{sup\ x\ b\ |b.\ b \in B\}$
by(*rule finite-surj*[**where** *f* = *sup x*, *OF B*(*1*)], *auto*)
have *finAB*: *finite* $\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\}$
proof −
have $\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} = (UN\ a{:}A.\ UN\ b{:}B.\ \{sup\ a\ b\})$
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: $\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B* **by** *blast*
have *sup* $(\bigsqcap_{fin}(insert\ x\ A))$ $(\bigsqcap_{fin} B) = sup\ (inf\ x\ (\bigsqcap_{fin} A))\ (\bigsqcap_{fin} B)$
using *insert* **by** (*simp add*: *fold1-insert-idem-def* [*OF Inf-fin-def*])
also have $\ldots = inf\ (sup\ x\ (\bigsqcap_{fin} B))\ (sup\ (\bigsqcap_{fin} A)\ (\bigsqcap_{fin} B))$ **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 fold1-Un2* [*OF finB - finAB ne*])
also have *?M* = $\{sup\ a\ b\ |a\ b.\ a \in insert\ x\ A \wedge b \in B\}$
by *blast*
finally show *?case* **.**
qed

lemma *inf-Sup1-distrib*:

assumes *finite A* **and** $A \neq \{\}$
shows *inf x* $(\bigsqcup_{fin} A) = \bigsqcup_{fin}\{inf\ x\ a | a.\ a \in A\}$
proof −
interpret *ab-semigroup-idem-mult* [*sup*]
by (*rule ab-semigroup-idem-mult-sup*)
from *assms* **show** *?thesis*
by (*simp add*: *Sup-fin-def image-def hom-fold1-commute* [**where** *h*=*inf x*, *OF inf-sup-distrib1*])
(*rule arg-cong* [**where** *f*=*fold1 sup*], *blast*)
qed

lemma *inf-Sup2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows *inf* $(\bigsqcup_{fin} A)$ $(\bigsqcup_{fin} B) = \bigsqcup_{fin}\{inf\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by(*simp add*: *inf-Sup1-distrib* [*OF B*] *fold1-singleton-def* [*OF Sup-fin-def*])
next
case (*insert x A*)
have *finB*: *finite* $\{inf\ x\ b\ | b.\ b \in B\}$
by(*rule finite-surj*[**where** $f = \%b.\ inf\ x\ b$, *OF B*(*1*)], *auto*)
have *finAB*: *finite* $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\}$
proof −
have $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\} = (UN\ a{:}A.\ UN\ b{:}B.\ \{inf\ a\ b\})$
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B* **by** *blast*
interpret *ab-semigroup-idem-mult* [*sup*]
by (*rule ab-semigroup-idem-mult-sup*)
have *inf* $(\bigsqcup_{fin}(insert\ x\ A))$ $(\bigsqcup_{fin} B) = inf\ (sup\ x\ (\bigsqcup_{fin} A))\ (\bigsqcup_{fin} B)$
using *insert* **by** (*simp add*: *fold1-insert-idem-def* [*OF Sup-fin-def*])
also have $\ldots = sup\ (inf\ x\ (\bigsqcup_{fin} B))\ (inf\ (\bigsqcup_{fin} A)\ (\bigsqcup_{fin} B))$ **by**(*rule inf-sup-distrib2*)
also have $\ldots = sup\ (\bigsqcup_{fin}\{inf\ x\ b | b.\ b \in B\})\ (\bigsqcup_{fin}\{inf\ a\ b | a\ b.\ a \in A \wedge b \in B\})$
using *insert* **by**(*simp add*:*inf-Sup1-distrib*[*OF B*])
also have $\ldots = \bigsqcup_{fin}(\{inf\ x\ b\ | b.\ b \in B\} \cup \{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\})$
(**is** - = $\bigsqcup_{fin}$ *?M*)
using *B insert*
by (*simp add*: *Sup-fin-def fold1-Un2* [*OF finB - finAB ne*])
also have *?M* = $\{inf\ a\ b\ | a\ b.\ a \in insert\ x\ A \wedge b \in B\}$
by *blast*
finally show *?case* **.**
qed

end

context *complete-lattice*
begin

Coincidence on finite sets in complete lattices:

lemma *Inf-fin-Inf*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcap_{fin} A = Inf\ A$
proof −
interpret *ab-semigroup-idem-mult* [*inf*]
by (*rule ab-semigroup-idem-mult-inf*)
from *assms* **show** *?thesis*
unfolding *Inf-fin-def* **by** (*induct A set*: *finite*)
(*simp-all add*: *Inf-insert-simp*)
qed

lemma *Sup-fin-Sup*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcup_{fin} A = Sup\ A$
proof −
interpret *ab-semigroup-idem-mult* [*sup*]
by (*rule ab-semigroup-idem-mult-sup*)
from *assms* **show** *?thesis*
unfolding *Sup-fin-def* **by** (*induct A set*: *finite*)
(*simp-all add*: *Sup-insert-simp*)
qed

end

20.7.4 Fold1 in linear orders with *min* and *max*

As an application of *fold1* we define minimum and maximum in (not necessarily complete!) linear orders over (non-empty) sets by means of *fold1*.

context *linorder*
begin

lemma *ab-semigroup-idem-mult-min*:
ab-semigroup-idem-mult min
by *unfold-locales* (*auto simp add*: *min-def*)

lemma *ab-semigroup-idem-mult-max*:
ab-semigroup-idem-mult max
by *unfold-locales* (*auto simp add*: *max-def*)

lemma *min-lattice*:
lower-semilattice (*op* ≤) (*op* <) *min*
by *unfold-locales* (*auto simp add*: *min-def*)

lemma *max-lattice*:
lower-semilattice (*op* ≥) (*op* >) *max*
by *unfold-locales* (*auto simp add*: *max-def*)

lemma *dual-max*:
ord.max (*op* ≥) = *min*
by (*auto simp add*: *ord.max-def-raw min-def-raw expand-fun-eq*)

lemma *dual-min*:
ord.min (*op* ≥) = *max*
by (*auto simp add*: *ord.min-def-raw max-def-raw expand-fun-eq*)

lemma *strict-below-fold1-iff*:
assumes *finite A* **and** *A* ≠ {}
shows *x* < *fold1 min A* ⟷ (∀ *a*∈*A*. *x* < *a*)
proof −
interpret *ab-semigroup-idem-mult* [*min*]
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert*)
qed

lemma *fold1-below-iff*:
assumes *finite A* **and** *A* ≠ {}
shows *fold1 min A* ≤ *x* ⟷ (∃ *a*∈*A*. *a* ≤ *x*)
proof −
interpret *ab-semigroup-idem-mult* [*min*]
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert min-le-iff-disj*)
qed

lemma *fold1-strict-below-iff*:
assumes *finite A* **and** *A* ≠ {}
shows *fold1 min A* < *x* ⟷ (∃ *a*∈*A*. *a* < *x*)
proof −
interpret *ab-semigroup-idem-mult* [*min*]
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert min-less-iff-disj*)
qed

lemma *fold1-antimono*:
assumes *A* ≠ {} **and** *A* ⊆ *B* **and** *finite B*
shows *fold1 min B* ≤ *fold1 min A*
proof *cases*
assume *A* = *B* **thus** *?thesis* **by** *simp*
next
interpret *ab-semigroup-idem-mult* [*min*]
by (*rule ab-semigroup-idem-mult-min*)

```
  assume A ≠ B
  have B: B = A ∪ (B−A) using ‹A ⊆ B› by blast
  have fold1 min B = fold1 min (A ∪ (B−A)) by(subst B)(rule refl)
  also have ... = min (fold1 min A) (fold1 min (B−A))
  proof −
    have finite A by(rule finite-subset[OF ‹A ⊆ B› ‹finite B›])
    moreover have finite(B−A) by(rule finite-Diff[OF ‹finite B›])
    moreover have (B−A) ≠ {} using prems by blast
    moreover have A Int (B−A) = {} using prems by blast
    ultimately show ?thesis using ‹A ≠ {}› by (rule-tac fold1-Un)
  qed
  also have ... ≤ fold1 min A by (simp add: min-le-iff-disj)
  finally show ?thesis .
qed

definition
  Min :: 'a set ⇒ 'a
where
  Min = fold1 min

definition
  Max :: 'a set ⇒ 'a
where
  Max = fold1 max

lemmas Min-singleton [simp] = fold1-singleton-def [OF Min-def]
lemmas Max-singleton [simp] = fold1-singleton-def [OF Max-def]

lemma Min-insert [simp]:
  assumes finite A and A ≠ {}
  shows Min (insert x A) = min x (Min A)
proof −
  interpret ab-semigroup-idem-mult [min]
    by (rule ab-semigroup-idem-mult-min)
  from assms show ?thesis by (rule fold1-insert-idem-def [OF Min-def])
qed

lemma Max-insert [simp]:
  assumes finite A and A ≠ {}
  shows Max (insert x A) = max x (Max A)
proof −
  interpret ab-semigroup-idem-mult [max]
    by (rule ab-semigroup-idem-mult-max)
  from assms show ?thesis by (rule fold1-insert-idem-def [OF Max-def])
qed

lemma Min-in [simp]:
  assumes finite A and A ≠ {}
  shows Min A ∈ A
```

proof −
interpret *ab-semigroup-idem-mult* [*min*]
by (*rule ab-semigroup-idem-mult-min*)
from *assms fold1-in* **show** *?thesis* **by** (*fastsimp simp*: *Min-def min-def*)
qed

lemma *Max-in* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Max\ A \in A$
proof −
interpret *ab-semigroup-idem-mult* [*max*]
by (*rule ab-semigroup-idem-mult-max*)
from *assms fold1-in* [*of A*] **show** *?thesis* **by** (*fastsimp simp*: *Max-def max-def*)
qed

lemma *Min-Un*:
assumes *finite A* **and** $A \neq \{\}$ **and** *finite B* **and** $B \neq \{\}$
shows $Min\ (A \cup B) = min\ (Min\ A)\ (Min\ B)$
proof −
interpret *ab-semigroup-idem-mult* [*min*]
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-Un2*)
qed

lemma *Max-Un*:
assumes *finite A* **and** $A \neq \{\}$ **and** *finite B* **and** $B \neq \{\}$
shows $Max\ (A \cup B) = max\ (Max\ A)\ (Max\ B)$
proof −
interpret *ab-semigroup-idem-mult* [*max*]
by (*rule ab-semigroup-idem-mult-max*)
from *assms* **show** *?thesis*
by (*simp add*: *Max-def fold1-Un2*)
qed

lemma *hom-Min-commute*:
assumes $\bigwedge x\ y.\ h\ (min\ x\ y) = min\ (h\ x)\ (h\ y)$
and *finite N* **and** $N \neq \{\}$
shows $h\ (Min\ N) = Min\ (h\ `\ N)$
proof −
interpret *ab-semigroup-idem-mult* [*min*]
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def hom-fold1-commute*)
qed

lemma *hom-Max-commute*:
assumes $\bigwedge x\ y.\ h\ (max\ x\ y) = max\ (h\ x)\ (h\ y)$
and *finite N* **and** $N \neq \{\}$

shows $h\ (Max\ N) = Max\ (h\ `\ N)$
proof −
interpret *ab-semigroup-idem-mult* [*max*]
by (*rule ab-semigroup-idem-mult-max*)
from *assms* **show** *?thesis*
by (*simp add*: *Max-def hom-fold1-commute* [*of h*])
qed

lemma *Min-le* [*simp*]:
assumes *finite A* **and** $x \in A$
shows $Min\ A \leq x$
proof −
interpret *lower-semilattice* [$op \leq op < min$]
by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def fold1-belowI*)
qed

lemma *Max-ge* [*simp*]:
assumes *finite A* **and** $x \in A$
shows $x \leq Max\ A$
proof −
invoke *lower-semilattice* [$op \geq op > max$]
by (*rule max-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Max-def fold1-belowI*)
qed

lemma *Min-ge-iff* [*simp*, *noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x \leq Min\ A \longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
proof −
interpret *lower-semilattice* [$op \leq op < min$]
by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def below-fold1-iff*)
qed

lemma *Max-le-iff* [*simp*, *noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Max\ A \leq x \longleftrightarrow (\forall\, a{\in}A.\ a \leq x)$
proof −
invoke *lower-semilattice* [$op \geq op > max$]
by (*rule max-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Max-def below-fold1-iff*)
qed

lemma *Min-gr-iff* [*simp*, *noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x < Min\ A \longleftrightarrow (\forall\, a{\in}A.\ x < a)$
proof −
interpret *lower-semilattice* [$op \leq op < min$]

by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def strict-below-fold1-iff*)
qed

lemma *Max-less-iff* [*simp*, *noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Max\ A < x \longleftrightarrow (\forall\, a{\in}A.\ a < x)$
proof −
note *Max* = *Max-def*
interpret *linorder* [$op \geq\ op >$]
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max strict-below-fold1-iff* [*folded dual-max*])
qed

lemma *Min-le-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
proof −
interpret *lower-semilattice* [$op \leq\ op <\ min$]
by (*rule min-lattice*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-below-iff*)
qed

lemma *Max-ge-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x \leq Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x \leq a)$
proof −
note *Max* = *Max-def*
interpret *linorder* [$op \geq\ op >$]
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-below-iff* [*folded dual-max*])
qed

lemma *Min-less-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
proof −
interpret *lower-semilattice* [$op \leq\ op <\ min$]
by (*rule min-lattice*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-strict-below-iff*)
qed

lemma *Max-gr-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x < Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x < a)$

proof −
note *Max* = *Max-def*
interpret *linorder* [*op* ≥ *op* >]
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-strict-below-iff* [*folded dual-max*])
qed

lemma *Min-antimono*:
assumes *M* ⊆ *N* **and** *M* ≠ {} **and** *finite N*
shows *Min N* ≤ *Min M*
proof −
interpret *distrib-lattice* [*op* ≤ *op* < *min max*]
by (*rule distrib-lattice-min-max*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def fold1-antimono*)
qed

lemma *Max-mono*:
assumes *M* ⊆ *N* **and** *M* ≠ {} **and** *finite N*
shows *Max M* ≤ *Max N*
proof −
note *Max* = *Max-def*
interpret *linorder* [*op* ≥ *op* >]
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-antimono* [*folded dual-max*])
qed

lemma *finite-linorder-induct*[*consumes 1*, *case-names empty insert*]:
finite A ⟹ *P* {} ⟹
(!!*A b*. *finite A* ⟹ *ALL a*:*A*. *a* < *b* ⟹ *P A* ⟹ *P*(*insert b A*))
⟹ *P A*
proof (*induct A rule*: *measure-induct-rule*[**where** *f*=*card*])
fix *A* :: *'a set*
assume *IH*: !! *B*. *card B* < *card A* ⟹ *finite B* ⟹ *P* {} ⟹
(!!*A b*. *finite A* ⟹ (∀ *a*∈*A*. *a*<*b*) ⟹ *P A* ⟹ *P* (*insert b A*))
⟹ *P B*
and *finite A* **and** *P* {}
and *step*: !!*A b*. ⟦*finite A*; ∀ *a*∈*A*. *a* < *b*; *P A*⟧ ⟹ *P* (*insert b A*)
show *P A*
proof (*cases A* = {})
assume *A* = {} **thus** *P A* **using** ‹*P* {}› **by** *simp*
next
let *?B* = *A* − {*Max A*} **let** *?A* = *insert* (*Max A*) *?B*
assume *A* ≠ {}
with ‹*finite A*› **have** *Max A* : *A* **by** *auto*
hence *A*: *?A* = *A* **using** *insert-Diff-single insert-absorb* **by** *auto*
note *card-Diff1-less*[*OF* ‹*finite A*› ‹*Max A* : *A*›]
moreover have *finite ?B* **using** ‹*finite A*› **by** *simp*

```
    ultimately have P ?B using ‹P {}› step IH by blast
    moreover have ∀ a∈?B. a < Max A
      using Max-ge [OF ‹finite A›] by fastsimp
    ultimately show P A
      using A insert-Diff-single step[OF ‹finite ?B›] by fastsimp
  qed
qed

end

context ordered-ab-semigroup-add
begin

lemma add-Min-commute:
  fixes k
  assumes finite N and N ≠ {}
  shows k + Min N = Min {k + m | m. m ∈ N}
proof −
  have ⋀x y. k + min x y = min (k + x) (k + y)
    by (simp add: min-def not-le)
      (blast intro: antisym less-imp-le add-left-mono)
  with assms show ?thesis
    using hom-Min-commute [of plus k N]
    by simp (blast intro: arg-cong [where f = Min])
qed

lemma add-Max-commute:
  fixes k
  assumes finite N and N ≠ {}
  shows k + Max N = Max {k + m | m. m ∈ N}
proof −
  have ⋀x y. k + max x y = max (k + x) (k + y)
    by (simp add: max-def not-le)
      (blast intro: antisym less-imp-le add-left-mono)
  with assms show ?thesis
    using hom-Max-commute [of plus k N]
    by simp (blast intro: arg-cong [where f = Max])
qed

end

end
```

21 Equiv-Relations: Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*

imports *Finite-Set Relation*
begin

21.1 Equivalence relations

locale *equiv* =
  fixes A **and** r
  assumes *refl*: *refl* A r
    and *sym*: *sym* r
    and *trans*: *trans* r

Suppes, Theorem 70: r is an equiv relation iff $r^{-1}\ O\ r = r$.

First half: *equiv* A r ==> $r^{-1}\ O\ r = r$.

lemma *sym-trans-comp-subset*:
    sym r ==> *trans* r ==> $r^{-1}\ O\ r \subseteq r$
  by (*unfold trans-def sym-def converse-def*) *blast*

lemma *refl-comp-subset*: *refl* A r ==> $r \subseteq r^{-1}\ O\ r$
  by (*unfold refl-def*) *blast*

lemma *equiv-comp-eq*: *equiv* A r ==> $r^{-1}\ O\ r = r$
  apply (*unfold equiv-def*)
  apply *clarify*
  apply (*rule equalityI*)
   apply (*iprover intro*: *sym-trans-comp-subset refl-comp-subset*)+
  done

Second half.

lemma *comp-equivI*:
    $r^{-1}\ O\ r = r$ ==> *Domain* $r = A$ ==> *equiv* A r
  apply (*unfold equiv-def refl-def sym-def trans-def*)
  apply (*erule equalityE*)
  apply (*subgoal-tac* $\forall x\ y.\ (x, y) \in r$ --> $(y, x) \in r$)
   apply *fast*
  apply *fast*
  done

21.2 Equivalence classes

lemma *equiv-class-subset*:
  equiv A r ==> $(a, b) \in r$ ==> $r``\{a\} \subseteq r``\{b\}$
  — lemma for the next result
  by (*unfold equiv-def trans-def sym-def*) *blast*

theorem *equiv-class-eq*: *equiv* A r ==> $(a, b) \in r$ ==> $r``\{a\} = r``\{b\}$
  apply (*assumption* | *rule equalityI equiv-class-subset*)+
  apply (*unfold equiv-def sym-def*)
  apply *blast*
  done

lemma *equiv-class-self*: *equiv A r ==> a ∈ 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) ∈ r) = (r``{x} = r``{y} & x ∈ A & y ∈ A)
by (*blast intro*!: *equiv-class-eq dest*: *eq-equiv-class equiv-type*)

theorem *eq-equiv-class-iff*:
equiv A r ==> x ∈ A ==> y ∈ A ==> (r``{x} = r``{y}) = ((x, y) ∈ r)
by (*blast intro*!: *equiv-class-eq dest*: *eq-equiv-class equiv-type*)

21.3 Quotients

constdefs
quotient :: ['*a set*, ('*a*∗'*a*) *set*] => '*a set set* (**infixl** '//' *90*)
A//r == ⋃*x* ∈ *A*. {*r``{x}*} — set of equiv classes

lemma *quotientI*: *x ∈ A ==> r``{x} ∈ A//r*
by (*unfold quotient-def*) *blast*

lemma *quotientE*:
X ∈ A//r ==> (!!x. X = r``{x} ==> x ∈ A ==> P) ==> P
by (*unfold quotient-def*) *blast*

lemma *Union-quotient*: *equiv A r ==> Union (A//r) = A*
by (*unfold equiv-def refl-def quotient-def*) *blast*

lemma *quotient-disj*:
equiv A r ==> X ∈ A//r ==> Y ∈ A//r ==> X = Y | (X ∩ Y = {})
apply (*unfold quotient-def*)
apply *clarify*
apply (*rule equiv-class-eq*)

apply *assumption*
apply (*unfold equiv-def trans-def sym-def*)
apply *blast*
done

lemma *quotient-eqI*:
[|*equiv A r*; $X \in A//r$; $Y \in A//r$; $x \in X$; $y \in Y$; $(x,y) \in r$|] ==> $X = Y$
apply (*clarify elim*!: *quotientE*)
apply (*rule equiv-class-eq*, *assumption*)
apply (*unfold equiv-def sym-def trans-def*, *blast*)
done

lemma *quotient-eq-iff*:
[|*equiv A r*; $X \in A//r$; $Y \in A//r$; $x \in X$; $y \in Y$|] ==> $(X = Y) = ((x,y) \in r)$
apply (*rule iffI*)
prefer *2* **apply** (*blast del*: *equalityI intro*: *quotient-eqI*)
apply (*clarify elim*!: *quotientE*)
apply (*unfold equiv-def sym-def trans-def*, *blast*)
done

lemma *eq-equiv-class-iff2*:
⟦ *equiv A r*; $x \in A$; $y \in A$ ⟧ ⟹ $(\{x\}//r = \{y\}//r) = ((x,y) : r)$
by(*simp add*:*quotient-def eq-equiv-class-iff*)

lemma *quotient-empty* [*simp*]: $\{\}//r = \{\}$
by(*simp add*: *quotient-def*)

lemma *quotient-is-empty* [*iff*]: $(A//r = \{\}) = (A = \{\})$
by(*simp add*: *quotient-def*)

lemma *quotient-is-empty2* [*iff*]: $(\{\} = A//r) = (A = \{\})$
by(*simp add*: *quotient-def*)

lemma *singleton-quotient*: $\{x\}//r = \{r `` \{x\}\}$
by(*simp add*:*quotient-def*)

lemma *quotient-diff1*:
⟦ *inj-on* (%*a*. $\{a\}//r$) *A*; $a \in A$ ⟧ ⟹ $(A - \{a\})//r = A//r - \{a\}//r$
apply(*simp add*:*quotient-def inj-on-def*)
apply *blast*
done

21.4 Defining unary operations upon equivalence classes

A congruence-preserving function

locale *congruent* =

fixes r **and** f
assumes *congruent*: $(y,z) \in r \Longrightarrow f\ y = f\ z$

abbreviation
$RESPECTS :: ('a \Rightarrow 'b) \Rightarrow ('a * 'a)\ set \Rightarrow bool$
(**infixr** *respects 80*) **where**
$f\ respects\ r == congruent\ r\ f$

lemma *UN-constant-eq*: $a \in A \Longrightarrow \forall y \in A.\ f\ y = c \Longrightarrow (\bigcup y \in A.\ f(y))=c$
— lemma required to prove *UN-equiv-class*
by *auto*

lemma *UN-equiv-class*:
$equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow a \in A$
$\Longrightarrow (\bigcup x \in r``\{a\}.\ f\ x) = f\ a$
— Conversion rule
apply (*rule equiv-class-self* [*THEN UN-constant-eq*], *assumption*+)
apply (*unfold equiv-def congruent-def sym-def*)
apply (*blast del*: *equalityI*)
done

lemma *UN-equiv-class-type*:
$equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow X \in A//r \Longrightarrow$
$(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow (\bigcup x \in X.\ f\ x) \in B$
apply (*unfold quotient-def*)
apply *clarify*
apply (*subst UN-equiv-class*)
apply *auto*
done

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; bcong could be $!!y.\ y \in A \Longrightarrow f\ y \in B$.

lemma *UN-equiv-class-inject*:
$equiv\ A\ r \Longrightarrow f\ respects\ r \Longrightarrow$
$(\bigcup x \in X.\ f\ x) = (\bigcup y \in Y.\ f\ y) \Longrightarrow X \in A//r \Longrightarrow Y \in A//r$
$\Longrightarrow (!!x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \Longrightarrow (x,\ y) \in r)$
$\Longrightarrow X = Y$
apply (*unfold quotient-def*)
apply *clarify*
apply (*rule equiv-class-eq*)
apply *assumption*
apply (*subgoal-tac* $f\ x = f\ xa$)
apply *blast*
apply (*erule box-equals*)
apply (*assumption* | *rule UN-equiv-class*)+
done

21.5 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments

locale *congruent2* =
  fixes *r1* **and** *r2* **and** *f*
  assumes *congruent2*:
    (*y1*,*z1*) ∈ *r1* ==> (*y2*,*z2*) ∈ *r2* ==> *f y1 y2* = *f z1 z2*

Abbreviation for the common case where the relations are identical

abbreviation
  RESPECTS2:: ['*a* => '*a* => '*b*, ('*a* * '*a*) *set*] => *bool*
    (**infixr** *respects2* 80) **where**
  f respects2 r == *congruent2 r r f*

lemma *congruent2-implies-congruent*:
    equiv A r1 ==> *congruent2 r1 r2 f* ==> *a* ∈ *A* ==> *congruent r2* (*f a*)
  by (*unfold congruent-def congruent2-def equiv-def refl-def*) *blast*

lemma *congruent2-implies-congruent-UN*:
  equiv A1 r1 ==> *equiv A2 r2* ==> *congruent2 r1 r2 f* ==> *a* ∈ *A2* ==>
    congruent r1 (λ*x1*. ⋃ *x2* ∈ *r2*``{*a*}. *f x1 x2*)
  apply (*unfold congruent-def*)
  apply *clarify*
  apply (*rule equiv-type* [*THEN subsetD*, *THEN SigmaE2*], *assumption*+)
  apply (*simp add*: *UN-equiv-class congruent2-implies-congruent*)
  apply (*unfold congruent2-def equiv-def refl-def*)
  apply (*blast del*: *equalityI*)
  done

lemma *UN-equiv-class2*:
  equiv A1 r1 ==> *equiv A2 r2* ==> *congruent2 r1 r2 f* ==> *a1* ∈ *A1* ==> *a2* ∈ *A2*
    ==> (⋃ *x1* ∈ *r1*``{*a1*}. ⋃ *x2* ∈ *r2*``{*a2*}. *f x1 x2*) = *f a1 a2*
  by (*simp add*: *UN-equiv-class congruent2-implies-congruent*
    congruent2-implies-congruent-UN)

lemma *UN-equiv-class-type2*:
  equiv A1 r1 ==> *equiv A2 r2* ==> *congruent2 r1 r2 f*
    ==> *X1* ∈ *A1*//*r1* ==> *X2* ∈ *A2*//*r2*
    ==> (!!*x1 x2*. *x1* ∈ *A1* ==> *x2* ∈ *A2* ==> *f x1 x2* ∈ *B*)
    ==> (⋃ *x1* ∈ *X1*. ⋃ *x2* ∈ *X2*. *f x1 x2*) ∈ *B*
  apply (*unfold quotient-def*)
  apply *clarify*
  apply (*blast intro*: *UN-equiv-class-type congruent2-implies-congruent-UN*
    congruent2-implies-congruent quotientI)
  done

lemma *UN-UN-split-split-eq*:

$(\bigcup(x1,\ x2) \in X.\ \bigcup(y1,\ y2) \in Y.\ A\ x1\ x2\ y1\ y2) =$
$(\bigcup x \in X.\ \bigcup y \in Y.\ (\lambda(x1,\ x2).\ (\lambda(y1,\ y2).\ A\ x1\ x2\ y1\ y2)\ y)\ x)$
— 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 (*rule congt* | *assumption* |
erule equivA [*THEN equiv-type*, *THEN subsetD*, *THEN SigmaE2*])+
done

21.6 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 ==> ∀ X ∈ A//r. k dvd card X
==> k dvd card A
apply (*rule Union-quotient* [*THEN subst* [**where** *P=λA. k dvd card A*]])
apply *assumption*
apply (*rule dvd-partition*)
prefer *3* **apply** (*blast dest*: *quotient-disj*)
apply (*simp-all add*: *Union-quotient equiv-type*)
done

lemma *card-quotient-disjoint*:
⟦ *finite A*; *inj-on* (*λx. {x} // r*) *A* ⟧ ⟹ *card*(*A//r*) = *card A*
apply(*simp add*:*quotient-def*)
apply(*subst card-UN-disjoint*)
apply *assumption*
apply *simp*
apply(*fastsimp simp add*:*inj-on-def*)
apply (*simp add*:*setsum-constant*)
done

end

22 Wellfounded: Well-founded Recursion

theory *Wellfounded*
imports *Finite-Set Nat*
uses (*Tools/function-package/size.ML*)
begin

22.1 Basic Definitions

inductive
wfrec-rel :: (*'a* * *'a*) *set* => ((*'a* => *'b*) => *'a* => *'b*) => *'a* => *'b* => *bool*
for *R* :: (*'a* * *'a*) *set*
and *F* :: (*'a* => *'b*) => *'a* => *'b*
where
wfrecI: *ALL z*. (*z*, *x*) : *R* --> *wfrec-rel R F z* (*g z*) ==>
wfrec-rel R F x (*F g x*)

constdefs
wf :: (*'a* * *'a*)*set* => *bool*
wf(*r*) == (!*P*. (!*x*. (!*y*. (*y*,*x*):*r* --> *P*(*y*)) --> *P*(*x*)) --> (!*x*. *P*(*x*)))

wfP :: (*'a* => *'a* => *bool*) => *bool*
wfP r == *wf* {(*x*, *y*). *r x y*}

acyclic :: (*'a***'a*)*set* => *bool*

acyclic r == !x. (x,x) ~: r^+

cut :: *('a => 'b) => ('a * 'a)set => 'a => 'a => 'b*
cut f r x == (%y. if (y,x):r then f y else 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 (λx y. (x, y) ∈ r) = wf r*
by (*simp add*: *wfP-def*)

lemma *wfUNIVI*:
(!!P x. (ALL x. (ALL y. (y,x) : r --> P(y)) --> P(x)) ==> P(x)) ==> wf(r)
unfolding *wf-def* **by** *blast*

lemmas *wfPUNIVI = wfUNIVI* [*to-pred*]

Restriction to domain *A* and range *B*. If *r* is well-founded over their intersection, then *wf r*

lemma *wfI*:
[| *r ⊆ A <*> B*;
!!x P. [|∀ x. (∀ y. (y,x) : r --> P y) --> P x; x : A; x : B |] ==> P x |]
==> wf r
unfolding *wf-def* **by** *blast*

lemma *wf-induct*:
[| *wf(r)*;
!!x.[| ALL y. (y,x): r --> P(y) |] ==> P(x)
|] ==> P(a)
unfolding *wf-def* **by** *blast*

lemmas *wfP-induct = wf-induct* [*to-pred*]

lemmas *wf-induct-rule = wf-induct* [*rule-format*, *consumes 1*, *case-names less*, *induct set*: *wf*]

lemmas *wfP-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set*: *wfP*]

lemma *wf-not-sym*: *wf r* ==> (*a*, *x*) : *r* ==> (*x*, *a*) ~: *r*
by (*induct a arbitrary*: *x set*: *wf*) *blast*

lemmas *wf-asym* = *wf-not-sym* [*elim-format*]

lemma *wf-not-refl* [*simp*]: *wf r* ==> (*a*, *a*) ~: *r*
by (*blast elim*: *wf-asym*)

lemmas *wf-irrefl* = *wf-not-refl* [*elim-format*]

22.2 Basic Results

transitive closure of a well-founded relation is well-founded!

lemma *wf-trancl*:
assumes *wf r*
shows *wf* (*r*^+)
proof −
{
fix *P* **and** *x*
assume *induct-step*: !!*x*. (!!*y*. (*y*, *x*) : *r*^+ ==> *P y*) ==> *P x*
have *P x*
proof (*rule induct-step*)
fix *y* **assume** (*y*, *x*) : *r*^+
with ‹*wf r*› **show** *P y*
proof (*induct x arbitrary*: *y*)
case (*less x*)
note *hyp* = ‹⋀*x'* *y'*. (*x'*, *x*) : *r* ==> (*y'*, *x'*) : *r*^+ ==> *P y'*›
from ‹(*y*, *x*) : *r*^+› **show** *P y*
proof *cases*
case *base*
show *P y*
proof (*rule induct-step*)
fix *y'* **assume** (*y'*, *y*) : *r*^+
with ‹(*y*, *x*) : *r*› **show** *P y'* **by** (*rule hyp* [*of y y'*])
qed
next
case *step*
then obtain *x'* **where** (*x'*, *x*) : *r* **and** (*y*, *x'*) : *r*^+ **by** *simp*
then show *P y* **by** (*rule hyp* [*of x' y*])
qed
qed
qed
} then show *?thesis* **unfolding** *wf-def* **by** *blast*
qed

lemmas *wfP-trancl* = *wf-trancl* [*to-pred*]

lemma *wf-converse-trancl*: *wf* (*r*^−1) ==> *wf* ((*r*^+)^−1)
apply (*subst trancl-converse* [*symmetric*])
apply (*erule wf-trancl*)
done

Minimal-element characterization of well-foundedness

lemma *wf-eq-minimal*: *wf r* = (∀ *Q x*. *x*∈*Q* −−> (∃ *z*∈*Q*. ∀ *y*. (*y*,*z*)∈*r* −−> *y*∉*Q*))
proof (*intro iffI strip*)
fix *Q* :: *'a set* **and** *x*
assume *wf r* **and** *x* ∈ *Q*
then show ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *r* ⟶ *y* ∉ *Q*
unfolding *wf-def*
by (*blast dest*: *spec* [*of* - %*x*. *x*∈*Q* ⟶ (∃ *z*∈*Q*. ∀ *y*. (*y*,*z*) ∈ *r* ⟶ *y*∉*Q*)])
next
assume *1*: ∀ *Q x*. *x* ∈ *Q* ⟶ (∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *r* ⟶ *y* ∉ *Q*)
show *wf r*
proof (*rule wfUNIVI*)
fix *P* :: *'a* ⇒ *bool* **and** *x*
assume *2*: ∀ *x*. (∀ *y*. (*y*, *x*) ∈ *r* ⟶ *P y*) ⟶ *P x*
let *?Q* = {*x*. ¬ *P x*}
have *x* ∈ *?Q* ⟶ (∃ *z* ∈ *?Q*. ∀ *y*. (*y*, *z*) ∈ *r* ⟶ *y* ∉ *?Q*)
by (*rule 1* [*THEN spec*, *THEN spec*])
then have ¬ *P x* ⟶ (∃ *z*. ¬ *P z* ∧ (∀ *y*. (*y*, *z*) ∈ *r* ⟶ *P y*)) **by** *simp*
with *2* **have** ¬ *P x* ⟶ (∃ *z*. ¬ *P z* ∧ *P z*) **by** *fast*
then show *P x* **by** *simp*
qed
qed

lemma *wfE-min*:
assumes *wf R x* ∈ *Q*
obtains *z* **where** *z* ∈ *Q* ⋀*y*. (*y*, *z*) ∈ *R* ⟹ *y* ∉ *Q*
using *assms* **unfolding** *wf-eq-minimal* **by** *blast*

lemma *wfI-min*:
(⋀*x Q*. *x* ∈ *Q* ⟹ ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *R* ⟶ *y* ∉ *Q*)
⟹ *wf R*
unfolding *wf-eq-minimal* **by** *blast*

lemmas *wfP-eq-minimal* = *wf-eq-minimal* [*to-pred*]

Well-foundedness of subsets

lemma *wf-subset*: [| *wf*(*r*); *p*<=*r* |] ==> *wf*(*p*)
apply (*simp* (*no-asm-use*) *add*: *wf-eq-minimal*)
apply *fast*
done

lemmas *wfP-subset* = *wf-subset* [*to-pred*]

Well-foundedness of the empty relation

lemma *wf-empty* [*iff*]: *wf*({})
by (*simp add*: *wf-def*)

lemmas *wfP-empty* [*iff*] =
wf-empty [*to-pred bot-empty-eq2*, *simplified bot-fun-eq bot-bool-eq*]

lemma *wf-Int1*: *wf r* ==> *wf* (*r Int r'*)
apply (*erule wf-subset*)
apply (*rule Int-lower1*)
done

lemma *wf-Int2*: *wf r* ==> *wf* (*r' Int r*)
apply (*erule wf-subset*)
apply (*rule Int-lower2*)
done

Well-foundedness of insert

lemma *wf-insert* [*iff*]: *wf*(*insert* (*y*,*x*) *r*) = (*wf*(*r*) & (*x*,*y*) ~: *r*^*)
apply (*rule iffI*)
apply (*blast elim*: *wf-trancl* [*THEN wf-irrefl*]
intro: *rtrancl-into-trancl1 wf-subset*
rtrancl-mono [*THEN* [*2*] *rev-subsetD*])
apply (*simp add*: *wf-eq-minimal*, *safe*)
apply (*rule allE*, *assumption*, *erule impE*, *blast*)
apply (*erule bexE*)
apply (*rename-tac a*, *case-tac a* = *x*)
prefer *2*
apply *blast*
apply (*case-tac y*:*Q*)
prefer *2* **apply** *blast*
apply (*rule-tac x* = {*z*. *z*:*Q* & (*z*,*y*) : *r*^*} **in** *allE*)
apply *assumption*
apply (*erule-tac V* = *ALL Q*. (*EX x*. *x* : *Q*) −−> *?P Q* **in** *thin-rl*)
— essential for speed

Blast with new substOccur fails

apply (*fast intro*: *converse-rtrancl-into-rtrancl*)
done

Well-foundedness of image

lemma *wf-prod-fun-image*: [| *wf r*; *inj f* |] ==> *wf*(*prod-fun f f* ‘ *r*)
apply (*simp only*: *wf-eq-minimal*, *clarify*)
apply (*case-tac EX p*. *f p* : *Q*)
apply (*erule-tac x* = {*p*. *f p* : *Q*} **in** *allE*)
apply (*fast dest*: *inj-onD*, *blast*)
done

22.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:
assumes *wf R wf S*
assumes $S\ O\ R \subseteq R$
shows *wf* $(R \cup S)$
proof (*rule wfI-min*)
fix $x :: 'a$ **and** Q
let $?Q' = \{x \in Q.\ \forall y.\ (y, x) \in R \longrightarrow y \notin Q\}$
assume $x \in Q$
obtain a **where** $a \in ?Q'$
by (*rule wfE-min* [*OF* ‹*wf R*› ‹$x \in Q$›]) *blast*
with ‹*wf S*›
obtain z **where** $z \in ?Q'$ **and** *zmin*: $\bigwedge y.\ (y, z) \in S \Longrightarrow y \notin ?Q'$ **by** (*erule wfE-min*)
{
fix y **assume** $(y, z) \in S$
then have $y \notin ?Q'$ **by** (*rule zmin*)

have $y \notin Q$
proof
assume $y \in Q$
with ‹$y \notin ?Q'$›
obtain w **where** $(w, y) \in R$ **and** $w \in Q$ **by** *auto*
from ‹$(w, y) \in R$› ‹$(y, z) \in S$› **have** $(w, z) \in S\ O\ R$ **by** (*rule rel-compI*)
with ‹$S\ O\ R \subseteq R$› **have** $(w, z) \in R$ **..**
with ‹$z \in ?Q'$› **have** $w \notin Q$ **by** *blast*
with ‹$w \in Q$› **show** *False* **by** *contradiction*
qed
}
with ‹$z \in ?Q'$› **show** $\exists z{\in}Q.\ \forall y.\ (y, z) \in R \cup S \longrightarrow y \notin Q$ **by** *blast*
qed

Well-foundedness of indexed union with disjoint domains and ranges

lemma *wf-UN*: [| *ALL i:I. wf(r i)*;
ALL i:I. ALL j:I. r i ~= r j --> Domain(r i) Int Range(r j) = {}
|] ==> *wf(UN i:I. r i)*
apply (*simp only*: *wf-eq-minimal*, *clarify*)
apply (*rename-tac A a*, *case-tac EX i:I. EX a:A. EX b:A. (b,a) : r i*)
prefer *2*
apply *force*
apply *clarify*
apply (*drule bspec*, *assumption*)
apply (*erule-tac x={a. a:A & (EX b:A. (b,a) : r i) } in allE*)
apply (*blast elim!: allE*)
done

lemmas *wfP-SUP = wf-UN* [**where** *I=UNIV* **and** *r=λi. {(x, y). r i x y}*,
to-pred SUP-UN-eq2 bot-empty-eq pred-equals-eq, *simplified*, *standard*]

lemma *wf-Union*:
[| *ALL r*:*R*. *wf r*;
ALL r:*R*. *ALL s*:*R*. *r* ~= *s* −−> *Domain r Int Range s* = {}
|] ==> *wf*(*Union R*)
apply (*simp add*: *Union-def*)
apply (*blast intro*: *wf-UN*)
done

lemma *wf-Un*:
[| *wf r*; *wf s*; *Domain r Int Range s* = {} |] ==> *wf*(*r Un s*)
using *wf-union-compatible*[*of s r*]
by (*auto simp*: *Un-ac*)

lemma *wf-union-merge*:
wf (*R* ∪ *S*) = *wf* (*R O R* ∪ *R O S* ∪ *S*) (**is** *wf ?A* = *wf ?B*)
proof
assume *wf ?A*
with *wf-trancl* **have** *wfT*: *wf* (*?A* ^+) **.**
moreover have *?B* ⊆ *?A* ^+
by (*subst trancl-unfold*, *subst trancl-unfold*) *blast*
ultimately show *wf ?B* **by** (*rule wf-subset*)
next
assume *wf ?B*

show *wf ?A*
proof (*rule wfI-min*)
fix *Q* :: *'a set* **and** *x*
assume *x* ∈ *Q*

with ‹*wf ?B*›
obtain *z* **where** *z* ∈ *Q* **and** ⋀*y*. (*y*, *z*) ∈ *?B* ⟹ *y* ∉ *Q*
by (*erule wfE-min*)
then have *A1*: ⋀*y*. (*y*, *z*) ∈ *R O R* ⟹ *y* ∉ *Q*
and *A2*: ⋀*y*. (*y*, *z*) ∈ *R O S* ⟹ *y* ∉ *Q*
and *A3*: ⋀*y*. (*y*, *z*) ∈ *S* ⟹ *y* ∉ *Q*
by *auto*

show ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *?A* ⟶ *y* ∉ *Q*
proof (*cases* ∀ *y*. (*y*, *z*) ∈ *R* ⟶ *y* ∉ *Q*)
case *True*
with ‹*z* ∈ *Q*› *A3* **show** *?thesis* **by** *blast*
next
case *False*
then obtain *z'* **where** *z'*∈*Q* (*z'*, *z*) ∈ *R* **by** *blast*

have ∀ *y*. (*y*, *z'*) ∈ *?A* ⟶ *y* ∉ *Q*
proof (*intro allI impI*)
fix *y* **assume** (*y*, *z'*) ∈ *?A*

```
      then show y ∉ Q
      proof
        assume (y, z') ∈ R
        then have (y, z) ∈ R O R using ‹(z', z) ∈ R› ..
        with A1 show y ∉ Q .
      next
        assume (y, z') ∈ S
        then have (y, z) ∈ R O S using  ‹(z', z) ∈ R› ..
        with A2 show y ∉ Q .
      qed
    qed
    with ‹z' ∈ Q› show ?thesis ..
  qed
 qed
qed

lemma wf-comp-self: wf R = wf (R O R)  — special case
 by (rule wf-union-merge [where S = {}, simplified])
```

22.3.1 acyclic

```
lemma acyclicI: ALL x. (x, x) ~: r^+ ==> acyclic r
 by (simp add: acyclic-def)

lemma wf-acyclic: wf r ==> acyclic r
apply (simp add: acyclic-def)
apply (blast elim: wf-trancl [THEN wf-irrefl])
done

lemmas wfP-acyclicP = wf-acyclic [to-pred]

lemma acyclic-insert [iff]:
    acyclic(insert (y,x) r) = (acyclic r & (x,y) ~: r^*)
apply (simp add: acyclic-def trancl-insert)
apply (blast intro: rtrancl-trans)
done

lemma acyclic-converse [iff]: acyclic(r^-1) = acyclic r
by (simp add: acyclic-def trancl-converse)

lemmas acyclicP-converse [iff] = acyclic-converse [to-pred]

lemma acyclic-impl-antisym-rtrancl: acyclic r ==> antisym(r^*)
apply (simp add: acyclic-def antisym-def)
apply (blast elim: rtranclE intro: rtrancl-into-trancl1 rtrancl-trancl-trancl)
done
```

lemma *acyclic-subset*: [| *acyclic s*; *r* <= *s* |] ==> *acyclic r*
apply (*simp add*: *acyclic-def*)
apply (*blast intro*: *trancl-mono*)
done

Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf* [*rule-format*]: *finite r* ==> *acyclic r* --> *wf r*
apply (*erule finite-induct*, *blast*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply *simp*
done

lemma *finite-acyclic-wf-converse*: [|*finite r*; *acyclic r*|] ==> *wf* (*r*^-1)
apply (*erule finite-converse* [*THEN iffD2*, *THEN finite-acyclic-wf*])
apply (*erule acyclic-converse* [*THEN iffD2*])
done

lemma *wf-iff-acyclic-if-finite*: *finite r* ==> *wf r* = *acyclic r*
by (*blast intro*: *finite-acyclic-wf wf-acyclic*)

22.4 Well-Founded Recursion

cut

lemma *cuts-eq*: (*cut f r x* = *cut g r x*) = (*ALL y*. (*y*,*x*):*r* --> *f*(*y*)=*g*(*y*))
by (*simp add*: *expand-fun-eq cut-def*)

lemma *cut-apply*: (*x*,*a*):*r* ==> (*cut f r a*)(*x*) = *f*(*x*)
by (*simp add*: *cut-def*)

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application"

lemma *wfrec-unique*: [| *adm-wf R F*; *wf R* |] ==> *EX*! *y*. *wfrec-rel R F x y*
apply (*simp add*: *adm-wf-def*)
apply (*erule-tac a*=*x* **in** *wf-induct*)
apply (*rule ex1I*)
apply (*rule-tac g* = %*x*. *THE y*. *wfrec-rel R F x y* **in** *wfrec-rel.wfrecI*)
apply (*fast dest*!: *theI'*)
apply (*erule wfrec-rel.cases*, *simp*)
apply (*erule allE*, *erule allE*, *erule allE*, *erule mp*)
apply (*fast intro*: *the-equality* [*symmetric*])
done

lemma *adm-lemma*: *adm-wf R* (%*f x*. *F* (*cut f R x*) *x*)
apply (*simp add*: *adm-wf-def*)
apply (*intro strip*)
apply (*rule cuts-eq* [*THEN iffD2*, *THEN subst*], *assumption*)
apply (*rule refl*)
done

lemma *wfrec*: *wf*(*r*) ==> *wfrec r H a* = *H* (*cut* (*wfrec r H*) *r a*) *a*
apply (*simp add*: *wfrec-def*)
apply (*rule adm-lemma* [*THEN wfrec-unique*, *THEN the1-equality*], *assumption*)
apply (*rule wfrec-rel.wfrecI*)
apply (*intro strip*)
apply (*erule adm-lemma* [*THEN wfrec-unique*, *THEN theI'*])
done

22.5 Code generator setup

consts-code
wfrec (⟨**module**⟩*wfrec?*)
attach ⟪
fun wfrec f x = *f* (*wfrec f*) *x*;
⟫

22.6 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

22.7 *nat* is well-founded

lemma *less-nat-rel*: *op* < = (λ*m n*. *n* = *Suc m*) ^++
proof (*rule ext*, *rule ext*, *rule iffI*)
fix *n m* :: *nat*
assume *m* < *n*
then show (λ*m n*. *n* = *Suc m*) ^++ *m n*
proof (*induct n*)
case *0* **then show** *?case* **by** *auto*
next
case (*Suc n*) **then show** *?case*
by (*auto simp add*: *less-Suc-eq-le le-less intro*: *tranclp.trancl-into-trancl*)
qed
next
fix *n m* :: *nat*
assume (λ*m n*. *n* = *Suc m*) ^++ *m n*
then show *m* < *n*
by (*induct n*)
(*simp-all add*: *less-Suc-eq-le reflexive le-less*)
qed

definition
pred-nat :: (*nat* ∗ *nat*) *set* **where**
pred-nat = {(*m*, *n*). *n* = *Suc m*}

definition
less-than :: (*nat* ∗ *nat*) *set* **where**
less-than = *pred-nat*^+

lemma *less-eq*: (*m*, *n*) ∈ *pred-nat*^+ ⟷ *m* < *n*
unfolding *less-nat-rel pred-nat-def trancl-def* **by** *simp*

lemma *pred-nat-trancl-eq-le*:
(*m*, *n*) ∈ *pred-nat*^∗ ⟷ *m* ≤ *n*
unfolding *less-eq rtrancl-eq-or-trancl* **by** *auto*

lemma *wf-pred-nat*: *wf pred-nat*
apply (*unfold wf-def pred-nat-def*, *clarify*)
apply (*induct-tac x*, *blast*+)
done

lemma *wf-less-than* [*iff*]: *wf less-than*
by (*simp add*: *less-than-def wf-pred-nat* [*THEN wf-trancl*])

lemma *trans-less-than* [*iff*]: *trans less-than*
by (*simp add*: *less-than-def trans-trancl*)

lemma *less-than-iff* [*iff*]: $((x,y)$: *less-than*$) = (x<y)$
by (*simp add*: *less-than-def less-eq*)

lemma *wf-less*: *wf* $\{(x, y::nat).\ x < y\}$
using *wf-less-than* **by** (*simp add*: *less-than-def less-eq* [*symmetric*])

Type *nat* is a wellfounded order

instance *nat* :: *wellorder*
by *intro-classes*
(*assumption* |
rule le-refl le-trans le-anti-sym nat-less-le nat-le-linear wf-less)+

LEAST theorems for type *nat*

lemma *Least-Suc*:
[| *P n*; ~ *P 0* |] ==> (*LEAST n. P n*) = *Suc* (*LEAST m. P*(*Suc m*))
apply (*case-tac n*, *auto*)
apply (*frule LeastI*)
apply (*drule-tac P* = %*x. P* (*Suc x*) **in** *LeastI*)
apply (*subgoal-tac* (*LEAST x. P x*) ≤ *Suc* (*LEAST x. P* (*Suc x*)))
apply (*erule-tac* [*2*] *Least-le*)
apply (*case-tac LEAST x. P x*, *auto*)
apply (*drule-tac P* = %*x. P* (*Suc x*) **in** *Least-le*)
apply (*blast intro*: *order-antisym*)
done

lemma *Least-Suc2*:
[|*P n*; *Q m*; ~*P 0*; !*k. P* (*Suc k*) = *Q k*|] ==> *Least P* = *Suc* (*Least Q*)
apply (*erule* (*1*) *Least-Suc* [*THEN ssubst*])
apply *simp*
done

lemma *ex-least-nat-le*: ¬*P*(*0*) ⟹ *P*(*n::nat*) ⟹ ∃ *k*≤*n*. (∀ *i*<*k*. ¬*P i*) & *P*(*k*)
apply (*cases n*)
apply *blast*
apply (*rule-tac x*=*LEAST k. P*(*k*) **in** *exI*)
apply (*blast intro*: *Least-le dest*: *not-less-Least intro*: *LeastI-ex*)
done

lemma *ex-least-nat-less*: ¬*P*(*0*) ⟹ *P*(*n::nat*) ⟹ ∃ *k*<*n*. (∀ *i*≤*k*. ¬*P i*) & *P*(*k+1*)
apply (*cases n*)
apply *blast*
apply (*frule* (*1*) *ex-least-nat-le*)
apply (*erule exE*)
apply (*case-tac k*)
apply *simp*
apply (*rename-tac k1*)
apply (*rule-tac x*=*k1* **in** *exI*)
apply *fastsimp*

done

22.8 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [**?**].

inductive-set
acc :: $('a * 'a)$ *set* => $'a$ *set*
for *r* :: $('a * 'a)$ *set*
where
accI: (!!*y*. (*y*, *x*) : *r* ==> *y* : *acc r*) ==> *x* : *acc r*

abbreviation
termip :: $('a$ => $'a$ => *bool*) => $'a$ => *bool* **where**
termip r == *accp* (r^{-1-1})

abbreviation
termi :: $('a * 'a)$ *set* => $'a$ *set* **where**
termi r == *acc* (r^{-1})

lemmas *accpI* = *accp.accI*

Induction rules

theorem *accp-induct*:
assumes *major*: *accp r a*
assumes *hyp*: !!*x*. *accp r x* ==> ∀ *y*. *r y x* --> *P y* ==> *P x*
shows *P a*
apply (*rule major* [*THEN accp.induct*])
apply (*rule hyp*)
apply (*rule accp.accI*)
apply *fast*
apply *fast*
done

theorems *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: *accp r b* ==> *r a b* ==> *accp r a*
apply (*erule accp.cases*)
apply *fast*
done

lemma *not-accp-down*:
assumes *na*: ¬ *accp R x*
obtains *z* **where** *R z x* **and** ¬ *accp R z*
proof −
assume *a*: ⋀*z*. ⟦*R z x*; ¬ *accp R z*⟧ ⟹ *thesis*

show *thesis*
proof (*cases* ∀ *z*. *R z x* ⟶ *accp R z*)
case *True*

```
    hence ⋀z. R z x ⟹ accp R z by auto
    hence accp R x
      by (rule accp.accI)
    with na show thesis ..
  next
    case False then obtain z where R z x and ¬ accp R z
      by auto
    with a show thesis .
  qed
qed

lemma accp-downwards-aux: r** b a ==> accp r a --> accp r b
  apply (erule rtranclp-induct)
   apply blast
  apply (blast dest: accp-downward)
  done

theorem accp-downwards: accp r a ==> r** b a ==> accp r b
  apply (blast dest: accp-downwards-aux)
  done

theorem accp-wfPI: ∀ 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 (rule predicate1I)
  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*: $D \leq accp\ R$
and *dcl*: $\bigwedge x\ z.\ [\![D\ x;\ R\ z\ x]\!] \Longrightarrow D\ z$
and $D\ x$
and *istep*: $\bigwedge x.\ [\![D\ x;\ (\bigwedge z.\ R\ z\ x \Longrightarrow P\ z)]\!] \Longrightarrow P\ x$
shows $P\ x$
proof −
from *subset* **and** ‹$D\ x$›
have $accp\ R\ x$ **..**
then show $P\ x$ **using** ‹$D\ x$›
proof (*induct x*)
fix x
assume $D\ x$
and $\bigwedge y.\ R\ y\ x \Longrightarrow D\ y \Longrightarrow P\ y$
with *dcl* **and** *istep* **show** $P\ x$ **by** *blast*
qed
qed

Set versions of the above theorems

lemmas *acc-induct* = *accp-induct* [*to-set*]

lemmas *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

lemmas *acc-downward* = *accp-downward* [*to-set*]

lemmas *not-acc-down* = *not-accp-down* [*to-set*]

lemmas *acc-downwards-aux* = *accp-downwards-aux* [*to-set*]

lemmas *acc-downwards* = *accp-downwards* [*to-set*]

lemmas *acc-wfI* = *accp-wfPI* [*to-set*]

lemmas *acc-wfD* = *accp-wfPD* [*to-set*]

lemmas *wf-acc-iff* = *wfP-accp-iff* [*to-set*]

lemmas *acc-subset* = *accp-subset* [*to-set pred-subset-eq*]

lemmas *acc-subset-induct* = *accp-subset-induct* [*to-set pred-subset-eq*]

22.9 Tools for building wellfounded relations

Inverse Image

lemma *wf-inv-image* [*simp*,*intro*!]: *wf*(*r*) ==> *wf*(*inv-image r* (*f*::'*a*=>'*b*))
apply (*simp* (*no-asm-use*) *add*: *inv-image-def wf-eq-minimal*)
apply *clarify*
apply (*subgoal-tac EX* (*w*::'*b*) . *w* : {*w*. *EX* (*x*::'*a*) . *x*: *Q* & (*f x* = *w*) })
prefer *2* **apply** (*blast del*: *allE*)
apply (*erule allE*)
apply (*erule* (*1*) *notE impE*)
apply *blast*
done

lemma *in-inv-image*[*simp*]: ((*x*,*y*) : *inv-image r f*) = ((*f x*, *f y*) : *r*)
by (*auto simp*:*inv-image-def*)

Measure functions into *nat*

definition *measure* :: ('*a* => *nat*) => ('*a* * '*a*)*set*
where *measure* == *inv-image less-than*

lemma *in-measure*[*simp*]: ((*x*,*y*) : *measure f*) = (*f x* < *f y*)
by (*simp add*:*measure-def*)

lemma *wf-measure* [*iff*]: *wf* (*measure f*)
apply (*unfold measure-def*)
apply (*rule wf-less-than* [*THEN wf-inv-image*])
done

Lexicographic combinations

definition
lex-prod :: [('*a**'*a*)*set*, ('*b**'*b*)*set*] => (('*a**'*b*)*('*a**'*b*))*set*
(**infixr** <**lex**> *80*)
where
ra <**lex**> *rb* == {((*a*,*b*),(*a*',*b*')). (*a*,*a*') : *ra* | *a*=*a*' & (*b*,*b*') : *rb*}

lemma *wf-lex-prod* [*intro*!]: [| *wf*(*ra*); *wf*(*rb*) |] ==> *wf*(*ra* <**lex**> *rb*)
apply (*unfold wf-def lex-prod-def*)
apply (*rule allI*, *rule impI*)
apply (*simp* (*no-asm-use*) *only*: *split-paired-All*)
apply (*drule spec*, *erule mp*)
apply (*rule allI*, *rule impI*)
apply (*drule spec*, *erule mp*, *blast*)
done

lemma *in-lex-prod*[*simp*]:
(((*a*,*b*),(*a*',*b*')): *r* <**lex**> *s*) = ((*a*,*a*'): *r* ∨ (*a* = *a*' ∧ (*b*, *b*') : *s*))
by (*auto simp*:*lex-prod-def*)

op <**lex**> preserves transitivity

lemma *trans-lex-prod* [*intro*!]:
  [| *trans R1*; *trans R2* |] ==> *trans* (*R1* <*lex*> *R2*)
by (*unfold trans-def lex-prod-def*, *blast*)

lexicographic combinations with measure functions

definition
  mlex-prod :: ('*a* ⇒ *nat*) ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* (**infixr** <*mlex*> *80*)
where
  f <*mlex*> *R* = *inv-image* (*less-than* <*lex*> *R*) (%*x*. (*f x*, *x*))

lemma *wf-mlex*: *wf R* ⟹ *wf* (*f* <*mlex*> *R*)
unfolding *mlex-prod-def*
by *auto*

lemma *mlex-less*: *f x* < *f y* ⟹ (*x*, *y*) ∈ *f* <*mlex*> *R*
unfolding *mlex-prod-def* **by** *simp*

lemma *mlex-leq*: *f x* ≤ *f y* ⟹ (*x*, *y*) ∈ *R* ⟹ (*x*, *y*) ∈ *f* <*mlex*> *R*
unfolding *mlex-prod-def* **by** *auto*

proper subset relation on finite sets

definition *finite-psubset* :: ('*a set* * '*a set*) *set*
where *finite-psubset* == {(*A*,*B*). *A* < *B* & *finite B*}

lemma *wf-finite-psubset*: *wf*(*finite-psubset*)
apply (*unfold finite-psubset-def*)
apply (*rule wf-measure* [*THEN wf-subset*])
apply (*simp add*: *measure-def inv-image-def less-than-def less-eq*)
apply (*fast elim*!: *psubset-card-mono*)
done

lemma *trans-finite-psubset*: *trans finite-psubset*
by (*simp add*: *finite-psubset-def less-le trans-def*, *blast*)

Wellfoundedness of *same-fst*

definition
 same-fst :: ('*a* => *bool*) => ('*a* => ('*b* * '*b*)*set*) => (('*a**'*b*)*('*a**'*b*))*set*
where
    same-fst P R == {((*x'*,*y'*),(*x*,*y*)) . *x'*=*x* & *P x* & (*y'*,*y*) : *R x*}
  — For *rec-def* declarations where the first n parameters stay unchanged in the recursive call. See *Library/While-Combinator.thy* for an application.

lemma *same-fstI* [*intro*!]:
    [| *P x*; (*y'*,*y*) : *R x* |] ==> ((*x*,*y'*),(*x*,*y*)) : *same-fst P R*
by (*simp add*: *same-fst-def*)

lemma *wf-same-fst*:
  assumes *prem*: (!!*x*. *P x* ==> *wf*(*R x*))
  shows *wf*(*same-fst P R*)

apply (*simp cong del*: *imp-cong add*: *wf-def same-fst-def*)
apply (*intro strip*)
apply (*rename-tac a b*)
apply (*case-tac wf* (*R a*))
apply (*erule-tac a = b* **in** *wf-induct*, *blast*)
apply (*blast intro*: *prem*)
done

22.10 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.

This material does not appear to be used any longer.

lemma *lemma1*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^* |] ==> (*f* (*i*+*k*), *f i*) : *r*^*
apply (*induct-tac k*, *simp-all*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *lemma2*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^*; *wf* (*r*^+) |]
    ==> *ALL m*. *f m* = *x* --> (*EX i*. *ALL k*. *f* (*m*+*i*+*k*) = *f* (*m*+*i*))
apply (*erule wf-induct*, *clarify*)
apply (*case-tac EX j*. (*f* (*m*+*j*), *f m*) : *r*^+)
apply *clarify*
apply (*subgoal-tac EX i*. *ALL k*. *f* ((*m*+*j*) +*i*+*k*) = *f* ((*m*+*j*) +*i*))
apply *clarify*
apply (*rule-tac x* = *j*+*i* **in** *exI*)
apply (*simp add*: *add-ac*, *blast*)
apply (*rule-tac x* = *0* **in** *exI*, *clarsimp*)
apply (*drule-tac i* = *m* **and** *k* = *k* **in** *lemma1*)
apply (*blast elim*: *rtranclE dest*: *rtrancl-into-trancl1*)
done

lemma *wf-weak-decr-stable*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^*; *wf* (*r*^+) |]
    ==> *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
apply (*drule-tac x* = *0* **in** *lemma2* [*THEN spec*], *auto*)
done

lemma *weak-decr-stable*:
    ALL i. *f* (*Suc i*) <= ((*f i*)::*nat*) ==> *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
apply (*rule-tac r* = *pred-nat* **in** *wf-weak-decr-stable*)
apply (*simp add*: *pred-nat-trancl-eq-le*)
apply (*intro wf-trancl wf-pred-nat*)
done

22.11 size of a datatype value

use *Tools/function-package/size.ML*

setup *Size.setup*

lemma *nat-size* [*simp*, *code func*]: *size* (*n*::*nat*) = *n*
by (*induct n*) *simp-all*

end

23 Int: The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *Int*
imports *Equiv-Relations Nat Wellfounded*
uses
(*Tools/numeral.ML*)
(*Tools/numeral-syntax.ML*)
(~~/*src/Provers/Arith/assoc-fold.ML*)
~~/*src/Provers/Arith/cancel-numerals.ML*
~~/*src/Provers/Arith/combine-numerals.ML*
(*int-arith1.ML*)
begin

23.1 The equivalence relation underlying the integers

definition
intrel :: ((*nat* × *nat*) × (*nat* × *nat*)) *set*
where
intrel = {((*x*, *y*), (*u*, *v*)) | *x y u v*. *x* + *v* = *u* + *y* }

typedef (*Integ*)
int = *UNIV*//*intrel*
by (*auto simp add*: *quotient-def*)

instantiation *int* :: {*zero*, *one*, *plus*, *minus*, *uminus*, *times*, *ord*, *abs*, *sgn*}
begin

definition
Zero-int-def [*code func del*]: *0* = *Abs-Integ* (*intrel* `` {(*0*, *0*)})

definition
One-int-def [*code func del*]: *1* = *Abs-Integ* (*intrel* `` {(*1*, *0*)})

definition
add-int-def [*code func del*]: *z* + *w* = *Abs-Integ*
(⋃(*x*, *y*) ∈ *Rep-Integ z*. ⋃(*u*, *v*) ∈ *Rep-Integ w*.
intrel `` {(*x* + *u*, *y* + *v*)})

definition

minus-int-def [*code func del*]:
$- z = \mathit{Abs\text{-}Integ}\ (\bigcup(x, y) \in \mathit{Rep\text{-}Integ}\ z.\ \mathit{intrel}\ ``\ \{(y, x)\})$

definition
diff-int-def [*code func del*]: $z - w = z + (-w :: \mathit{int})$

definition
mult-int-def [*code func del*]: $z * w = \mathit{Abs\text{-}Integ}$
$(\bigcup(x, y) \in \mathit{Rep\text{-}Integ}\ z.\ \bigcup(u,v) \in \mathit{Rep\text{-}Integ}\ w.$
$\mathit{intrel}\ ``\ \{(x*u + y*v, x*v + y*u)\})$

definition
le-int-def [*code func del*]:
$z \leq w \longleftrightarrow (\exists x\ y\ u\ v.\ x+v \leq u+y \wedge (x, y) \in \mathit{Rep\text{-}Integ}\ z \wedge (u, v) \in \mathit{Rep\text{-}Integ}\ w)$

definition
less-int-def [*code func del*]: $(z::\mathit{int}) < w \longleftrightarrow z \leq w \wedge z \neq w$

definition
zabs-def: $|i::\mathit{int}| = (\mathit{if}\ i < 0\ \mathit{then} - i\ \mathit{else}\ i)$

definition
zsgn-def: $\mathit{sgn}\ (i::\mathit{int}) = (\mathit{if}\ i=0\ \mathit{then}\ 0\ \mathit{else\ if}\ 0<i\ \mathit{then}\ 1\ \mathit{else} - 1)$

instance ..

end

23.2 Construction of the Integers

lemma *intrel-iff* [*simp*]: $(((x,y),(u,v)) \in \mathit{intrel}) = (x+v = u+y)$
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: $(\mathit{intrel}\ ``\ \{x\} = \mathit{intrel}\ ``\ \{y\}) = ((x, y) \in \mathit{intrel})$

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*]: $\mathit{intrel}``\{(x,y)\} \in \mathit{Integ}$
by (*auto simp add*: *Integ-def intrel-def quotient-def*)

Reduces equality on abstractions to equality on representatives: $[\![x \in \mathit{Integ}; y \in \mathit{Integ}]\!] \Longrightarrow (\mathit{Abs\text{-}Integ}\ x = \mathit{Abs\text{-}Integ}\ y) = (x = y)$

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-Integ*(*intrel*``{(*x*,*y*)}) ==> *P*) ==> *P*
apply (*rule Abs-Integ-cases* [*of z*])
apply (*auto simp add*: *Integ-def quotient-def*)
done

23.3 Arithmetic Operations

lemma *minus*: − *Abs-Integ*(*intrel*``{(*x*,*y*)}) = *Abs-Integ*(*intrel* `` {(*y*,*x*)})
proof −
have (λ(*x*,*y*). *intrel*``{(*y*,*x*)}) *respects intrel*
by (*simp add*: *congruent-def*)
thus *?thesis*
by (*simp add*: *minus-int-def UN-equiv-class* [*OF equiv-intrel*])
qed

lemma *add*:
Abs-Integ (*intrel*``{(*x*,*y*)}) + *Abs-Integ* (*intrel*``{(*u*,*v*)}) =
Abs-Integ (*intrel*``{(*x*+*u*, *y*+*v*)})
proof −
have (λ*z w*. (λ(*x*,*y*). (λ(*u*,*v*). *intrel* `` {(*x*+*u*, *y*+*v*)}) *w*) *z*)
respects2 intrel
by (*simp add*: *congruent2-def*)
thus *?thesis*
by (*simp add*: *add-int-def UN-UN-split-split-eq*
UN-equiv-class2 [*OF equiv-intrel equiv-intrel*])
qed

Congruence property for multiplication

lemma *mult-congruent2*:
(%*p1 p2*. (%(*x*,*y*). (%(*u*,*v*). *intrel*``{(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)}) *p2*) *p1*)
respects2 intrel
apply (*rule equiv-intrel* [*THEN congruent2-commuteI*])
apply (*force simp add*: *mult-ac*, *clarify*)
apply (*simp add*: *congruent-def mult-ac*)
apply (*rename-tac u v w x y z*)
apply (*subgoal-tac u*∗*y* + *x*∗*y* = *w*∗*y* + *v*∗*y* & *u*∗*z* + *x*∗*z* = *w*∗*z* + *v*∗*z*)
apply (*simp add*: *mult-ac*)
apply (*simp add*: *add-mult-distrib* [*symmetric*])
done

lemma *mult*:
Abs-Integ((*intrel*``{(*x*,*y*)})) ∗ *Abs-Integ*((*intrel*``{(*u*,*v*)})) =
Abs-Integ(*intrel* `` {(*x*∗*u* + *y*∗*v*, *x*∗*v* + *y*∗*u*)})
by (*simp add*: *mult-int-def UN-UN-split-split-eq mult-congruent2*
UN-equiv-class2 [*OF equiv-intrel equiv-intrel*])

The integers form a *comm-ring-1*

instance *int* :: *comm-ring-1*
proof
  fix *i j k* :: *int*
  show $(i + j) + k = i + (j + k)$
    by (*cases i*, *cases j*, *cases k*) (*simp add*: *add add-assoc*)
  show $i + j = j + i$
    by (*cases i*, *cases j*) (*simp add*: *add-ac add*)
  show $0 + i = i$
    by (*cases i*) (*simp add*: *Zero-int-def add*)
  show $- i + i = 0$
    by (*cases i*) (*simp add*: *Zero-int-def minus add*)
  show $i - j = i + - j$
    by (*simp add*: *diff-int-def*)
  show $(i * j) * k = i * (j * k)$
    by (*cases i*, *cases j*, *cases k*) (*simp add*: *mult 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 $0 \neq (1::int)$
    by (*simp add*: *Zero-int-def One-int-def*)
qed

lemma *int-def*: *of-nat m* = *Abs-Integ* (*intrel* `` {(*m*, *0*)})
by (*induct m*, *simp-all add*: *Zero-int-def One-int-def add*)

23.4 The $\leq$ Ordering

lemma *le*:
  (*Abs-Integ*(*intrel*``{(*x*,*y*)}) $\leq$ *Abs-Integ*(*intrel*``{(*u*,*v*)})) = ($x+v \leq u+y$)
by (*force simp add*: *le-int-def*)

lemma *less*:
  (*Abs-Integ*(*intrel*``{(*x*,*y*)}) < *Abs-Integ*(*intrel*``{(*u*,*v*)})) = ($x+v < u+y$)
by (*simp add*: *less-int-def le order-less-le*)

instance *int* :: *linorder*
proof
  fix *i j k* :: *int*
  show $(i < j) = (i \leq j \wedge i \neq j)$
    by (*simp add*: *less-int-def*)
  show $i \leq i$
    by (*cases i*) (*simp add*: *le*)
  show $i \leq j \Longrightarrow j \leq k \Longrightarrow i \leq k$
    by (*cases i*, *cases j*, *cases k*) (*simp add*: *le*)
  show $i \leq j \Longrightarrow j \leq i \Longrightarrow i = j$

by (*cases i*, *cases j*) (*simp add*: *le*)
show $i \leq j \vee j \leq i$
by (*cases i*, *cases j*) (*simp add*: *le linorder-linear*)
qed

instantiation *int* :: *distrib-lattice*
begin

definition
(*inf* :: *int* ⇒ *int* ⇒ *int*) = *min*

definition
(*sup* :: *int* ⇒ *int* ⇒ *int*) = *max*

instance
by *intro-classes*
(*auto simp add*: *inf-int-def sup-int-def min-max.sup-inf-distrib1*)

end

instance *int* :: *pordered-cancel-ab-semigroup-add*
proof
fix *i j k* :: *int*
show $i \leq j \Longrightarrow k + i \leq k + j$
by (*cases i*, *cases j*, *cases k*) (*simp add*: *le add*)
qed

Strict Monotonicity of Multiplication

strict, in 1st argument; proof is by induction on k¿0

lemma *zmult-zless-mono2-lemma*:
(*i*::*int*)<*j* ==> 0<*k* ==> *of-nat k* * *i* < *of-nat k* * *j*
apply (*induct k*, *simp*)
apply (*simp add*: *left-distrib*)
apply (*case-tac k=0*)
apply (*simp-all add*: *add-strict-mono*)
done

lemma *zero-le-imp-eq-int*: (0::*int*) ≤ *k* ==> ∃ *n*. *k* = *of-nat n*
apply (*cases k*)
apply (*auto simp add*: *le add int-def Zero-int-def*)
apply (*rule-tac x=x−y* **in** *exI*, *simp*)
done

lemma *zero-less-imp-eq-int*: (0::*int*) < *k* ==> ∃ *n*>0. *k* = *of-nat n*
apply (*cases k*)
apply (*simp add*: *less int-def Zero-int-def*)
apply (*rule-tac x=x−y* **in** *exI*, *simp*)
done

lemma *zmult-zless-mono2*: [| *i*<*j*; (*0*::*int*) < *k* |] ==> *k*∗*i* < *k*∗*j*
apply (*drule zero-less-imp-eq-int*)
apply (*auto simp add*: *zmult-zless-mono2-lemma*)
done

The integers form an ordered integral domain

instance *int* :: *ordered-idom*
proof
  fix *i j k* :: *int*
  show *i* < *j* ⟹ *0* < *k* ⟹ *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

instance *int* :: *lordered-ring*
proof
  fix *k* :: *int*
  show *abs k* = *sup k* (− *k*)
   by (*auto simp add*: *sup-int-def zabs-def max-def less-minus-self-iff* [*symmetric*])
qed

lemma *zless-imp-add1-zle*: *w* < *z* ⟹ *w* + (*1*::*int*) ≤ *z*
apply (*cases w*, *cases z*)
apply (*simp add*: *less le add One-int-def*)
done

lemma *zless-iff-Suc-zadd*:
  (*w* :: *int*) < *z* ⟷ (∃ *n*. *z* = *w* + *of-nat* (*Suc n*))
apply (*cases z*, *cases w*)
apply (*auto simp add*: *less add int-def*)
apply (*rename-tac a b c d*)
apply (*rule-tac x*=*a*+*d* − *Suc*(*c*+*b*) **in** *exI*)
apply *arith*
done

lemmas *int-distrib* =
  left-distrib [*of z1*::*int z2 w*, *standard*]
  right-distrib [*of w*::*int z1 z2*, *standard*]
  left-diff-distrib [*of z1*::*int z2 w*, *standard*]
  right-diff-distrib [*of w*::*int z1 z2*, *standard*]

23.5 Embedding of the Integers into any *ring-1*: *of-int*

context *ring-1*
begin

definition
of-int :: *int* ⇒ *'a*
where
[*code func del*]: *of-int z* = *contents* (⋃(*i*, *j*) ∈ *Rep-Integ z*. { *of-nat i* − *of-nat j* })

lemma *of-int*: *of-int* (*Abs-Integ* (*intrel* `` {(*i*,*j*)})) = *of-nat i* − *of-nat j*
proof −
have (λ(*i*,*j*). { *of-nat i* − (*of-nat j* :: *'a*) }) *respects intrel*
by (*simp add*: *congruent-def 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 OrderedGroup.compare-rls add*)

lemma *of-int-minus* [*simp*]: *of-int* (−*z*) = − (*of-int z*)
by (*cases z*, *simp add*: *compare-rls of-int minus*)

lemma *of-int-diff* [*simp*]: *of-int* (*w* − *z*) = *of-int w* − *of-int z*
by (*simp add*: *OrderedGroup.diff-minus diff-minus*)

lemma *of-int-mult* [*simp*]: *of-int* (*w*∗*z*) = *of-int w* ∗ *of-int z*
apply (*cases w*, *cases z*)
apply (*simp add*: *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* (*of-nat n*) = *of-nat n*
by (*induct n*) *auto*

end

context *ordered-idom*
begin

lemma *of-int-le-iff* [*simp*]:
of-int w ≤ *of-int z* ⟷ *w* ≤ *z*
by (*cases w*, *cases z*, *simp add*: *of-int le minus compare-rls of-nat-add* [*symmetric*]
del: *of-nat-add*)

Special cases where either operand is zero

lemmas *of-int-0-le-iff* [*simp*] = *of-int-le-iff* [*of 0*, *simplified*]
lemmas *of-int-le-0-iff* [*simp*] = *of-int-le-iff* [*of - 0*, *simplified*]

lemma *of-int-less-iff* [*simp*]:
of-int w < *of-int z* ⟷ *w* < *z*
by (*simp add*: *not-le* [*symmetric*] *linorder-not-le* [*symmetric*])

Special cases where either operand is zero

lemmas *of-int-0-less-iff* [*simp*] = *of-int-less-iff* [*of 0*, *simplified*]
lemmas *of-int-less-0-iff* [*simp*] = *of-int-less-iff* [*of - 0*, *simplified*]

end

Class for unital rings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *ring-char-0* = *ring-1* + *semiring-char-0*
begin

lemma *of-int-eq-iff* [*simp*]:
of-int w = *of-int z* ⟷ *w* = *z*
apply (*cases w*, *cases z*, *simp add*: *of-int*)
apply (*simp only*: *diff-eq-eq diff-add-eq eq-diff-eq*)
apply (*simp only*: *of-nat-add* [*symmetric*] *of-nat-eq-iff*)
done

Special cases where either operand is zero

lemmas *of-int-0-eq-iff* [*simp*] = *of-int-eq-iff* [*of 0*, *simplified*]
lemmas *of-int-eq-0-iff* [*simp*] = *of-int-eq-iff* [*of - 0*, *simplified*]

end

Every *ordered-idom* has characteristic zero.

subclass (**in** *ordered-idom*) *ring-char-0* **by** *intro-locales*

lemma *of-int-eq-id* [*simp*]: *of-int* = *id*
proof
fix *z* **show** *of-int z* = *id z*
by (*cases z*) (*simp add*: *of-int add minus int-def diff-minus*)
qed

23.6 Magnitude of an Integer, as a Natural Number: *nat*

definition
nat :: *int* ⇒ *nat*
where
[*code func del*]: *nat z* = *contents* (⋃(*x*, *y*) ∈ *Rep-Integ z*. {*x*−*y*})

lemma *nat*: *nat* (*Abs-Integ* (*intrel*``{(*x*,*y*)})) = *x*−*y*
proof −
  have (λ(*x*,*y*). {*x*−*y*}) *respects intrel*
    by (*simp add*: *congruent-def*) *arith*
  thus *?thesis*
    by (*simp add*: *nat-def UN-equiv-class* [*OF equiv-intrel*])
qed

lemma *nat-int* [*simp*]: *nat* (*of-nat n*) = *n*
by (*simp add*: *nat int-def*)

lemma *nat-zero* [*simp*]: *nat 0* = *0*
by (*simp add*: *Zero-int-def nat*)

lemma *int-nat-eq* [*simp*]: *of-nat* (*nat z*) = (*if 0* ≤ *z then z else 0*)
by (*cases z*, *simp add*: *nat le int-def Zero-int-def*)

corollary *nat-0-le*: *0* ≤ *z* ==> *of-nat* (*nat z*) = *z*
by *simp*

lemma *nat-le-0* [*simp*]: *z* ≤ *0* ==> *nat z* = *0*
by (*cases z*, *simp add*: *nat le Zero-int-def*)

lemma *nat-le-eq-zle*: *0* < *w* | *0* ≤ *z* ==> (*nat w* ≤ *nat z*) = (*w*≤*z*)
apply (*cases w*, *cases z*)
apply (*simp add*: *nat le linorder-not-le* [*symmetric*] *Zero-int-def*, *arith*)
done

An alternative condition is (*0*::′*a*) ≤ *w*

corollary *nat-mono-iff*: *0* < *z* ==> (*nat w* < *nat z*) = (*w* < *z*)
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

corollary *nat-less-eq-zless*: *0* ≤ *w* ==> (*nat w* < *nat z*) = (*w*<*z*)
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

lemma *zless-nat-conj* [*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* ≤ *z* **and** ⋀*m*. *z* = *of-nat m* ⟹ *P*
  shows *P*
  using *assms* **by** (*blast dest*: *nat-0-le sym*)

lemma *nat-eq-iff*: (*nat w* = *m*) = (*if 0* ≤ *w then w* = *of-nat m else m*=*0*)
by (*cases w*, *simp add*: *nat le int-def Zero-int-def*, *arith*)

corollary *nat-eq-iff2*: $(m = nat\ w) = (if\ 0 \leq w\ then\ w = of\text{-}nat\ m\ else\ m{=}0)$
by (*simp only*: *eq-commute* [*of m*] *nat-eq-iff*)

lemma *nat-less-iff*: $0 \leq w \Longrightarrow (nat\ w < m) = (w < of\text{-}nat\ m)$
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*:
$[\!|\ (0{::}int) \leq z;\ \ 0 \leq z'\ |\!] \Longrightarrow nat\ (z{+}z') = nat\ z + nat\ z'$
by (*cases z*, *cases z'*, *simp add*: *nat add le Zero-int-def*)

lemma *nat-diff-distrib*:
$[\!|\ (0{::}int) \leq z';\ \ z' \leq z\ |\!] \Longrightarrow nat\ (z{-}z') = nat\ z - nat\ z'$
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*)

context *ring-1*
begin

lemma *of-nat-nat*: $0 \leq z \Longrightarrow of\text{-}nat\ (nat\ z) = of\text{-}int\ z$
by (*cases z rule*: *eq-Abs-Integ*)
(*simp add*: *nat le of-int Zero-int-def of-nat-diff*)

end

23.7 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-nat* $n \leq$ (*of-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-nat* (*Suc* n) :: *int*))
by (*subst le-minus-iff*, *simp del*: *of-nat-Suc*)

lemma *int-zle-neg*: ((*of-nat* n :: *int*) $\leq -$ *of-nat* m) = (n = 0 & m = 0)
by (*simp add*: *int-def le minus Zero-int-def*)

lemma *not-int-zless-negative* [*simp*]: ~ ((*of-nat* n :: *int*) $<$ $-$ *of-nat* m)
by (*simp add*: *linorder-not-less*)

lemma *negative-eq-positive* [*simp*]: (($-$ *of-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*) $\leq z \longleftrightarrow$ ($\exists n.\ z = w +$ *of-nat* n)
proof $-$
  have ($w \leq z$) = (0 $\leq z - w$)
    by (*simp only*: *le-diff-eq add-0-left*)
  also have ... = ($\exists n.\ z - w =$ *of-nat* n)
    by (*auto elim*: *zero-le-imp-eq-int*)
  also have ... = ($\exists n.\ z = w +$ *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 $\leq a$ −−> $P\ a$) & ($a < 0$ −−> $P(-a)$))
by (*force dest*: *order-less-le-trans simp add*: *abs-if linorder-not-less*)

lemma *negD*: (x :: *int*) $<$ 0 $\Longrightarrow \exists n.\ x = -$ (*of-nat* (*Suc* n))
apply (*cases* x)
apply (*auto simp add*: *le minus Zero-int-def int-def order-less-le*)
apply (*rule-tac* x=y $-$ *Suc* x **in** *exI*, *arith*)
done

23.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

theorem *int-cases* [*cases type*: *int*, *case-names nonneg neg*]:
[|!! *n*. (*z* :: *int*) = *of-nat n* ==> *P*; !! *n*. *z* = − (*of-nat* (*Suc n*)) ==> *P* |] ==> *P*
apply (*cases z* < *0*, *blast dest*!: *negD*)
apply (*simp add*: *linorder-not-less del*: *of-nat-Suc*)
apply *auto*
apply (*blast dest*: *nat-0-le* [*THEN sym*])
done

theorem *int-induct* [*induct type*: *int*, *case-names nonneg neg*]:
[|!! *n*. *P* (*of-nat n* :: *int*); !!*n*. *P* (− (*of-nat* (*Suc n*))) |] ==> *P z*
by (*cases z rule*: *int-cases*) *auto*

Contributed by Brian Huffman

theorem *int-diff-cases*:
obtains (*diff*) *m n* **where** (*z*::*int*) = *of-nat m* − *of-nat n*
apply (*cases z rule*: *eq-Abs-Integ*)
apply (*rule-tac m*=*x* **and** *n*=*y* **in** *diff*)
apply (*simp add*: *int-def diff-def minus add*)
done

23.9 Binary representation

This formalization defines binary arithmetic in terms of the integers rather than using a datatype. This avoids multiple representations (leading zeroes, etc.) See *ZF/Tools/twos−compl.ML*, function *int-of-binary*, for the numerical interpretation.

The representation expects that (*m mod 2*) is 0 or 1, even if m is negative; For instance, −*5 div 2* = −*3* and −*5 mod 2* = *1*; thus −*5* = (−*3*)∗*2* + *1*.

This two's complement binary representation derives from the paper "An Efficient Representation of Arithmetic for Term Rewriting" by Dave Cohen and Phil Watson, Rewriting Techniques and Applications, Springer LNCS 488 (240-251), 1991.

definition
Pls :: *int* **where**
[*code func del*]: *Pls* = *0*

definition
Min :: *int* **where**
[*code func del*]: *Min* = − *1*

definition

Bit0 :: *int* ⇒ *int* **where**
[*code func del*]: *Bit0 k* = *k* + *k*

definition
Bit1 :: *int* ⇒ *int* **where**
[*code func del*]: *Bit1 k* = *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* (*Bit1 Pls*)

lemma *Let-number-of* [*simp*]: *Let* (*number-of v*) *f* = *f* (*number-of v*)
— Unfold all *let*s involving constants
unfolding *Let-def* **..**

definition
succ :: *int* ⇒ *int* **where**
[*code 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 Bit0-def Bit1-def

Removal of leading zeroes

lemma *Bit0-Pls* [*simp*, *code post*]:

Bit0 Pls = Pls
unfolding *numeral-simps* **by** *simp*

lemma *Bit1-Min* [*simp*, *code post*]:
Bit1 Min = Min
unfolding *numeral-simps* **by** *simp*

lemmas *normalize-bin-simps* =
Bit0-Pls Bit1-Min

23.10 The Functions *succ*, *pred* and *uminus*

lemma *succ-Pls* [*simp*]:
succ Pls = Bit1 Pls
unfolding *numeral-simps* **by** *simp*

lemma *succ-Min* [*simp*]:
succ Min = Pls
unfolding *numeral-simps* **by** *simp*

lemma *succ-Bit0* [*simp*]:
succ (Bit0 k) = Bit1 k
unfolding *numeral-simps* **by** *simp*

lemma *succ-Bit1* [*simp*]:
succ (Bit1 k) = Bit0 (succ k)
unfolding *numeral-simps* **by** *simp*

lemmas *succ-bin-simps* =
succ-Pls succ-Min succ-Bit0 succ-Bit1

lemma *pred-Pls* [*simp*]:
pred Pls = Min
unfolding *numeral-simps* **by** *simp*

lemma *pred-Min* [*simp*]:
pred Min = Bit0 Min
unfolding *numeral-simps* **by** *simp*

lemma *pred-Bit0* [*simp*]:
pred (Bit0 k) = Bit1 (pred k)
unfolding *numeral-simps* **by** *simp*

lemma *pred-Bit1* [*simp*]:
pred (Bit1 k) = Bit0 k
unfolding *numeral-simps* **by** *simp*

lemmas *pred-bin-simps* =
pred-Pls pred-Min pred-Bit0 pred-Bit1

lemma *minus-Pls* [*simp*]:
− Pls = Pls
unfolding *numeral-simps* **by** *simp*

lemma *minus-Min* [*simp*]:
− Min = Bit1 Pls
unfolding *numeral-simps* **by** *simp*

lemma *minus-Bit0* [*simp*]:
− (Bit0 k) = Bit0 (− k)
unfolding *numeral-simps* **by** *simp*

lemma *minus-Bit1* [*simp*]:
− (Bit1 k) = Bit1 (pred (− k))
unfolding *numeral-simps* **by** *simp*

lemmas *minus-bin-simps =*
minus-Pls minus-Min minus-Bit0 minus-Bit1

23.11 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-Bit0-Bit0* [*simp*]:
(Bit0 k) + (Bit0 l) = Bit0 (k + l)
unfolding *numeral-simps* **by** *simp-all*

lemma *add-Bit0-Bit1* [*simp*]:
(Bit0 k) + (Bit1 l) = Bit1 (k + l)
unfolding *numeral-simps* **by** *simp-all*

lemma *add-Bit1-Bit0* [*simp*]:
(Bit1 k) + (Bit0 l) = Bit1 (k + l)
unfolding *numeral-simps* **by** *simp*

lemma *add-Bit1-Bit1* [*simp*]:
(Bit1 k) + (Bit1 l) = Bit0 (k + succ l)
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*

lemmas *add-bin-simps* =
add-Pls add-Min add-Pls-right add-Min-right
add-Bit0-Bit0 add-Bit0-Bit1 add-Bit1-Bit0 add-Bit1-Bit1

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-Bit0* [*simp*]:
*(Bit0 k) * l = Bit0 (k * l)*
unfolding *numeral-simps int-distrib* **by** *simp*

lemma *mult-Bit1* [*simp*]:
*(Bit1 k) * l = (Bit0 (k * l)) + l*
unfolding *numeral-simps int-distrib* **by** *simp*

lemmas *mult-bin-simps* =
mult-Pls mult-Min mult-Bit0 mult-Bit1

23.12 Converting Numerals to Rings: *number-of*

class *number-ring* = *number* + *comm-ring-1* +
assumes *number-of-eq*: *number-of k = of-int k*

self-embedding of the integers

instantiation *int* :: *number-ring*
begin

definition
int-number-of-def [*code func del*]: *number-of w = (of-int w :: int)*

instance
by *intro-classes* (*simp only*: *int-number-of-def*)

end

lemma *number-of-is-id*:
number-of (k::int) = k
unfolding *int-number-of-def* **by** *simp*

lemma *number-of-succ*:
number-of (*succ k*) = (*1* + *number-of k* ::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-pred*:
number-of (*pred w*) = (− *1* + *number-of w* ::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-minus*:
number-of (*uminus w*) = (− (*number-of w*)::'*a*::*number-ring*)
unfolding *number-of-eq 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-Bit0*:
(*1* + *1*) ∗ *number-of w* = (*number-of* (*Bit0 w*) ::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps left-distrib* **by** *simp*

Converting numerals 0 and 1 to their abstract versions.

lemma *numeral-0-eq-0* [*simp*]:
Numeral0 = (*0*::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *numeral-1-eq-1* [*simp*]:
Numeral1 = (*1*::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

Special-case simplification for small constants.

Unary minus for the abstract constant 1. Cannot be inserted as a simprule until later: it is *number-of-Min* re-oriented!

lemma *numeral-m1-eq-minus-1*:
(*−1*::'*a*::*number-ring*) = − *1*
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *mult-minus1* [*simp*]:
−1 ∗ *z* = −(*z*::'*a*::*number-ring*)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *mult-minus1-right* [*simp*]:
z ∗ *−1* = −(*z*::'*a*::*number-ring*)

unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *minus-number-of-mult* [*simp*]:
− (number-of w) ∗ z = number-of (uminus w) ∗ (z::'a::number-ring)
unfolding *number-of-eq* **by** *simp*

Subtraction

lemma *diff-number-of-eq*:
number-of v − number-of w =
(number-of (v + uminus w)::'a::number-ring)
unfolding *number-of-eq* **by** *simp*

lemma *number-of-Pls*:
number-of Pls = (0::'a::number-ring)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-Min*:
number-of Min = (− 1::'a::number-ring)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-Bit0*:
number-of (Bit0 w) = (0::'a::number-ring) + (number-of w) + (number-of w)
unfolding *number-of-eq numeral-simps* **by** *simp*

lemma *number-of-Bit1*:
number-of (Bit1 w) = (1::'a::number-ring) + (number-of w) + (number-of w)
unfolding *number-of-eq numeral-simps* **by** *simp*

23.13 Equality of Binary Numbers

First version by Norbert Voelker

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*: $(\sim neg\ x) = (0 \leq x)$
by (*simp add*: *neg-def linorder-not-less*)

To simplify inequalities when Numeral1 can get simplified to 1

lemma *not-neg-0*: $\sim\ neg\ 0$
by (*simp add*: *One-int-def neg-def*)

lemma *not-neg-1*: $\sim\ neg\ 1$
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*: $\sim$ *iszero 1*
by (*simp add*: *iszero-def eq-commute*)

lemma *neg-nat*: *neg z* $\Longrightarrow$ *nat z* = *0*
by (*simp add*: *neg-def order-less-imp-le*)

lemma *not-neg-nat*: $\sim$ *neg z* $\Longrightarrow$ *of-nat* (*nat z*) = *z*
by (*simp add*: *linorder-not-less neg-def*)

lemma *eq-number-of-eq*:
(($number\text{-}of\ x :: 'a :: number\text{-}ring$) = *number-of y*) =
iszero (*number-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*:
$\sim$ *iszero* (($number\text{-}of\ Min) :: 'a :: number\text{-}ring$)
unfolding *iszero-def numeral-m1-eq-minus-1* **by** *simp*

23.14 Comparisons, for Ordered Rings

lemmas *double-eq-0-iff* = *double-zero*

lemma *le-imp-0-less*:
assumes *le*: *0* $\leq$ *z*
shows (*0*::*int*) < *1* + *z*
proof −
have *0* $\leq$ *z* **by** *fact*
also have ... < *z* + *1* **by** (*rule less-add-one*)
also have ... = *1* + *z* **by** (*simp add*: *add-ac*)
finally show *0* < *1* + *z* **.**

qed

lemma *odd-nonzero*:
1 + z + z ≠ (0::int)
proof (*cases z rule*: *int-cases*)
case (*nonneg n*)
have *le*: *0 ≤ z+z* **by** (*simp add*: *nonneg add-increasing*)
thus *?thesis* **using** *le-imp-0-less* [*OF le*]
by (*auto simp add*: *add-assoc*)
next
case (*neg n*)
show *?thesis*
proof
assume *eq*: *1 + z + z = 0*
have (*0::int*) < *1* + (*of-nat n* + *of-nat n*)
by (*simp add*: *le-imp-0-less add-increasing*)
also have *... = − (1 + z + z)*
by (*simp add*: *neg add-assoc* [*symmetric*])
also have *... = 0* **by** (*simp add*: *eq*)
finally have *0<0* **..**
thus *False* **by** *blast*
qed
qed

lemma *iszero-number-of-Bit0*:
iszero (*number-of* (*Bit0 w*)::*'a*) =
iszero (*number-of w*::*'a*::{*ring-char-0*,*number-ring*})
proof −
have (*of-int w* + *of-int w* = (*0*::*'a*)) ⟹ (*w* = *0*)
proof −
assume *eq*: *of-int w* + *of-int w* = (*0*::*'a*)
then have *of-int* (*w* + *w*) = (*of-int 0* :: *'a*) **by** *simp*
then have *w* + *w* = *0* **by** (*simp only*: *of-int-eq-iff*)
then show *w* = *0* **by** (*simp only*: *double-eq-0-iff*)
qed
thus *?thesis*
by (*auto simp add*: *iszero-def number-of-eq numeral-simps*)
qed

lemma *iszero-number-of-Bit1*:
~ *iszero* (*number-of* (*Bit1 w*)::*'a*::{*ring-char-0*,*number-ring*})
proof −
have *1* + *of-int w* + *of-int w* ≠ (*0*::*'a*)
proof
assume *eq*: *1* + *of-int w* + *of-int w* = (*0*::*'a*)
hence *of-int* (*1* + *w* + *w*) = (*of-int 0* :: *'a*) **by** *simp*
hence *1* + *w* + *w* = *0* **by** (*simp only*: *of-int-eq-iff*)
with *odd-nonzero* **show** *False* **by** *blast*
qed

thus *?thesis*
by (*auto simp add*: *iszero-def number-of-eq numeral-simps*)
qed

23.15 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

lemma *neg-number-of-Bit0*:
neg (*number-of* (*Bit0 w*)::$'a$) =
neg (*number-of w* :: $'a$::{*ordered-idom*,*number-ring*})
by (*simp add*: *neg-def number-of-eq numeral-simps double-less-0-iff*)

lemma *neg-number-of-Bit1*:
neg (*number-of* (*Bit1 w*)::'*a*) =
neg (*number-of w* :: '*a*::{*ordered-idom*,*number-ring*})
proof −
have ((*1*::'*a*) + *of-int w* + *of-int w* < *0*) = (*of-int* (*1* + *w* + *w*) < (*of-int 0* :: '*a*))
by *simp*
also have ... = (*w* < *0*) **by** (*simp only*: *of-int-less-iff odd-less-0*)
finally show *?thesis*
by (*simp add*: *neg-def number-of-eq numeral-simps*)
qed

Less-Than or Equals

Reduces $a \leq b$ to $\neg\ b < a$ for ALL numerals.

lemmas *le-number-of-eq-not-less* =
linorder-not-less [*of number-of w number-of v*, *symmetric*, *standard*]

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*

23.16 Simplification of arithmetic operations on integer constants.

lemmas *arith-extra-simps* [*standard*, *simp*] =
number-of-add [*symmetric*]
number-of-minus [*symmetric*] *numeral-m1-eq-minus-1* [*symmetric*]
number-of-mult [*symmetric*]
diff-number-of-eq abs-number-of

For making a minimal simpset, one must include these default simprules. Also include *simp-thms*.

lemmas *arith-simps* =

normalize-bin-simps pred-bin-simps succ-bin-simps
add-bin-simps minus-bin-simps mult-bin-simps
abs-zero abs-one arith-extra-simps

Simplification of relational operations

lemmas *rel-simps* [*simp*] =
eq-number-of-eq iszero-0 nonzero-number-of-Min
iszero-number-of-Bit0 iszero-number-of-Bit1
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-Bit0 neg-number-of-Bit1
le-number-of-eq

23.17 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

23.18 The Set of Integers

context *ring-1*
begin

definition
Ints :: $'a$ *set*
where
Ints = *range of-int*

end

notation (*xsymbols*)
  Ints (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** $q = \mathit{of\text{-}int}\ z$
  unfolding *Ints-def*
proof −
  from ‹$q \in \mathbb{Z}$› **have** $q \in \mathit{range}\ \mathit{of\text{-}int}$ **unfolding** *Ints-def* **.**
  then obtain z **where** $q = \mathit{of\text{-}int}\ z$ **..**
  then show *thesis* **..**
qed

lemma *Ints-induct* [*case-names of-int*, *induct set*: *Ints*]:
  $q \in \mathbb{Z} \Longrightarrow (\bigwedge z.\ P\ (\mathit{of\text{-}int}\ z)) \Longrightarrow P\ q$
  by (*rule Ints-cases*) *auto*

end

lemma *Ints-diff* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a - b \in \mathbb{Z}$
apply (*auto simp add*: *Ints-def*)
apply (*rule range-eqI*)
apply (*rule of-int-diff* [*symmetric*])
done

The premise involving $\mathbb{Z}$ prevents $a = (1::'a) \,/\, (2::'a)$.

lemma *Ints-double-eq-0-iff*:
  assumes *in-Ints*: $a \in Ints$
  shows $(a + a = 0) = (a = (0::'a::ring\text{-}char\text{-}0))$
proof −
  from *in-Ints* **have** $a \in range\ of\text{-}int$ **unfolding** *Ints-def* [*symmetric*] **.**
  then obtain *z* **where** *a*: $a = of\text{-}int\ z$ **..**
  show *?thesis*
  proof
    assume $a = 0$
    thus $a + a = 0$ **by** *simp*
  next
    assume *eq*: $a + a = 0$
    hence $(of\text{-}int\ (z + z) = (of\text{-}int\ 0 :: 'a)$ **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*: $a \in Ints$
  shows $1 + a + a \neq (0::'a::ring\text{-}char\text{-}0)$
proof −
  from *in-Ints* **have** $a \in range\ of\text{-}int$ **unfolding** *Ints-def* [*symmetric*] **.**
  then obtain *z* **where** *a*: $a = of\text{-}int\ z$ **..**
  show *?thesis*
  proof
    assume *eq*: $1 + a + a = 0$
    hence $(of\text{-}int\ (1 + z + z) = (of\text{-}int\ 0 :: 'a)$ **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*:
  $(number\text{-}of\ w :: 'a::number\text{-}ring) \in Ints$
  unfolding *number-of-eq Ints-def* **by** *simp*

lemma *Ints-odd-less-0*:

assumes *in-Ints*: $a \in Ints$
shows $(1 + a + a < 0) = (a < (0::'a::ordered\text{-}idom))$
proof −
from *in-Ints* **have** $a \in range\ of\text{-}int$ **unfolding** *Ints-def* [*symmetric*] **.**
then obtain z **where** *a*: $a = of\text{-}int\ z$ **..**
hence $((1::'a) + a + a < 0) = (of\text{-}int\ (1 + z + z) < (of\text{-}int\ 0 :: 'a))$
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

23.19 *setsum* and *setprod*

By Jeremy Avigad

lemma *of-nat-setsum*: $of\text{-}nat\ (setsum\ f\ A) = (\sum x{\in}A.\ of\text{-}nat(f\ x))$
apply (*cases finite A*)
apply (*erule finite-induct*, *auto*)
done

lemma *of-int-setsum*: $of\text{-}int\ (setsum\ f\ A) = (\sum x{\in}A.\ of\text{-}int(f\ x))$
apply (*cases finite A*)
apply (*erule finite-induct*, *auto*)
done

lemma *of-nat-setprod*: $of\text{-}nat\ (setprod\ f\ A) = (\prod x{\in}A.\ of\text{-}nat(f\ x))$
apply (*cases finite A*)
apply (*erule finite-induct*, *auto simp add*: *of-nat-mult*)
done

lemma *of-int-setprod*: $of\text{-}int\ (setprod\ f\ A) = (\prod x{\in}A.\ of\text{-}int(f\ x))$
apply (*cases finite A*)
apply (*erule finite-induct*, *auto*)
done

lemma *setprod-nonzero-nat*:
$finite\ A \Longrightarrow (\forall x \in A.\ f\ x \neq (0::nat)) \Longrightarrow setprod\ f\ A \neq 0$
by (*rule setprod-nonzero*, *auto*)

lemma *setprod-zero-eq-nat*:
$finite\ A \Longrightarrow (setprod\ f\ A = (0::nat)) = (\exists x \in A.\ f\ x = 0)$
by (*rule setprod-zero-eq*, *auto*)

lemma *setprod-nonzero-int*:
$finite\ A \Longrightarrow (\forall x \in A.\ f\ x \neq (0::int)) \Longrightarrow setprod\ f\ A \neq 0$
by (*rule setprod-nonzero*, *auto*)

lemma *setprod-zero-eq-int*:
$finite\ A \Longrightarrow (setprod\ f\ A = (0::int)) = (\exists x \in A.\ f\ x = 0)$

by (*rule setprod-zero-eq*, *auto*)

lemmas *int-setsum* = *of-nat-setsum* [**where** *'a=int*]
lemmas *int-setprod* = *of-nat-setprod* [**where** *'a=int*]

23.20 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

23.21 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*]

Legacy theorems

lemmas *zle-int = of-nat-le-iff* [**where** *'a=int*]
lemmas *int-int-eq = of-nat-eq-iff* [**where** *'a=int*]

use *~~/src/Provers/Arith/assoc-fold.ML*
use *int-arith1.ML*
declaration ⟪ *K int-arith-setup* ⟫

23.22 Lemmas About Small Numerals

lemma *of-int-m1* [*simp*]: *of-int −1 = (−1 :: 'a :: number-ring)*
proof −
  have *(of-int −1 :: 'a) = of-int (− 1)* **by** *simp*
  also have *... = − of-int 1* **by** (*simp only*: *of-int-minus*)
  also have *... = −1* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *abs-minus-one* [*simp*]: *abs (−1) = (1::'a::{ordered-idom,number-ring})*
by (*simp add*: *abs-if*)

lemma *abs-power-minus-one* [*simp*]:
    abs(−1 ^ n) = (1::'a::{ordered-idom,number-ring,recpower})
by (*simp add*: *power-abs*)

lemma *of-int-number-of-eq*:
    of-int (number-of v) = (number-of v :: 'a :: number-ring)
by (*simp add*: *number-of-eq*)

Lemmas for specialist use, NOT as default simprules

lemma *mult-2*: *2 * z = (z+z::'a::number-ring)*
proof −
  have *2*z = (1 + 1)*z* **by** *simp*
  also have *... = z+z* **by** (*simp add*: *left-distrib*)
  finally show *?thesis* **.**
qed

lemma *mult-2-right*: *z * 2 = (z+z::'a::number-ring)*
by (*subst mult-commute*, *rule mult-2*)

23.23 More Inequality Reasoning

lemma *zless-add1-eq*: *(w < z + (1::int)) = (w<z | w=z)*
by *arith*

lemma *add1-zle-eq*: *(w + (1::int) ≤ z) = (w<z)*
by *arith*

lemma *zle-diff1-eq* [*simp*]: *(w ≤ z − (1::int)) = (w<z)*

by *arith*

lemma *zle-add1-eq-le* [*simp*]: $(w < z + (1::int)) = (w \leq z)$
by *arith*

lemma *int-one-le-iff-zero-less*: $((1::int) \leq z) = (0 < z)$
by *arith*

23.24 The Functions *nat* and *int*

Simplify the terms *int* $(0::'a)$, *int* (*Suc 0*) and $w + - z$

declare *Zero-int-def* [*symmetric*, *simp*]
declare *One-int-def* [*symmetric*, *simp*]

lemmas *diff-int-def-symmetric* = *diff-int-def* [*symmetric*, *simp*]

lemma *nat-0*: *nat 0* = *0*
by (*simp add*: *nat-eq-iff*)

lemma *nat-1*: *nat 1* = *Suc 0*
by (*subst nat-eq-iff*, *simp*)

lemma *nat-2*: *nat 2* = *Suc* (*Suc 0*)
by (*subst nat-eq-iff*, *simp*)

lemma *one-less-nat-eq* [*simp*]: (*Suc 0* < *nat z*) = (*1* < *z*)
apply (*insert zless-nat-conj* [*of 1 z*])
apply (*auto simp add*: *nat-1*)
done

This simplifies expressions of the form *int n* = *z* where z is an integer literal.

lemmas *int-eq-iff-number-of* [*simp*] = *int-eq-iff* [*of - number-of v*, *standard*]

lemma *split-nat* [*arith-split*]:
  P(*nat*(*i*::*int*)) = ((∀ *n*. *i* = *of-nat 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*:
fixes *z z′* :: *int*
assumes *0 ≤ z*
shows *nat (z ∗ z′) = nat z ∗ nat z′*
proof (*cases 0 ≤ z′*)
case *False* **with** *assms* **have** *z ∗ z′ ≤ 0*
by (*simp add*: *not-le mult-le-0-iff*)
then have *nat (z ∗ z′) = 0* **by** *simp*
moreover from *False* **have** *nat z′ = 0* **by** *simp*
ultimately show *?thesis* **by** *simp*
next
case *True* **with** *assms* **have** *ge-0*: *z ∗ z′ ≥ 0* **by** (*simp add*: *zero-le-mult-iff*)
show *?thesis*
by (*rule injD* [*of of-nat* :: *nat ⇒ int*, *OF inj-of-nat*])
(*simp only*: *of-nat-mult of-nat-nat* [*OF True*]
of-nat-nat [*OF assms*] *of-nat-nat* [*OF ge-0*], *simp*)
qed

lemma *nat-mult-distrib-neg*: *z ≤ (0*::*int) ==> nat(z∗z′) = nat(−z) ∗ nat(−z′)*
apply (*rule trans*)
apply (*rule-tac* [*2*] *nat-mult-distrib*, *auto*)
done

lemma *nat-abs-mult-distrib*: *nat (abs (w ∗ z)) = nat (abs w) ∗ nat (abs z)*
apply (*cases z=0 | w=0*)
apply (*auto simp add*: *abs-if nat-mult-distrib* [*symmetric*]
nat-mult-distrib-neg [*symmetric*] *mult-less-0-iff*)
done

23.25 Induction principles for int

Well-founded segments of the integers

definition
int-ge-less-than :: *int => (int ∗ int) set*

where
int-ge-less-than d = {(*z'*,*z*). *d* ≤ *z'* & *z'* < *z*}

theorem *wf-int-ge-less-than*: *wf* (*int-ge-less-than d*)
proof −
have *int-ge-less-than d* ⊆ *measure* (%*z*. *nat* (*z*−*d*))
by (*auto simp add*: *int-ge-less-than-def*)
thus *?thesis*
by (*rule wf-subset* [*OF wf-measure*])
qed

This variant looks odd, but is typical of the relations suggested by Rank-Finder.

definition
int-ge-less-than2 :: *int* => (*int* ∗ *int*) *set*
where
int-ge-less-than2 d = {(*z'*,*z*). *d* ≤ *z* & *z'* < *z*}

theorem *wf-int-ge-less-than2*: *wf* (*int-ge-less-than2 d*)
proof −
have *int-ge-less-than2 d* ⊆ *measure* (%*z*. *nat* (*1*+*z*−*d*))
by (*auto simp add*: *int-ge-less-than2-def*)
thus *?thesis*
by (*rule wf-subset* [*OF wf-measure*])
qed

abbreviation
int :: *nat* ⇒ *int*
where
int ≡ *of-nat*

theorem *int-ge-induct* [*case-names base step*, *induct set*: *int*]:
fixes *i* :: *int*
assumes *ge*: *k* ≤ *i* **and**
base: *P k* **and**
step: ⋀*i*. *k* ≤ *i* ⟹ *P i* ⟹ *P* (*i* + *1*)
shows *P i*
proof −
{ **fix** *n* **have** ⋀*i*::*int*. *n* = *nat*(*i*−*k*) ⟹ *k* ≤ *i* ⟹ *P i*
proof (*induct n*)
case *0*
hence *i* = *k* **by** *arith*
thus *P i* **using** *base* **by** *simp*
next
case (*Suc n*)
then have *n* = *nat*((*i* − *1*) − *k*) **by** *arith*
moreover
have *ki1*: *k* ≤ *i* − *1* **using** *Suc.prems* **by** *arith*

```
    ultimately
    have P(i − 1) by(rule Suc.hyps)
    from step[OF ki1 this] show ?case by simp
   qed
  }
  with ge show ?thesis by fast
qed


theorem int-gr-induct [case-names base step, induct set: int]:
  assumes gr: k < (i::int) and
        base: P(k+1) and
        step: ⋀i. ⟦k < i; P i⟧ ⟹ P(i+1)
  shows P i
apply(rule int-ge-induct[of k + 1])
  using gr apply arith
 apply(rule base)
apply (rule step, simp+)
done

theorem int-le-induct[consumes 1,case-names base step]:
  assumes le: i ≤ (k::int) and
        base: P(k) and
        step: ⋀i. ⟦i ≤ k; P i⟧ ⟹ P(i − 1)
  shows P i
proof −
  { fix n have ⋀i::int. n = nat(k−i) ⟹ i ≤ k ⟹ P i
    proof (induct n)
      case 0
      hence i = k by arith
      thus P i using base by simp
    next
      case (Suc n)
      hence n = nat(k − (i+1)) by arith
      moreover
      have ki1: i + 1 ≤ k using Suc.prems by arith
      ultimately
      have P(i+1) by(rule Suc.hyps)
      from step[OF ki1 this] show ?case by simp
    qed
  }
  with le show ?thesis by fast
qed

theorem int-less-induct [consumes 1,case-names base step]:
  assumes less: (i::int) < k and
        base: P(k − 1) and
        step: ⋀i. ⟦i < k; P i⟧ ⟹ P(i − 1)
  shows P i
```

apply(*rule int-le-induct*[*of* - $k - 1$])
using *less* **apply** *arith*
apply(*rule base*)
apply (*rule step*, *simp*+)
done

23.26 Intermediate value theorems

lemma *int-val-lemma*:
$(\forall i{<}n{::}nat.\ abs(f(i{+}1) - f\ i) \leq 1) \longrightarrow$
$f\ 0 \leq k \longrightarrow k \leq f\ n \longrightarrow (\exists i \leq n.\ f\ i = (k{::}int))$
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*:
$[\!|\ \forall i.\ m \leq i\ \&\ i < n \longrightarrow abs(f(i + 1{::}nat) - f\ i) \leq 1;\ m < n;$
$f\ m \leq k;\ k \leq f\ n\ |\!] \Longrightarrow ?\ i.\ m \leq i\ \&\ i \leq n\ \&\ f\ i = (k{::}int)$
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

23.27 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*: $(|m * n| = 1) \Longrightarrow |m| = (1{::}int)$
apply (*cases* $|n|{=}1$)
apply (*simp add*: *abs-mult*)
apply (*rule ccontr*)
apply (*auto simp add*: *linorder-neq-iff abs-mult*)
apply (*subgoal-tac* $2 \leq |m|\ \&\ 2 \leq |n|$)
prefer 2 **apply** *arith*
apply (*subgoal-tac* $2{*}2 \leq |m| * |n|$, *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*: $\sim finite(UNIV::int\ set)$
proof
assume $finite(UNIV::int\ set)$
moreover have $\sim(EX\ i::int.\ 2*i = 1)$
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

23.28 Integer Powers

instantiation *int* :: *recpower*
begin

primrec *power-int* **where**
$p \ \hat{} \ 0 = (1::int)$
$\mid p \ \hat{} \ (Suc\ n) = (p::int) * (p \ \hat{} \ n)$

instance 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

end

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* ^ *n*) ⟷ (*x* ≠ (*0*::*int*) | *n* = *0*)
by (*induct n*) (*auto simp add*: *zero-less-mult-iff*)

lemma *zero-le-zpower-abs* [*simp*]: (*0*::*int*) ≤ *abs x* ^ *n*
by (*induct n*) (*auto simp add*: *zero-le-mult-iff*)

lemma *of-int-power*:
of-int (*z* ^ *n*) = (*of-int z* ^ *n* :: *'a*::{*recpower*, *ring-1*})
by (*induct n*) (*simp-all add*: *power-Suc*)

lemma *int-power*: *int* (*m*^*n*) = (*int m*) ^ *n*
by (*rule of-nat-power*)

lemmas *zpower-int* = *int-power* [*symmetric*]

23.29 Configuration of the code generator

code-datatype *Pls Min Bit0 Bit1 number-of* :: *int* ⇒ *int*

lemmas *pred-succ-numeral-code* [*code func*] =
pred-bin-simps succ-bin-simps

lemmas *plus-numeral-code* [*code func*] =
add-bin-simps
arith-extra-simps(*1*) [**where** *'a* = *int*]

lemmas *minus-numeral-code* [*code func*] =
minus-bin-simps
arith-extra-simps(*2*) [**where** *'a* = *int*]
arith-extra-simps(*5*) [**where** *'a* = *int*]

lemmas *times-numeral-code* [*code func*] =
mult-bin-simps
arith-extra-simps(*4*) [**where** *'a* = *int*]

instantiation *int* :: *eq*
begin

definition [*code func del*]: *eq-class.eq k l* ⟷ *k* − *l* = (*0*::*int*)

instance by *default* (*simp add*: *eq-int-def*)

end

lemma *eq-number-of-int-code* [*code func*]:
eq-class.eq (*number-of k* :: *int*) (*number-of l*) ⟷ *eq-class.eq k l*

unfolding *eq-int-def number-of-is-id* **..**

lemma *eq-int-code* [*code func*]:
eq-class.eq Int.Pls Int.Pls ⟷ *True*
eq-class.eq Int.Pls Int.Min ⟷ *False*
eq-class.eq Int.Pls (*Int.Bit0 k2*) ⟷ *eq-class.eq Int.Pls k2*
eq-class.eq Int.Pls (*Int.Bit1 k2*) ⟷ *False*
eq-class.eq Int.Min Int.Pls ⟷ *False*
eq-class.eq Int.Min Int.Min ⟷ *True*
eq-class.eq Int.Min (*Int.Bit0 k2*) ⟷ *False*
eq-class.eq Int.Min (*Int.Bit1 k2*) ⟷ *eq-class.eq Int.Min k2*
eq-class.eq (*Int.Bit0 k1*) *Int.Pls* ⟷ *eq-class.eq Int.Pls k1*
eq-class.eq (*Int.Bit1 k1*) *Int.Pls* ⟷ *False*
eq-class.eq (*Int.Bit0 k1*) *Int.Min* ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) *Int.Min* ⟷ *eq-class.eq Int.Min k1*
eq-class.eq (*Int.Bit0 k1*) (*Int.Bit0 k2*) ⟷ *eq-class.eq k1 k2*
eq-class.eq (*Int.Bit0 k1*) (*Int.Bit1 k2*) ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) (*Int.Bit0 k2*) ⟷ *False*
eq-class.eq (*Int.Bit1 k1*) (*Int.Bit1 k2*) ⟷ *eq-class.eq k1 k2*
unfolding *eq-number-of-int-code* [*symmetric*, *of Int.Pls*]
eq-number-of-int-code [*symmetric*, *of Int.Min*]
eq-number-of-int-code [*symmetric*, *of Int.Bit0 k1*]
eq-number-of-int-code [*symmetric*, *of Int.Bit1 k1*]
eq-number-of-int-code [*symmetric*, *of Int.Bit0 k2*]
eq-number-of-int-code [*symmetric*, *of Int.Bit1 k2*]
by (*simp-all add*: *eq Pls-def*,
simp-all only: *Min-def succ-def pred-def number-of-is-id*)
(*auto simp add*: *iszero-def*)

lemma *less-eq-number-of-int-code* [*code func*]:
(*number-of k* :: *int*) ≤ *number-of l* ⟷ *k* ≤ *l*
unfolding *number-of-is-id* **..**

lemma *less-eq-int-code* [*code func*]:
Int.Pls ≤ *Int.Pls* ⟷ *True*
Int.Pls ≤ *Int.Min* ⟷ *False*
Int.Pls ≤ *Int.Bit0 k* ⟷ *Int.Pls* ≤ *k*
Int.Pls ≤ *Int.Bit1 k* ⟷ *Int.Pls* ≤ *k*
Int.Min ≤ *Int.Pls* ⟷ *True*
Int.Min ≤ *Int.Min* ⟷ *True*
Int.Min ≤ *Int.Bit0 k* ⟷ *Int.Min* < *k*
Int.Min ≤ *Int.Bit1 k* ⟷ *Int.Min* ≤ *k*
Int.Bit0 k ≤ *Int.Pls* ⟷ *k* ≤ *Int.Pls*
Int.Bit1 k ≤ *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit0 k ≤ *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit1 k ≤ *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit0 k1 ≤ *Int.Bit0 k2* ⟷ *k1* ≤ *k2*
Int.Bit0 k1 ≤ *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
Int.Bit1 k1 ≤ *Int.Bit0 k2* ⟷ *k1* < *k2*

Int.Bit1 k1 ≤ *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
unfolding *less-eq-number-of-int-code* [*symmetric*, *of Int.Pls*]
less-eq-number-of-int-code [*symmetric*, *of Int.Min*]
less-eq-number-of-int-code [*symmetric*, *of Int.Bit0 k1*]
less-eq-number-of-int-code [*symmetric*, *of Int.Bit1 k1*]
less-eq-number-of-int-code [*symmetric*, *of Int.Bit0 k2*]
less-eq-number-of-int-code [*symmetric*, *of Int.Bit1 k2*]
by (*simp-all add*: *Pls-def*, *simp-all only*: *Min-def succ-def pred-def number-of-is-id*)
(*auto simp add*: *neg-def linorder-not-less group-simps*
zle-add1-eq-le [*symmetric*] *del*: *iffI* , *auto simp only*: *Bit0-def Bit1-def*)

lemma *less-number-of-int-code* [*code func*]:
(*number-of k* :: *int*) < *number-of l* ⟷ *k* < *l*
unfolding *number-of-is-id* **..**

lemma *less-int-code* [*code func*]:
Int.Pls < *Int.Pls* ⟷ *False*
Int.Pls < *Int.Min* ⟷ *False*
Int.Pls < *Int.Bit0 k* ⟷ *Int.Pls* < *k*
Int.Pls < *Int.Bit1 k* ⟷ *Int.Pls* ≤ *k*
Int.Min < *Int.Pls* ⟷ *True*
Int.Min < *Int.Min* ⟷ *False*
Int.Min < *Int.Bit0 k* ⟷ *Int.Min* < *k*
Int.Min < *Int.Bit1 k* ⟷ *Int.Min* < *k*
Int.Bit0 k < *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit1 k < *Int.Pls* ⟷ *k* < *Int.Pls*
Int.Bit0 k < *Int.Min* ⟷ *k* ≤ *Int.Min*
Int.Bit1 k < *Int.Min* ⟷ *k* < *Int.Min*
Int.Bit0 k1 < *Int.Bit0 k2* ⟷ *k1* < *k2*
Int.Bit0 k1 < *Int.Bit1 k2* ⟷ *k1* ≤ *k2*
Int.Bit1 k1 < *Int.Bit0 k2* ⟷ *k1* < *k2*
Int.Bit1 k1 < *Int.Bit1 k2* ⟷ *k1* < *k2*
unfolding *less-number-of-int-code* [*symmetric*, *of Int.Pls*]
less-number-of-int-code [*symmetric*, *of Int.Min*]
less-number-of-int-code [*symmetric*, *of Int.Bit0 k1*]
less-number-of-int-code [*symmetric*, *of Int.Bit1 k1*]
less-number-of-int-code [*symmetric*, *of Int.Bit0 k2*]
less-number-of-int-code [*symmetric*, *of Int.Bit1 k2*]
by (*simp-all add*: *Pls-def*, *simp-all only*: *Min-def succ-def pred-def number-of-is-id*)
(*auto simp add*: *neg-def group-simps zle-add1-eq-le* [*symmetric*] *del*: *iffI*,
auto simp only: *Bit0-def Bit1-def*)

definition
int-aux :: *nat* ⇒ *int* ⇒ *int* **where**
[*code func del*]: *int-aux* = *of-nat-aux*

lemmas *int-aux-code* = *of-nat-aux-code* [**where** *?'a* = *int*, *simplified int-aux-def* [*symmetric*], *code*]

lemma [*code*, *code unfold*, *code inline del*]:
of-nat n = int-aux n 0
by (*simp add*: *int-aux-def of-nat-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*)

hide (**open**) *const int-aux nat-aux*

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*
Int Integer

code-modulename *OCaml*
Int Integer

code-modulename *Haskell*
Int Integer

types-code
int (*int*)
attach (*term-of*) ⟪
val term-of-int = HOLogic.mk-number HOLogic.intT;
⟫
attach (*test*) ⟪
fun gen-int i =
let val j = one-of [~*1*, *1*] ∗ *random-range 0 i*
in (*j*, *fn* () => *term-of-int j*) *end*;
⟫

setup ⟪
let

```
fun strip-number-of (@{term Int.number-of :: int => int} $ t) = t
  | strip-number-of t = t;

fun numeral-codegen thy defs 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, HO-
Logic.intT)),
      Codegen.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 Bit0 Bit1 succ pred*

23.30 Legacy theorems

lemmas *zminus-zminus* = *minus-minus* [*of z*::*int*, *standard*]
lemmas *zminus-0* = *minus-zero* [**where** *'a*=*int*]
lemmas *zminus-zadd-distrib* = *minus-add-distrib* [*of z*::*int w*, *standard*]
lemmas *zadd-commute* = *add-commute* [*of z*::*int w*, *standard*]
lemmas *zadd-assoc* = *add-assoc* [*of z1*::*int z2 z3*, *standard*]
lemmas *zadd-left-commute* = *add-left-commute* [*of x*::*int y z*, *standard*]
lemmas *zadd-ac* = *zadd-assoc zadd-commute zadd-left-commute*
lemmas *zmult-ac* = *OrderedGroup.mult-ac*
lemmas *zadd-0* = *OrderedGroup.add-0-left* [*of z*::*int*, *standard*]
lemmas *zadd-0-right* = *OrderedGroup.add-0-left* [*of z*::*int*, *standard*]
lemmas *zadd-zminus-inverse2* = *left-minus* [*of z*::*int*, *standard*]
lemmas *zmult-zminus* = *mult-minus-left* [*of z*::*int w*, *standard*]
lemmas *zmult-commute* = *mult-commute* [*of z*::*int w*, *standard*]
lemmas *zmult-assoc* = *mult-assoc* [*of z1*::*int z2 z3*, *standard*]
lemmas *zadd-zmult-distrib* = *left-distrib* [*of z1*::*int z2 w*, *standard*]

lemmas *zadd-zmult-distrib2 = right-distrib* [*of w*::*int z1 z2*, *standard*]
lemmas *zdiff-zmult-distrib = left-diff-distrib* [*of z1*::*int z2 w*, *standard*]
lemmas *zdiff-zmult-distrib2 = right-diff-distrib* [*of w*::*int z1 z2*, *standard*]

lemmas *zmult-1 = mult-1-left* [*of z*::*int*, *standard*]
lemmas *zmult-1-right = mult-1-right* [*of z*::*int*, *standard*]

lemmas *zle-refl = order-refl* [*of w*::*int*, *standard*]
lemmas *zle-trans = order-trans* [**where** *'a=int* **and** *x=i* **and** *y=j* **and** *z=k*, *standard*]
lemmas *zle-anti-sym = order-antisym* [*of z*::*int w*, *standard*]
lemmas *zle-linear = linorder-linear* [*of z*::*int w*, *standard*]
lemmas *zless-linear = linorder-less-linear* [**where** *'a = int*]

lemmas *zadd-left-mono = add-left-mono* [*of i*::*int j k*, *standard*]
lemmas *zadd-strict-right-mono = add-strict-right-mono* [*of i*::*int j k*, *standard*]
lemmas *zadd-zless-mono = add-less-le-mono* [*of w'*::*int w z' z*, *standard*]

lemmas *int-0-less-1 = zero-less-one* [**where** *'a=int*]
lemmas *int-0-neq-1 = zero-neq-one* [**where** *'a=int*]

lemmas *inj-int = inj-of-nat* [**where** *'a=int*]
lemmas *zadd-int = of-nat-add* [**where** *'a=int*, *symmetric*]
lemmas *int-mult = of-nat-mult* [**where** *'a=int*]
lemmas *zmult-int = of-nat-mult* [**where** *'a=int*, *symmetric*]
lemmas *int-eq-0-conv = of-nat-eq-0-iff* [**where** *'a=int* **and** *m=n*, *standard*]
lemmas *zless-int = of-nat-less-iff* [**where** *'a=int*]
lemmas *int-less-0-conv = of-nat-less-0-iff* [**where** *'a=int* **and** *m=k*, *standard*]
lemmas *zero-less-int-conv = of-nat-0-less-iff* [**where** *'a=int*]
lemmas *zero-zle-int = of-nat-0-le-iff* [**where** *'a=int*]
lemmas *int-le-0-conv = of-nat-le-0-iff* [**where** *'a=int* **and** *m=n*, *standard*]
lemmas *int-0 = of-nat-0* [**where** *'a=int*]
lemmas *int-1 = of-nat-1* [**where** *'a=int*]
lemmas *int-Suc = of-nat-Suc* [**where** *'a=int*]
lemmas *abs-int-eq = abs-of-nat* [**where** *'a=int* **and** *n=m*, *standard*]
lemmas *of-int-int-eq = of-int-of-nat-eq* [**where** *'a=int*]
lemmas *zdiff-int = of-nat-diff* [**where** *'a=int*, *symmetric*]
lemmas *zless-le = less-int-def*
lemmas *int-eq-of-nat = TrueI*

end

24 FunDef: General recursive function definitions

theory *FunDef*
imports *Wellfounded*
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/sum-tree.ML*)
(*Tools/function-package/mutual.ML*)
(*Tools/function-package/pattern-split.ML*)
(*Tools/function-package/fundef-package.ML*)
(*Tools/function-package/auto-term.ML*)
(*Tools/function-package/induction-scheme.ML*)
(*Tools/function-package/measure-functions.ML*)
(*Tools/function-package/lexicographic-order.ML*)
(*Tools/function-package/fundef-datatype.ML*)
begin

Definitions with default value.

definition
THE-default :: $'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
THE-default d P = (*if* $(\exists!x.\ P\ x)$ *then* (*THE x. P x*) *else d*)

lemma *THE-defaultI'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
by (*simp add*: *theI' THE-default-def*)

lemma *THE-default1-equality*:
$[\![\exists!x.\ P\ x;\ P\ a]\!] \Longrightarrow THE\text{-}default\ d\ P = a$
by (*simp add*: *the1-equality THE-default-def*)

lemma *THE-default-none*:
$\neg(\exists!x.\ P\ x) \Longrightarrow THE\text{-}default\ d\ P = d$
by (*simp add*:*THE-default-def*)

lemma *fundef-ex1-existence*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $G\ x\ (f\ x)$
apply (*simp only*: *f-def*)
apply (*rule THE-defaultI'*)
apply (*rule ex1*)
done

lemma *fundef-ex1-uniqueness*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
assumes *elm*: $G\ x\ (h\ x)$
shows $h\ x = f\ x$
apply (*simp only*: *f-def*)
apply (*rule THE-default1-equality* [*symmetric*])
apply (*rule ex1*)

apply (*rule elm*)
done

lemma *fundef-ex1-iff*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $(G\ x\ y) = (f\ x = y)$
apply (*auto simp*:*ex1 f-def THE-default1-equality*)
apply (*rule THE-defaultI'*)
apply (*rule ex1*)
done

lemma *fundef-default-value*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *graph*: $\bigwedge x\ y.\ G\ x\ y \Longrightarrow D\ x$
assumes $\neg\ D\ x$
shows $f\ x = d\ x$
proof −
have $\neg(\exists\, y.\ G\ x\ y)$
proof
assume $\exists\, y.\ G\ x\ y$
hence $D\ x$ **using** *graph* **..**
with ‹$\neg\ D\ x$› **show** *False* **..**
qed
hence $\neg(\exists!y.\ G\ x\ y)$ **by** *blast*

thus *?thesis*
unfolding *f-def*
by (*rule THE-default-none*)
qed

definition *in-rel-def*[*simp*]:
$in\text{-}rel\ R\ x\ y == (x,\ y) \in R$

lemma *wf-in-rel*:
$wf\ R \Longrightarrow wfP\ (in\text{-}rel\ R)$
by (*simp add*: *wfP-def*)

inductive *is-measure* :: $('a \Rightarrow nat) \Rightarrow bool$
where *is-measure-trivial*: *is-measure f*

use *Tools/function-package/fundef-lib.ML*
use *Tools/function-package/fundef-common.ML*
use *Tools/function-package/inductive-wrap.ML*
use *Tools/function-package/context-tree.ML*
use *Tools/function-package/fundef-core.ML*
use *Tools/function-package/sum-tree.ML*
use *Tools/function-package/mutual.ML*
use *Tools/function-package/pattern-split.ML*

use *Tools/function-package/auto-term.ML*
use *Tools/function-package/fundef-package.ML*
use *Tools/function-package/induction-scheme.ML*
use *Tools/function-package/measure-functions.ML*
use *Tools/function-package/lexicographic-order.ML*
use *Tools/function-package/fundef-datatype.ML*

setup ⟪
FundefPackage.setup
#> *InductionScheme.setup*
#> *MeasureFunctions.setup*
#> *LexicographicOrder.setup*
#> *FundefDatatype.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* **.**

24.1 Setup for termination proofs

Rules for generating measure functions

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

lemma [*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* $(\lambda p.\ f\ (fst\ p))$
by (*rule is-measure-trivial*)
lemma [*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* $(\lambda p.\ f\ (snd\ p))$
by (*rule is-measure-trivial*)

lemma *termination-basic-simps*[*termination-simp*]:
$x < (y::nat) \Longrightarrow x < y + z$
$x < z \Longrightarrow x < y + z$
$x \leq y \Longrightarrow x \leq y + (z::nat)$
$x \leq z \Longrightarrow x \leq y + (z::nat)$
$x < y \Longrightarrow x \leq (y::nat)$

by *arith+*

declare *le-imp-less-Suc*[*termination-simp*]

lemma *prod-size-simp*[*termination-simp*]:
prod-size f g p = f (fst p) + g (snd p) + Suc 0
by (*induct p*) *auto*

end

25 IntDiv: The Division Operators div and mod; the Divides Relation dvd

theory *IntDiv*
imports *Int 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*≤*r & r*<*b else b*<*r & r* ≤ *0*)

adjust :: [*int*, *int*∗*int*] => *int*∗*int*
— for the division algorithm
[*code func*]: *adjust b* == %(*q*,*r*). *if 0* ≤ *r*−*b then* (*2*∗*q + 1*, *r*−*b*)
else (*2*∗*q*, *r*)

algorithm for the case $a \geq 0,\ b > 0$

function
posDivAlg :: *int* ⇒ *int* ⇒ *int* × *int*
where
posDivAlg a b =
(*if* (*a*<*b* | *b*≤*0*) *then* (*0*,*a*)
else adjust b (*posDivAlg a* (*2*∗*b*)))
by *auto*
termination by (*relation measure* (%(*a*,*b*). *nat*(*a* − *b* + *1*))) *auto*

algorithm for the case $a < 0,\ b > 0$

function
negDivAlg :: *int* ⇒ *int* ⇒ *int* × *int*
where
negDivAlg a b =
(*if* (*0*≤*a*+*b* | *b*≤*0*) *then* (−*1*,*a*+*b*)
else adjust b (*negDivAlg a* (*2*∗*b*)))

by *auto*
termination by (*relation measure* (%(*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≤*a then*
if 0≤*b then posDivAlg a b*
else if a=*0 then* (*0*, *0*)
else negateSnd (*negDivAlg* (−*a*) (−*b*))
else
if 0<*b then negDivAlg a b*
else negateSnd (*posDivAlg* (−*a*) (−*b*))))

instantiation *int* :: *Divides.div*
begin

definition
div-def: *a div b* = *fst* (*divAlg* (*a*, *b*))

definition
mod-def: *a mod b* = *snd* (*divAlg* (*a*, *b*))

instance ..

end

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));
```

25.1 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'*)); $b \neq 0$ |]
==> $q = q'$
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 ≠ 0 |]
     ==> r = r'
apply (subgoal-tac q = q')
 apply (simp add: quorem-def)
apply (blast intro: unique-quotient)
done
```

25.2 Correctness of *posDivAlg*, the Algorithm for Non-Negative Dividends

And positive divisors

```
lemma adjust-eq [simp]:
    adjust b (q,r) =
     (let diff = r-b in
       if 0 ≤ diff then (2*q + 1, diff)
                   else (2*q, r))
by (simp add: Let-def adjust-def)

declare posDivAlg.simps [simp del]
```

use with a simproc to avoid repeatedly proving the premise

```
lemma posDivAlg-eqn:
    0 < b ==>
     posDivAlg a b = (if a<b then (0,a) else adjust b (posDivAlg a (2*b)))
by (rule posDivAlg.simps [THEN trans], simp)
```

Correctness of *posDivAlg*: it computes quotients correctly

```
theorem posDivAlg-correct:
  assumes 0 ≤ 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
```

25.3 Correctness of *negDivAlg*, the Algorithm for Negative Dividends

And positive divisors

```
declare negDivAlg.simps [simp del]
```

use with a simproc to avoid repeatedly proving the premise

lemma *negDivAlg-eqn*:
 0 < b ==>
 negDivAlg a b =
 (if 0≤a+b then (−1,a+b) else adjust b (negDivAlg a (2∗b)))
by (*rule negDivAlg.simps* [*THEN trans*], *simp*)

lemma *negDivAlg-correct*:
 assumes *a < 0* **and** *b > 0*
 shows *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

25.4 Existence Shown by Proving the Division Algorithm to be Correct

lemma *quorem-0*: *b ≠ 0 ==> quorem* ((*0*,*b*), (*0*,*0*))
by (*auto simp add*: *quorem-def linorder-neq-iff*)

lemma *posDivAlg-0* [*simp*]: *posDivAlg 0 b* = (*0*, *0*)
by (*subst posDivAlg.simps*, *auto*)

lemma *negDivAlg-minus1* [*simp*]: *negDivAlg −1 b* = (*−1*, *b − 1*)
by (*subst negDivAlg.simps*, *auto*)

lemma *negateSnd-eq* [*simp*]: *negateSnd*(*q*,*r*) = (*q*,*−r*)
by (*simp add*: *negateSnd-def*)

lemma *quorem-neg*: *quorem* ((*−a*,*−b*), *qr*) *==> quorem* ((*a*,*b*), *negateSnd qr*)
by (*auto simp add*: *split-ifs quorem-def*)

lemma *divAlg-correct*: *b ≠ 0 ==> quorem* ((*a*,*b*), *divAlg* (*a*, *b*))
by (*force simp add*: *linorder-neq-iff quorem-0 divAlg-def quorem-neg*
 posDivAlg-correct negDivAlg-correct)

Arbitrary definitions for division by zero. Useful to simplify certain equations.

lemma *DIVISION-BY-ZERO* [*simp*]: *a div* (*0*::*int*) = *0* & *a mod* (*0*::*int*) = *a*
by (*simp add*: *div-def mod-def divAlg-def posDivAlg.simps*)

Basic laws about division and remainder

lemma *zmod-zdiv-equality*: (*a*::*int*) = *b* ∗ (*a div b*) + (*a mod b*)

apply (*case-tac b = 0*, *simp*)
apply (*cut-tac a = a* **and** *b = b* **in** *divAlg-correct*)
apply (*auto simp add*: *quorem-def div-def mod-def*)
done

lemma *zdiv-zmod-equality*: $(b * (a\ div\ b) + (a\ mod\ b)) + k = (a::int)+k$
by(*simp add*: *zmod-zdiv-equality*[*symmetric*])

lemma *zdiv-zmod-equality2*: $((a\ div\ b) * b + (a\ mod\ b)) + k = (a::int)+k$
by(*simp add*: *mult-commute zmod-zdiv-equality*[*symmetric*])

Tool setup

ML ⟪
local

structure CancelDivMod = CancelDivModFun(
struct
val div-name = @{const-name Divides.div};
val mod-name = @{const-name Divides.mod};
val mk-binop = HOLogic.mk-binop;
val mk-sum = Int-Numeral-Simprocs.mk-sum HOLogic.intT;
val dest-sum = Int-Numeral-Simprocs.dest-sum;
val div-mod-eqs =
map mk-meta-eq [@{thm zdiv-zmod-equality},
@{thm zdiv-zmod-equality2}];
val trans = trans;
val prove-eq-sums =
let
val simps = @{thm diff-int-def} :: Int-Numeral-Simprocs.add-0s @ @{thms zadd-ac}
in ArithData.prove-conv all-tac (ArithData.simp-all-tac simps) end;
end)

in

val cancel-zdiv-zmod-proc = Simplifier.simproc @{theory}
cancel-zdiv-zmod [(*m*::*int*) + *n*] (*K CancelDivMod.proc*)

end;

Addsimprocs [*cancel-zdiv-zmod-proc*]
⟫

lemma *pos-mod-conj* : $(0::int) < b \Longrightarrow 0 \leq a\ mod\ b\ \&\ a\ mod\ b < b$
apply (*cut-tac a = a* **and** *b = b* **in** *divAlg-correct*)
apply (*auto simp add*: *quorem-def mod-def*)
done

lemmas *pos-mod-sign* [*simp*] = *pos-mod-conj* [*THEN conjunct1*, *standard*]

and *pos-mod-bound* [*simp*] = *pos-mod-conj* [*THEN conjunct2*, *standard*]

lemma *neg-mod-conj* : $b < (0::int) ==> a \bmod b \leq 0 \ \& \ b < a \bmod b$
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*]

25.5 General Properties of div and mod

lemma *quorem-div-mod*: $b \neq 0 ==> quorem\ ((a, b), (a \ div\ b, a \bmod b))$
apply (*cut-tac a* = *a* **and** *b* = *b* **in** *zmod-zdiv-equality*)
apply (*force simp add*: *quorem-def linorder-neq-iff*)
done

lemma *quorem-div*: $[|\ quorem((a,b),(q,r));\ b \neq 0\ |] ==> a \ div\ b = q$
by (*simp add*: *quorem-div-mod* [*THEN unique-quotient*])

lemma *quorem-mod*: $[|\ quorem((a,b),(q,r));\ b \neq 0\ |] ==> a \bmod b = r$
by (*simp add*: *quorem-div-mod* [*THEN unique-remainder*])

lemma *div-pos-pos-trivial*: $[|\ (0::int) \leq a;\ a < b\ |] ==> a \ div\ b = 0$
apply (*rule quorem-div*)
apply (*auto simp add*: *quorem-def*)
done

lemma *div-neg-neg-trivial*: $[|\ a \leq (0::int);\ b < a\ |] ==> a \ div\ b = 0$
apply (*rule quorem-div*)
apply (*auto simp add*: *quorem-def*)
done

lemma *div-pos-neg-trivial*: $[|\ (0::int) < a;\ a+b \leq 0\ |] ==> a \ div\ b = -1$
apply (*rule quorem-div*)
apply (*auto simp add*: *quorem-def*)
done

lemma *mod-pos-pos-trivial*: $[|\ (0::int) \leq a;\ a < b\ |] ==> a \bmod b = a$
apply (*rule-tac q* = *0* **in** *quorem-mod*)
apply (*auto simp add*: *quorem-def*)
done

lemma *mod-neg-neg-trivial*: $[|\ a \leq (0::int);\ b < a\ |] ==> a \bmod b = a$
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

25.6 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 ≠ (*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*:

$b \neq (0::int)$
$\Longrightarrow a\ div\ (-b) =$
$(if\ a\ mod\ b = 0\ then\ -\ (a\ div\ b)\ else\ -\ (a\ div\ b) - 1)$
by (*simp add*: *zdiv-zminus1-eq-if zdiv-zminus2*)

lemma *zmod-zminus2-eq-if*:
$a\ mod\ (-b::int) = (if\ a\ mod\ b = 0\ then\ 0\ else\ (a\ mod\ b) - b)$
by (*simp add*: *zmod-zminus1-eq-if zmod-zminus2*)

25.7 Division of a Number by Itself

lemma *self-quotient-aux1*: $[|\ (0::int) < a;\ a = r + a*q;\ r < a\ |] \Longrightarrow 1 \leq q$
apply (*subgoal-tac* $0 < a*q$)
apply (*simp add*: *zero-less-mult-iff*, *arith*)
done

lemma *self-quotient-aux2*: $[|\ (0::int) < a;\ a = r + a*q;\ 0 \leq r\ |] \Longrightarrow q \leq 1$
apply (*subgoal-tac* $0 \leq a*\ (1-q)$)
apply (*simp add*: *zero-le-mult-iff*)
apply (*simp add*: *right-diff-distrib*)
done

lemma *self-quotient*: $[|\ quorem((a,a),(q,r));\ \ a \neq (0::int)\ |] \Longrightarrow q = 1$
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*: $[|\ quorem((a,a),(q,r));\ \ a \neq (0::int)\ |] \Longrightarrow r = 0$
apply (*frule self-quotient*, *assumption*)
apply (*simp add*: *quorem-def*)
done

lemma *zdiv-self* [*simp*]: $a \neq 0 \Longrightarrow a\ div\ a = (1::int)$
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

25.8 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*: $(0::int) < b \Longrightarrow -1\ div\ b = -1$

by (*simp add*: *div-def divAlg-def*)

lemma *zmod-zero* [*simp*]: (*0*::*int*) *mod b* = *0*
by (*simp add*: *mod-def divAlg-def*)

lemma *zdiv-minus1*: (*0*::*int*) < *b* ==> *−1 div b* = *−1*
by (*simp add*: *div-def divAlg-def*)

lemma *zmod-minus1*: (*0*::*int*) < *b* ==> *−1 mod b* = *b* − *1*
by (*simp add*: *mod-def divAlg-def*)

a positive, b positive

lemma *div-pos-pos*: [| *0* < *a*; *0* ≤ *b* |] ==> *a div b* = *fst* (*posDivAlg a b*)
by (*simp add*: *div-def divAlg-def*)

lemma *mod-pos-pos*: [| *0* < *a*; *0* ≤ *b* |] ==> *a mod b* = *snd* (*posDivAlg a b*)
by (*simp add*: *mod-def divAlg-def*)

a negative, b positive

lemma *div-neg-pos*: [| *a* < *0*; *0* < *b* |] ==> *a div b* = *fst* (*negDivAlg a b*)
by (*simp add*: *div-def divAlg-def*)

lemma *mod-neg-pos*: [| *a* < *0*; *0* < *b* |] ==> *a mod b* = *snd* (*negDivAlg a b*)
by (*simp add*: *mod-def divAlg-def*)

a positive, b negative

lemma *div-pos-neg*:
[| *0* < *a*; *b* < *0* |] ==> *a div b* = *fst* (*negateSnd* (*negDivAlg* (*−a*) (*−b*)))
by (*simp add*: *div-def divAlg-def*)

lemma *mod-pos-neg*:
[| *0* < *a*; *b* < *0* |] ==> *a mod b* = *snd* (*negateSnd* (*negDivAlg* (*−a*) (*−b*)))
by (*simp add*: *mod-def divAlg-def*)

a negative, b negative

lemma *div-neg-neg*:
[| *a* < *0*; *b* ≤ *0* |] ==> *a div b* = *fst* (*negateSnd* (*posDivAlg* (*−a*) (*−b*)))
by (*simp add*: *div-def divAlg-def*)

lemma *mod-neg-neg*:
[| *a* < *0*; *b* ≤ *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*; *if 0* < *b then 0* ≤ *r* ∧ *r* < *b else b* < *r* ∧ *r* ≤ *0*⟧
⟹ *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*]

25.9 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*: [| *a* ≤ *a'*; 0 < (*b*::*int*) |] ==> *a div b* ≤ *a' div b*
apply (*cut-tac a* = *a* **and** *b* = *b* **in** *zmod-zdiv-equality*)
apply (*cut-tac a* = *a'* **and** *b* = *b* **in** *zmod-zdiv-equality*)
apply (*rule unique-quotient-lemma*)
apply (*erule subst*)
apply (*erule subst*, *simp-all*)
done

lemma *zdiv-mono1-neg*: [| *a* ≤ *a'*; (*b*::*int*) < 0 |] ==> *a' div b* ≤ *a div b*
apply (*cut-tac a* = *a* **and** *b* = *b* **in** *zmod-zdiv-equality*)
apply (*cut-tac a* = *a'* **and** *b* = *b* **in** *zmod-zdiv-equality*)
apply (*rule unique-quotient-lemma-neg*)
apply (*erule subst*)
apply (*erule subst*, *simp-all*)
done

25.10 Monotonicity in the Second Argument (Divisor)

lemma *q-pos-lemma*:
[| 0 ≤ *b'***q'* + *r'*; *r'* < *b'*; 0 < *b'* |] ==> 0 ≤ (*q'*::*int*)
apply (*subgoal-tac* 0 < *b'** (*q'* + *1*))

apply (*simp add*: *zero-less-mult-iff*)
apply (*simp add*: *right-distrib*)
done

lemma *zdiv-mono2-lemma*:
[| $b*q + r = b'*q' + r'$; $0 \leq b'*q' + r'$;
$r' < b'$; $0 \leq r$; $0 < b'$; $b' \leq b$ |]
==> $q \leq (q'::int)$
apply (*frule q-pos-lemma*, *assumption*+)
apply (*subgoal-tac* $b*q < b*(q' + 1)$)
apply (*simp add*: *mult-less-cancel-left*)
apply (*subgoal-tac* $b*q = r' - r + b'*q'$)
prefer *2* **apply** *simp*
apply (*simp* (*no-asm-simp*) *add*: *right-distrib*)
apply (*subst add-commute*, *rule zadd-zless-mono*, *arith*)
apply (*rule mult-right-mono*, *auto*)
done

lemma *zdiv-mono2*:
[| $(0::int) \leq a$; $0 < b'$; $b' \leq b$ |] ==> $a \ div\ b \leq a \ div\ b'$
apply (*subgoal-tac* $b \neq 0$)
prefer *2* **apply** *arith*
apply (*cut-tac* $a = a$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*cut-tac* $a = a$ **and** $b = b'$ **in** *zmod-zdiv-equality*)
apply (*rule zdiv-mono2-lemma*)
apply (*erule subst*)
apply (*erule subst*, *simp-all*)
done

lemma *q-neg-lemma*:
[| $b'*q' + r' < 0$; $0 \leq r'$; $0 < b'$ |] ==> $q' \leq (0::int)$
apply (*subgoal-tac* $b'*q' < 0$)
apply (*simp add*: *mult-less-0-iff*, *arith*)
done

lemma *zdiv-mono2-neg-lemma*:
[| $b*q + r = b'*q' + r'$; $b'*q' + r' < 0$;
$r < b$; $0 \leq r'$; $0 < b'$; $b' \leq b$ |]
==> $q' \leq (q::int)$
apply (*frule q-neg-lemma*, *assumption*+)
apply (*subgoal-tac* $b*q' < b*(q + 1)$)
apply (*simp add*: *mult-less-cancel-left*)
apply (*simp add*: *right-distrib*)
apply (*subgoal-tac* $b*q' \leq b'*q'$)
prefer *2* **apply** (*simp add*: *mult-right-mono-neg*, *arith*)
done

lemma *zdiv-mono2-neg*:
[| $a < (0::int)$; $0 < b'$; $b' \leq b$ |] ==> $a \ div\ b' \leq a \ div\ b$

apply (*cut-tac* $a = a$ **and** $b = b$ **in** *zmod-zdiv-equality*)
apply (*cut-tac* $a = a$ **and** $b = b'$ **in** *zmod-zdiv-equality*)
apply (*rule zdiv-mono2-neg-lemma*)
apply (*erule subst*)
apply (*erule subst*, *simp-all*)
done

25.11 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*)

instance *int* :: *semiring-div*
by *intro-classes auto*

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*]: $a \neq (0::int) \Longrightarrow (a+b)\ div\ a = b\ div\ a + 1$
by (*simp add*: *zdiv-zadd1-eq*)

lemma *zdiv-zadd-self2*[*simp*]: $a \neq (0::int) \Longrightarrow (b+a)\ div\ a = b\ div\ a + 1$
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

25.12 Proving $a\ div\ (b * c) = a\ div\ b\ div\ c$

first, four lemmas to bound the remainder for the cases b¡0 and b¿0

lemma *zmult2-lemma-aux1*: [| $(0::int) < c$; $b < r$; $r \leq 0$ |] ==> $b*c < b*(q\ mod\ c) + r$
apply (*subgoal-tac* $b * (c - q\ mod\ c) < r * 1$)
apply (*simp add*: *right-diff-distrib*)
apply (*rule order-le-less-trans*)
apply (*erule-tac* [*2*] *mult-strict-right-mono*)
apply (*rule mult-left-mono-neg*)
apply (*auto simp add*: *compare-rls add-commute* [*of 1*]
add1-zle-eq pos-mod-bound)
done

lemma *zmult2-lemma-aux2*:
[| $(0::int) < c$; $b < r$; $r \leq 0$ |] ==> $b * (q\ mod\ c) + r \leq 0$
apply (*subgoal-tac* $b * (q\ mod\ c) \leq 0$)
apply *arith*

apply (*simp add*: *mult-le-0-iff*)
done

lemma *zmult2-lemma-aux3*: [| (*0*::*int*) < *c*; 0 ≤ *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

25.13 Cancellation of Common Factors in div

lemma *zdiv-zmult-zmult1-aux1*:
[| (*0*::*int*) < *b*; *c* ≠ *0* |] ==> (*c*∗*a*) *div* (*c*∗*b*) = *a* *div* *b*
by (*subst zdiv-zmult2-eq*, *auto*)

lemma *zdiv-zmult-zmult1-aux2*:
[| *b* < (*0*::*int*); *c* ≠ *0* |] ==> (*c*∗*a*) *div* (*c*∗*b*) = *a* *div* *b*
apply (*subgoal-tac* (*c* * (−*a*)) *div* (*c* * (−*b*)) = (−*a*) *div* (−*b*))
apply (*rule-tac* [*2*] *zdiv-zmult-zmult1-aux1*, *auto*)

done

lemma *zdiv-zmult-zmult1*: $c \neq (0::int) \Longrightarrow (c*a)\ div\ (c*b) = a\ div\ b$
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*)

25.14 Distribution of Factors over mod

lemma *zmod-zmult-zmult1-aux1*:
$[|\ (0::int) < b;\ \ c \neq 0\ |] \Longrightarrow (c*a)\ mod\ (c*b) = c * (a\ mod\ b)$
by (*subst zmod-zmult2-eq*, *auto*)

lemma *zmod-zmult-zmult1-aux2*:
$[|\ b < (0::int);\ \ c \neq 0\ |] \Longrightarrow (c*a)\ mod\ (c*b) = c * (a\ mod\ b)$
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 $\ldots = (m * (k\ div\ m)\ mod\ n + k\ mod\ m\ mod\ n)\ mod\ n$
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

25.15 Splitting Rules for div and mod

The proofs of the two lemmas below are essentially identical

lemma *split-pos-lemma*:
0<k ==>
P(n div k :: int)(n mod k) = (∀ i j. 0≤j & j<k & n = k∗i + j --> P i j)
apply (*rule iffI*, *clarify*)
apply (*erule-tac P=P ?x ?y* **in** *rev-mp*)
apply (*subst zmod-zadd1-eq*)
apply (*subst zdiv-zadd1-eq*)
apply (*simp add*: *div-pos-pos-trivial mod-pos-pos-trivial*)

converse direction

apply (*drule-tac x = n div k* **in** *spec*)
apply (*drule-tac x = n mod k* **in** *spec*, *simp*)
done

lemma *split-neg-lemma*:
k<0 ==>
P(n div k :: int)(n mod k) = (∀ i j. k<j & j≤0 & n = k∗i + j --> P i j)
apply (*rule iffI*, *clarify*)
apply (*erule-tac P=P ?x ?y* **in** *rev-mp*)
apply (*subst zmod-zadd1-eq*)
apply (*subst zdiv-zadd1-eq*)
apply (*simp add*: *div-neg-neg-trivial mod-neg-neg-trivial*)

converse direction

apply (*drule-tac x = n div k* **in** *spec*)
apply (*drule-tac x = n mod k* **in** *spec*, *simp*)
done

lemma *split-zdiv*:
P(n div k :: int) =
((k = 0 --> P 0) &
(0<k --> (∀ i j. 0≤j & j<k & n = k∗i + j --> P i)) &
(k<0 --> (∀ i j. k<j & j≤0 & n = k∗i + j --> P i)))
apply (*case-tac k=0*, *simp*)
apply (*simp only*: *linorder-neq-iff*)
apply (*erule disjE*)
apply (*simp-all add*: *split-pos-lemma* [*of concl*: *%x y. P x*]
split-neg-lemma [*of concl*: *%x y. P x*])
done

lemma *split-zmod*:
P(n mod k :: int) =
((k = 0 --> P n) &
(0<k --> (∀ i j. 0≤j & j<k & n = k∗i + j --> P j)) &
(k<0 --> (∀ i j. k<j & j≤0 & n = k∗i + j --> P j)))
apply (*case-tac k=0*, *simp*)

apply (*simp only*: *linorder-neq-iff*)
apply (*erule disjE*)
apply (*simp-all add*: *split-pos-lemma* [*of concl*: *%x y. P y*]
split-neg-lemma [*of concl*: *%x y. P y*])
done

declare *split-zdiv* [*of - - number-of k*, *simplified*, *standard*, *arith-split*]
declare *split-zmod* [*of - - number-of k*, *simplified*, *standard*, *arith-split*]

25.16 Speeding up the Division Algorithm with Shifting

computing div by shifting

lemma *pos-zdiv-mult-2*: *(0::int) ≤ a ==> (1 + 2*b) div (2*a) = b div a*
proof *cases*
assume *a=0*
thus *?thesis* **by** *simp*
next
assume *a≠0* **and** *le-a*: *0≤a*
hence *a-pos*: *1 ≤ a* **by** *arith*
hence *one-less-a2*: *1 < 2*a* **by** *arith*
hence *le-2a*: *2 * (1 + b mod a) ≤ 2 * a*
by (*simp add*: *mult-le-cancel-left add-commute* [*of 1*] *add1-zle-eq*)
with *a-pos* **have** *0 ≤ b mod a* **by** *simp*
hence *le-addm*: *0 ≤ 1 mod (2*a) + 2*(b mod a)*
by (*simp add*: *mod-pos-pos-trivial one-less-a2*)
with *le-2a*
have *(1 mod (2*a) + 2*(b mod a)) div (2*a) = 0*
by (*simp add*: *div-pos-pos-trivial le-addm mod-pos-pos-trivial one-less-a2*
right-distrib)
thus *?thesis*
by (*subst zdiv-zadd1-eq*,
simp add: *zdiv-zmult-zmult1 zmod-zmult-zmult1 one-less-a2*
div-pos-pos-trivial)
qed

lemma *neg-zdiv-mult-2*: *a ≤ (0::int) ==> (1 + 2*b) div (2*a) = (b+1) div a*
apply (*subgoal-tac* *(1 + 2* (−b − 1)) div (2 * (−a)) = (−b − 1) div (−a)*)
apply (*rule-tac* [*2*] *pos-zdiv-mult-2*)
apply (*auto simp add*: *minus-mult-right* [*symmetric*] *right-diff-distrib*)
apply (*subgoal-tac* *(−1 − (2 * b)) = − (1 + (2 * b)))*
apply (*simp only*: *zdiv-zminus-zminus diff-minus minus-add-distrib* [*symmetric*],
simp)
done

lemma *not-0-le-lemma*: *~ 0 ≤ x ==> x ≤ (0::int)*
by *auto*

lemma *zdiv-number-of-Bit0* [*simp*]:
number-of (*Int.Bit0 v*) *div number-of* (*Int.Bit0 w*) =
number-of v div (*number-of w* :: *int*)
by (*simp only*: *number-of-eq numeral-simps*) *simp*

lemma *zdiv-number-of-Bit1* [*simp*]:
number-of (*Int.Bit1 v*) *div number-of* (*Int.Bit0 w*) =
(*if* (*0*::*int*) ≤ *number-of w*
then number-of v div (*number-of w*)
else (*number-of v* + (*1*::*int*)) *div* (*number-of w*))
apply (*simp only*: *number-of-eq numeral-simps UNIV-I split*: *split-if*)
apply (*simp add*: *zdiv-zmult-zmult1 pos-zdiv-mult-2 neg-zdiv-mult-2 add-ac*)
done

25.17 Computing mod by Shifting (proofs resemble those for div)

lemma *pos-zmod-mult-2*:
(*0*::*int*) ≤ *a* ==> (*1* + *2*∗*b*) *mod* (*2*∗*a*) = *1* + *2* ∗ (*b mod a*)
apply (*case-tac a* = *0*, *simp*)
apply (*subgoal-tac 1* < *a* ∗ *2*)
prefer *2* **apply** *arith*
apply (*subgoal-tac 2*∗ (*1* + *b mod a*) ≤ *2*∗*a*)
apply (*rule-tac* [*2*] *mult-left-mono*)
apply (*auto simp add*: *add-commute* [*of 1*] *mult-commute add1-zle-eq*
pos-mod-bound)
apply (*subst zmod-zadd1-eq*)
apply (*simp add*: *zmod-zmult-zmult2 mod-pos-pos-trivial*)
apply (*rule mod-pos-pos-trivial*)
apply (*auto simp add*: *mod-pos-pos-trivial ring-distribs*)
apply (*subgoal-tac 0* ≤ *b mod a*, *arith*, *simp*)
done

lemma *neg-zmod-mult-2*:
a ≤ (*0*::*int*) ==> (*1* + *2*∗*b*) *mod* (*2*∗*a*) = *2* ∗ ((*b*+*1*) *mod a*) − *1*
apply (*subgoal-tac* (*1* + *2*∗ (−*b* − *1*)) *mod* (*2*∗ (−*a*)) =
1 + *2*∗ ((−*b* − *1*) *mod* (−*a*)))
apply (*rule-tac* [*2*] *pos-zmod-mult-2*)
apply (*auto simp add*: *minus-mult-right* [*symmetric*] *right-diff-distrib*)
apply (*subgoal-tac* (−*1* − (*2* ∗ *b*)) = − (*1* + (*2* ∗ *b*)))
prefer *2* **apply** *simp*
apply (*simp only*: *zmod-zminus-zminus diff-minus minus-add-distrib* [*symmetric*])
done

lemma *zmod-number-of-Bit0* [*simp*]:
number-of (*Int.Bit0 v*) *mod number-of* (*Int.Bit0 w*) =
(*2*::*int*) ∗ (*number-of v mod number-of w*)
apply (*simp only*: *number-of-eq numeral-simps*)
apply (*simp add*: *zmod-zmult-zmult1 pos-zmod-mult-2*

not-0-le-lemma neg-zmod-mult-2 add-ac)
done

lemma *zmod-number-of-Bit1* [*simp*]:
number-of (*Int.Bit1 v*) *mod number-of* (*Int.Bit0 w*) =
(*if* (*0*::*int*) ≤ *number-of w*
then 2 ∗ (*number-of v mod number-of w*) + *1*
else 2 ∗ ((*number-of v* + (*1*::*int*)) *mod number-of w*) − *1*)
apply (*simp only*: *number-of-eq numeral-simps*)
apply (*simp add*: *zmod-zmult-zmult1 pos-zmod-mult-2*
not-0-le-lemma neg-zmod-mult-2 add-ac)
done

25.18 Quotients of Signs

lemma *div-neg-pos-less0*: [| *a* < (*0*::*int*); *0* < *b* |] ==> *a div b* < *0*
apply (*subgoal-tac a div b* ≤ −*1*, *force*)
apply (*rule order-trans*)
apply (*rule-tac a'* = −*1* **in** *zdiv-mono1*)
apply (*auto simp add*: *zdiv-minus1*)
done

lemma *div-nonneg-neg-le0*: [| (*0*::*int*) ≤ *a*; *b* < *0* |] ==> *a div b* ≤ *0*
by (*drule zdiv-mono1-neg*, *auto*)

lemma *pos-imp-zdiv-nonneg-iff*: (*0*::*int*) < *b* ==> (*0* ≤ *a div b*) = (*0* ≤ *a*)
apply *auto*
apply (*drule-tac* [*2*] *zdiv-mono1*)
apply (*auto simp add*: *linorder-neg-iff*)
apply (*simp* (*no-asm-use*) *add*: *linorder-not-less* [*symmetric*])
apply (*blast intro*: *div-neg-pos-less0*)
done

lemma *neg-imp-zdiv-nonneg-iff*:
b < (*0*::*int*) ==> (*0* ≤ *a div b*) = (*a* ≤ (*0*::*int*))
apply (*subst zdiv-zminus-zminus* [*symmetric*])
apply (*subst pos-imp-zdiv-nonneg-iff*, *auto*)
done

lemma *pos-imp-zdiv-neg-iff*: (*0*::*int*) < *b* ==> (*a div b* < *0*) = (*a* < *0*)
by (*simp add*: *linorder-not-le* [*symmetric*] *pos-imp-zdiv-nonneg-iff*)

lemma *neg-imp-zdiv-neg-iff*: *b* < (*0*::*int*) ==> (*a div b* < *0*) = (*0* < *a*)
by (*simp add*: *linorder-not-le* [*symmetric*] *neg-imp-zdiv-nonneg-iff*)

25.19 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*)

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* $m \Longrightarrow n * (m \ div \ n) = m$
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:$k \neq (0{::}int)$
shows m *dvd* n
proof−
from d **obtain** h **where** h: $k{*}n = k{*}m * h$ **unfolding** *dvd-def* **by** *blast*
{assume $n \neq m{*}h$ **hence** $k{*}\ n \neq k{*}\ (m{*}h)$ **using** kz **by** *simp*
with h **have** *False* **by** (*simp add*: *mult-assoc*)**}**
hence $n = m * h$ **by** *blast*
thus *?thesis* **by** *blast*
qed

lemma *zdvd-zmult-cancel-disj*[*simp*]:
$(k{*}m)$ *dvd* $(k{*}n) = (k{=}0 \mid m \ dvd \ (n{::}int))$
by (*auto simp*: *zdvd-zmult-mono dest*: *zdvd-mult-cancel*)

theorem *ex-nat*: $(\exists\, x{::}nat.\ P\ x) = (\exists\, x{::}int.\ 0 <= x \wedge P\ (nat\ x))$
apply (*simp split add*: *split-nat*)
apply (*rule iffI*)
apply (*erule exE*)
apply (*rule-tac* $x = int\ x$ **in** *exI*)
apply *simp*
apply (*erule exE*)
apply (*rule-tac* $x = nat\ x$ **in** *exI*)
apply (*erule conjE*)
apply (*erule-tac* $x = nat\ x$ **in** *allE*)
apply *simp*
done

theorem *zdvd-int*: $(x \ dvd \ y) = (int\ x \ dvd \ int\ y)$
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* ≠(*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 ≤ *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*: $[|\ z\ dvd\ n;\ 0 < n\ |] ==> z \leq (n::int)$
apply (*rule-tac z=n* **in** *int-cases*)
apply (*auto simp add*: *dvd-int-iff*)
apply (*rule-tac z=z* **in** *int-cases*)
apply (*auto simp add*: *dvd-imp-le*)
done

lemma *zpower-zmod*: $((x::int)\ mod\ m)\hat{}y\ mod\ m = x\hat{}y\ mod\ m$
apply (*induct y*, *auto*)
apply (*rule zmod-zmult1-eq* [*THEN trans*])
apply (*simp* (*no-asm-simp*))
apply (*rule zmod-zmult-distrib* [*symmetric*])
done

lemma *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*:
$[\![0 < m;\ 0 < k]\!] \Longrightarrow k\ \hat{}\ m\ div\ k = (k::int)\ \hat{}\ (m - Suc\ 0)$
apply (*subgoal-tac* $k\ \hat{}\ m = k\ \hat{}\ ((m - 1) + 1)$)
apply (*erule ssubst*)
apply (*simp only*: *power-add*)

apply *simp-all*
done

by Brian Huffman

lemma *zminus-zmod*: $-((x::int) \bmod m) \bmod m = -x \bmod m$
by (*simp only*: *zmod-zminus1-eq-if mod-mod-trivial*)

lemma *zdiff-zmod-left*: $(x \bmod m - y) \bmod m = (x - y) \bmod (m::int)$
by (*simp only*: *diff-def zmod-zadd-left-eq* [*symmetric*])

lemma *zdiff-zmod-right*: $(x - y \bmod m) \bmod m = (x - y) \bmod (m::int)$
proof −
  have $(x + -(y \bmod m) \bmod m) \bmod m = (x + -y \bmod m) \bmod m$
    by (*simp only*: *zminus-zmod*)
  hence $(x + -(y \bmod m)) \bmod m = (x + -y) \bmod m$
    by (*simp only*: *zmod-zadd-right-eq* [*symmetric*])
  thus $(x - y \bmod m) \bmod m = (x - y) \bmod m$
    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

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*: $k \bmod (2::int) \neq (0::int) \Longrightarrow$
    of-int $k =$ *of-int* $(k \operatorname{div} 2 * 2 + 1)$
  proof −
    have $k \bmod 2 < 2$ **by** (*auto intro*: *pos-mod-bound*)
    moreover have $0 \leq k \bmod 2$ **by** (*auto intro*: *pos-mod-sign*)
    moreover assume $k \bmod 2 \neq 0$
    ultimately have $k \bmod 2 = 1$ **by** *arith*
    moreover have *of-int* $k =$ *of-int* $(k \operatorname{div} 2 * 2 + k \bmod 2)$ **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*: $\bigwedge x.$ $x *$ *of-int 2* $= x + x$
proof −
fix x
have *int2*: (*2*::*int*) = *1* + *1* **by** *arith*
show $x *$ *of-int 2* $= x + x$
unfolding *int2 of-int-add right-distrib* **by** *simp*
qed
from *aux1* **show** *?thesis* **by** (*auto simp add*: *divAlg-mod-div Let-def aux2 aux3*)
qed

end

code-modulename *SML*
IntDiv Integer

code-modulename *OCaml*
IntDiv Integer

code-modulename *Haskell*
IntDiv Integer

end

26 NatBin: Binary arithmetic for the natural numbers

theory *NatBin*
imports *IntDiv*
begin

Arithmetic for naturals is reduced to that for the non-negative integers.

instantiation *nat* :: *number*
begin

definition
nat-number-of-def [*code inline*]: *number-of v* = *nat* (*number-of v*)

instance ..

end

lemma [*code post*]:
nat (*number-of v*) = *number-of v*
unfolding *nat-number-of-def* **..**

abbreviation (*xsymbols*)
square :: *'a*::*power* => *'a* ((-^{2}) [*1000*] *999*) **where**
x^2 == *x^2*

notation (*latex* **output**)
square ((-^{2}) [*1000*] *999*)

notation (*HTML* **output**)
square ((-^{2}) [*1000*] *999*)

26.1 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*⟧ ⟹ *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

26.2 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*)

26.2.1 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* (*Int.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* (*Int.succ v*))
apply (*cut-tac n* = *0* **in** *Suc-nat-number-of-add*)
apply (*simp cong del*: *if-weak-cong*)
done

26.2.2 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*])

26.2.3 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*)

26.2.4 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*])

26.2.5 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*])

26.2.6 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*])

26.2.7 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;
⟩⟩

26.3 Comparisons

26.3.1 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

26.3.2 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*

26.4 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*: $(x::'a::recpower)\ \hat{}\ 3 = x * x * x$
apply (*subgoal-tac* $3 = Suc\ (Suc\ (Suc\ 0))$)
apply (*erule ssubst*)
apply (*simp add*: *power-Suc mult-ac*)
apply (*unfold nat-number-of-def*)
apply (*subst nat-eq-iff*)
apply *simp*
done

Squares of literal numerals will be evaluated.

lemmas *power2-eq-square-number-of* =
power2-eq-square [*of number-of w*, *standard*]
declare *power2-eq-square-number-of* [*simp*]

lemma *zero-le-power2*[*simp*]: $0 \leq (a^2::'a::\{ordered\text{-}idom,recpower\})$
by (*simp add*: *power2-eq-square*)

lemma *zero-less-power2*[*simp*]:
$(0 < a^2) = (a \neq (0::'a::\{ordered\text{-}idom,recpower\}))$
by (*force simp add*: *power2-eq-square zero-less-mult-iff linorder-neq-iff*)

lemma *power2-less-0*[*simp*]:
fixes $a :: 'a::\{ordered\text{-}idom,recpower\}$
shows $\sim (a^2 < 0)$
by (*force simp add*: *power2-eq-square mult-less-0-iff*)

lemma *zero-eq-power2*[*simp*]:
$(a^2 = 0) = (a = (0::'a::\{ordered\text{-}idom,recpower\}))$
by (*force simp add*: *power2-eq-square mult-eq-0-iff*)

lemma *abs-power2*[*simp*]:
$abs(a^2) = (a^2::'a::\{ordered\text{-}idom,recpower\})$
by (*simp add*: *power2-eq-square abs-mult abs-mult-self*)

lemma *power2-abs*[*simp*]:
$(abs\ a)^2 = (a^2::'a::\{ordered\text{-}idom,recpower\})$
by (*simp add*: *power2-eq-square abs-mult-self*)

lemma *power2-minus*[*simp*]:

$(-\ a)^2 = (a^2::'a::\{comm\text{-}ring\text{-}1,recpower\})$
by (*simp add*: *power2-eq-square*)

lemma *power2-le-imp-le*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 \leq y^2;\ 0 \leq y]\!] \Longrightarrow x \leq y$
unfolding *numeral-2-eq-2* **by** (*rule power-le-imp-le-base*)

lemma *power2-less-imp-less*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 < y^2;\ 0 \leq y]\!] \Longrightarrow x < y$
by (*rule power-less-imp-less-base*)

lemma *power2-eq-imp-eq*:
fixes $x\ y :: 'a::\{ordered\text{-}semidom,recpower\}$
shows $[\![x^2 = y^2;\ 0 \leq x;\ 0 \leq y]\!] \Longrightarrow x = y$
unfolding *numeral-2-eq-2* **by** (*erule* (*2*) *power-eq-imp-eq-base*, *simp*)

lemma *power-minus1-even*[*simp*]: $(-\ 1)\ \hat{}\ (2*n) = (1::'a::\{comm\text{-}ring\text{-}1,recpower\})$
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*]:
$0 \leq (a::'a::\{ordered\text{-}idom,recpower\})\ \hat{}\ (2*n)$
proof (*induct n*)
case *0*
show *?case* **by** (*simp add*: *zero-le-one*)
next
case (*Suc n*)
have $a\ \hat{}\ (2 * Suc\ n) = (a*a) * a\ \hat{}\ (2*n)$
by (*simp add*: *mult-ac power-add power2-eq-square*)
thus *?case*
by (*simp add*: *prems zero-le-mult-iff*)
qed

lemma *odd-power-less-zero*:
$(a::'a::\{ordered\text{-}idom,recpower\}) < 0 ==> a\ \hat{}\ Suc(2*n) < 0$
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

26.4.1 Nat

lemma *Suc-pred'*: *0 < n ==> n = Suc(n - 1)*
by (*simp add*: *numerals*)

lemmas *expand-Suc = Suc-pred'* [*of number-of v*, *standard*]

26.4.2 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* ^ *m* :: *nat*) = (*if m=0 then 1 else p* ∗ (*p* ^ (*m* − *1*)))
apply (*case-tac m*)
apply (*simp-all add*: *numerals*)
done

26.5 Comparisons involving (0::nat)

Simplification already does $n < (0::'a)$, $n \leq (0::'a)$ and $(0::'a) \leq n$.

lemma *eq-number-of-0* [*simp*]:
(*number-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*)

26.6 Comparisons involving *Suc*

lemma *eq-number-of-Suc* [*simp*]:
(*number-of v* = *Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else nat pv = *n*)
apply (*simp only*: *simp-thms Let-def neg-eq-less-0 linorder-not-less*
number-of-pred nat-number-of-def
split add: *split-if*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply (*auto simp add*: *nat-eq-iff*)
done

lemma *Suc-eq-number-of* [*simp*]:
(*Suc n* = *number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then False else nat pv = *n*)
by (*rule trans* [*OF eq-sym-conv eq-number-of-Suc*])

lemma *less-number-of-Suc* [*simp*]:
(*number-of v < Suc n*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then True else nat pv < n)
apply (*simp only*: *simp-thms Let-def neg-eq-less-0 linorder-not-less*
number-of-pred nat-number-of-def
split add: *split-if*)
apply (*rule-tac x = number-of v* **in** *spec*)
apply (*auto simp add*: *nat-less-iff*)
done

lemma *less-Suc-number-of* [*simp*]:
(*Suc n < number-of v*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then False else n < nat pv)
apply (*simp only*: *simp-thms Let-def neg-eq-less-0 linorder-not-less*
number-of-pred nat-number-of-def
split add: *split-if*)
apply (*rule-tac x = number-of v* **in** *spec*)
apply (*auto simp add*: *zless-nat-eq-int-zless*)
done

lemma *le-number-of-Suc* [*simp*]:
(*number-of v <= Suc n*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then True else nat pv <= n)
by (*simp add*: *Let-def less-Suc-number-of linorder-not-less* [*symmetric*])

lemma *le-Suc-number-of* [*simp*]:
(*Suc n <= number-of v*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then False else n <= nat pv)
by (*simp add*: *Let-def less-number-of-Suc linorder-not-less* [*symmetric*])

lemma *eq-number-of-Pls-Min*: (*Numeral0* ::*int*) ~= *number-of Int.Min*
by *auto*

26.7 Max and Min Combined with *Suc*

lemma *max-number-of-Suc* [*simp*]:
max (*Suc n*) (*number-of v*) =
(*let pv = number-of* (*Int.pred v*) *in*
if neg pv then Suc n else Suc(*max n* (*nat pv*)))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
split add: *split-if nat.split*)
apply (*rule-tac x = number-of v* **in** *spec*)
apply *auto*
done

lemma *max-Suc-number-of* [*simp*]:
max (*number-of v*) (*Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then Suc n else Suc(*max* (*nat pv*) *n*))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
split add: *split-if nat.split*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply *auto*
done

lemma *min-number-of-Suc* [*simp*]:
min (*Suc n*) (*number-of v*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then 0 else Suc(*min n* (*nat pv*)))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
split add: *split-if nat.split*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply *auto*
done

lemma *min-Suc-number-of* [*simp*]:
min (*number-of v*) (*Suc n*) =
(*let pv* = *number-of* (*Int.pred v*) *in*
if neg pv then 0 else Suc(*min* (*nat pv*) *n*))
apply (*simp only*: *Let-def neg-eq-less-0 number-of-pred nat-number-of-def*
split add: *split-if nat.split*)
apply (*rule-tac x* = *number-of v* **in** *spec*)
apply *auto*
done

26.8 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* (*Int.Bit0 w*) = (*let w* = *z* ^ (*number-of w*) *in w* * *w*)
unfolding *Let-def nat-number-of-def number-of-Bit0*
apply (*rule-tac x* = *number-of w* **in** *spec*, *clarify*)
apply (*case-tac* (*0*::*int*) <= *x*)
apply (*auto simp add*: *nat-mult-distrib power-even-eq power2-eq-square*)
done

lemma *power-number-of-odd*:
fixes *z* :: *'a*::{*number-ring*,*recpower*}
shows *z* ^ *number-of* (*Int.Bit1 w*) = (*if* (*0*::*int*) <= *number-of w*
then (*let w* = *z* ^ (*number-of w*) *in z* * *w* * *w*) *else 1*)
unfolding *Let-def nat-number-of-def number-of-Bit1*
apply (*rule-tac x* = *number-of w* **in** *spec*, *auto*)
apply (*simp only*: *nat-add-distrib nat-mult-distrib*)
apply *simp*
apply (*auto simp add*: *nat-add-distrib nat-mult-distrib power-even-eq power2-eq-square neg-nat power-Suc*)
done

lemmas *zpower-number-of-even* = *power-number-of-even* [**where** *'a*=*int*]
lemmas *zpower-number-of-odd* = *power-number-of-odd* [**where** *'a*=*int*]

lemmas *power-number-of-even-number-of* [*simp*] =
power-number-of-even [*of number-of v*, *standard*]

lemmas *power-number-of-odd-number-of* [*simp*] =
power-number-of-odd [*of number-of v*, *standard*]

ML
⟪
val numeral-ss = @{*simpset*} *addsimps* @{*thms numerals*};

val nat-bin-arith-setup =
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*,
@{*thm not-neg-number-of-Pls*}, @{*thm neg-number-of-Min*},
@{*thm neg-number-of-Bit0*}, @{*thm neg-number-of-Bit1*}]})
⟫

declaration ⟪ *K nat-bin-arith-setup* ⟫

declare *split-div*[*of - - number-of k*, *standard*, *arith-split*]
declare *split-mod*[*of - - number-of k*, *standard*, *arith-split*]

lemma *nat-number-of-Pls*: *Numeral0* = (*0*::*nat*)
  by (*simp add*: *number-of-Pls nat-number-of-def*)

lemma *nat-number-of-Min*: *number-of Int.Min* = (*0*::*nat*)
  apply (*simp only*: *number-of-Min nat-number-of-def nat-zminus-int*)
  done

lemma *nat-number-of-Bit0*:
    number-of (*Int.Bit0 w*) = (*let n*::*nat* = *number-of w in n* + *n*)
  apply (*simp only*: *nat-number-of-def Let-def*)
  apply (*cases neg* (*number-of w* :: *int*))
   apply (*simp add*: *neg-nat neg-number-of-Bit0*)
  apply (*rule int-int-eq* [*THEN iffD1*])
  apply (*simp only*: *not-neg-nat neg-number-of-Bit0 int-Suc zadd-int* [*symmetric*]
simp-thms)
  apply (*simp only*: *number-of-Bit0 zadd-assoc*)
  apply *simp*
  done

lemma *nat-number-of-Bit1*:
  number-of (*Int.Bit1 w*) =
    (*if neg* (*number-of w* :: *int*) *then 0*
     else let n = *number-of w in Suc* (*n* + *n*))
  apply (*simp only*: *nat-number-of-def Let-def split*: *split-if*)
  apply (*intro conjI impI*)
   apply (*simp add*: *neg-nat neg-number-of-Bit1*)
  apply (*rule int-int-eq* [*THEN iffD1*])
  apply (*simp only*: *not-neg-nat neg-number-of-Bit1 int-Suc zadd-int* [*symmetric*]
simp-thms)
  apply (*simp only*: *number-of-Bit1 zadd-assoc*)
  done

lemmas *nat-number* =
  nat-number-of-Pls nat-number-of-Min
  nat-number-of-Bit0 nat-number-of-Bit1

lemma *Let-Suc* [*simp*]: *Let* (*Suc n*) *f* == *f* (*Suc n*)
  by (*simp add*: *Let-def*)

lemma *power-m1-even*: (*−1*) ^ (*2∗n*) = (*1*::*'a*::{*number-ring*,*recpower*})
by (*simp add*: *power-mult power-Suc*)

lemma *power-m1-odd*: (*−1*) ^ *Suc*(*2∗n*) = (*−1*::*'a*::{*number-ring*,*recpower*})
by (*simp add*: *power-mult power-Suc*)

26.9 Literal arithmetic and *of-nat*

lemma *of-nat-double*:
$0 \leq x \Longrightarrow$ *of-nat* (*nat* (2 * *x*)) = *of-nat* (*nat x*) + *of-nat* (*nat x*)
by (*simp only*: *mult-2 nat-add-distrib of-nat-add*)

lemma *nat-numeral-m1-eq-0*: $-1 = (0::nat)$
by (*simp only*: *nat-number-of-def*)

lemma *of-nat-number-of-lemma*:
of-nat (*number-of v* :: *nat*) =
(*if* 0 ≤ (*number-of v* :: *int*)
then (*number-of v* :: 'a :: *number-ring*)
else 0)
by (*simp add*: *int-number-of-def nat-number-of-def number-of-eq of-nat-nat*)

lemma *of-nat-number-of-eq* [*simp*]:
of-nat (*number-of v* :: *nat*) =
(*if neg* (*number-of v* :: *int*) *then* 0
else (*number-of v* :: 'a :: *number-ring*))
by (*simp only*: *of-nat-number-of-lemma neg-def*, *simp*)

26.10 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*

26.10.1 For *combine-numerals*

lemma *left-add-mult-distrib*: *i***u* + (*j***u* + *k*) = (*i*+*j*)**u* + (*k*::*nat*)
by (*simp add*: *add-mult-distrib*)

26.10.2 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*)

26.10.3 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*)

26.10.4 For *cancel-factor*

lemma *nat-mult-le-cancel-disj*: $(k*m <= k*n) = ((0::nat) < k \longrightarrow m<=n)$
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*)

end

27 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

27.1 Semiring normalization

setup *NormalizerData.setup*

locale *gb-semiring* =
fixes *add mul pwr r0 r1*
assumes *add-a*:$(add\ x\ (add\ y\ z) = add\ (add\ x\ y)\ z)$
and *add-c*: $add\ x\ y = add\ y\ x$ **and** *add-0*:$add\ r0\ x = x$
and *mul-a*:$mul\ x\ (mul\ y\ z) = mul\ (mul\ x\ y)\ z$ **and** *mul-c*:$mul\ x\ y = mul\ y\ x$
and *mul-1*:$mul\ r1\ x = x$ **and** *mul-0*:$mul\ r0\ x = r0$
and *mul-d*:$mul\ x\ (add\ y\ z) = add\ (mul\ x\ y)\ (mul\ x\ z)$
and *pwr-0*:$pwr\ x\ 0 = r1$ **and** *pwr-Suc*:$pwr\ x\ (Suc\ n) = mul\ x\ (pwr\ x\ n)$
begin

lemma *mul-pwr*:$mul\ (pwr\ x\ p)\ (pwr\ x\ q) = pwr\ x\ (p + q)$
proof (*induct p*)
case *0*
then show *?case* **by** (*auto simp add*: *pwr-0 mul-1*)

next
case *Suc*
from *this* [*symmetric*] **show** *?case*
by (*auto simp add*: *pwr-Suc mul-1 mul-a*)
qed

lemma *pwr-mul*: *pwr* (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*)
proof (*induct q arbitrary*: *x y*, *auto simp add*:*pwr-0 pwr-Suc mul-1*)
fix *q x y*
assume ⋀*x y*. *pwr* (*mul x y*) *q* = *mul* (*pwr x q*) (*pwr y q*)
have *mul* (*mul x y*) (*mul* (*pwr x q*) (*pwr y q*)) = *mul x* (*mul y* (*mul* (*pwr x q*) (*pwr y q*)))
by (*simp add*: *mul-a*)
also have ... = (*mul* (*mul y* (*mul* (*pwr y q*) (*pwr x q*))) *x*) **by** (*simp add*: *mul-c*)
also have ... = (*mul* (*mul y* (*pwr y q*)) (*mul* (*pwr x q*) *x*)) **by** (*simp add*: *mul-a*)
finally show *mul* (*mul x y*) (*mul* (*pwr x q*) (*pwr y q*)) =
mul (*mul x* (*pwr x q*)) (*mul y* (*pwr y q*)) **by** (*simp add*: *mul-c*)
qed

lemma *pwr-pwr*: *pwr* (*pwr x p*) *q* = *pwr x* (*p* ∗ *q*)
proof (*induct p arbitrary*: *q*)
case *0*
show *?case* **using** *pwr-Suc mul-1 pwr-0* **by** (*induct q*) *auto*
next
case *Suc*
thus *?case* **by** (*auto simp add*: *mul-pwr* [*symmetric*] *pwr-mul pwr-Suc*)
qed

27.1.1 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

lemmas *gb-semiring-axioms'* =
gb-semiring-axioms [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*]

end

interpretation *class-semiring*: *gb-semiring*
[*op* + *op* ∗ *op* ^ *0*::*'a*::{*comm-semiring-1*, *recpower*} *1*]
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-Bit1 iszero-number-of-Bit0 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.gb-semiring-axioms'} ⟫


locale gb-ring = gb-semiring +
  fixes sub :: 'a ⇒ 'a ⇒ 'a
    and neg :: 'a ⇒ 'a
```

assumes *neg-mul*: *neg x = mul (neg r1) x*
and *sub-add*: *sub x y = add x (neg y)*
begin

lemma *ring-ops*:
includes *meta-term-syntax*
shows *TERM (sub x y)* **and** *TERM (neg x)* **.**

lemmas *ring-rules = neg-mul sub-add*

lemmas *gb-ring-axioms' =*
gb-ring-axioms [normalizer
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules]*

end

interpretation *class-ring*: *gb-ring [op + op * op ^*
0::'a::{comm-semiring-1,recpower,number-ring} 1 op − uminus]
by *unfold-locales simp-all*

declaration ⟪ *normalizer-funs @{thm class-ring.gb-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

lemmas *gb-field-axioms' =*
gb-field-axioms [normalizer
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules]*

end

27.2 Groebner Bases

locale *semiringb* = *gb-semiring* +
  assumes *add-cancel*: *add* (*x*::'*a*) *y* = *add x z* ⟷ *y* = *z*
  and *add-mul-solve*: *add* (*mul w y*) (*mul x z*) =
    add (*mul w z*) (*mul x y*) ⟷ *w* = *x* ∨ *y* = *z*
begin

lemma *noteq-reduce*: *a* ≠ *b* ∧ *c* ≠ *d* ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
proof−
  have *a* ≠ *b* ∧ *c* ≠ *d* ⟷ ¬ (*a* = *b* ∨ *c* = *d*) **by** *simp*
  also have ... ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
    using *add-mul-solve* **by** *blast*
  finally show *a* ≠ *b* ∧ *c* ≠ *d* ⟷ *add* (*mul a c*) (*mul b d*) ≠ *add* (*mul a d*) (*mul b c*)
    by *simp*
qed

lemma *add-scale-eq-noteq*: ⟦*r* ≠ *r0* ; (*a* = *b*) ∧ ~(*c* = *d*)⟧
  ⟹ *add a* (*mul r c*) ≠ *add b* (*mul r d*)
proof(*clarify*)
  assume *nz*: *r*≠ *r0* **and** *cnd*: *c*≠*d*
    and *eq*: *add b* (*mul r c*) = *add b* (*mul r d*)
  hence *mul r c* = *mul r d* **using** *cnd add-cancel* **by** *simp*
  hence *add* (*mul r0 d*) (*mul r c*) = *add* (*mul r0 c*) (*mul r d*)
    using *mul-0 add-cancel* **by** *simp*
  thus *False* **using** *add-mul-solve nz cnd* **by** *simp*
qed

lemma *add-r0-iff*: *x* = *add x a* ⟷ *a* = *r0*
proof−
  have *a* = *r0* ⟷ *add x a* = *add x r0* **by** (*simp add*: *add-cancel*)
  thus *x* = *add x a* ⟷ *a* = *r0* **by** (*auto simp add*: *add-c add-0*)
qed

declare *gb-semiring-axioms'* [*normalizer del*]

lemmas *semiringb-axioms'* = *semiringb-axioms* [*normalizer*
  semiring ops: *semiring-ops*
  semiring rules: *semiring-rules*
  idom rules: *noteq-reduce add-scale-eq-noteq*]

end

locale *ringb* = *semiringb* + *gb-ring* +
  assumes *subr0-iff*: *sub x y* = *r0* ⟷ *x* = *y*

begin

declare *gb-ring-axioms'* [*normalizer del*]

lemmas *ringb-axioms'* = *ringb-axioms* [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*
idom rules: *noteq-reduce add-scale-eq-noteq*
ideal rules: *subr0-iff add-r0-iff*]

end

lemma *no-zero-divirors-neq0*:
assumes *az*: (*a*::'*a*::*no-zero-divisors*) ≠ *0*
and *ab*: *a*∗*b* = *0* **shows** *b* = *0*
proof −
{ assume *bz*: *b* ≠ *0*
from *no-zero-divisors* [*OF az bz*] *ab* **have** *False* **by** *blast* **}**
thus *b* = *0* **by** *blast*
qed

interpretation *class-ringb*: *ringb*
[*op* + *op* ∗ *op* ^ *0*::'*a*::{*idom*,*recpower*,*number-ring*} *1 op* − *uminus*]
proof(*unfold-locales*, *simp add*: *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.ringb-axioms'*} ⟫

interpretation *natgb*: *semiringb*
[*op* + *op* ∗ *op* ^ *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** $\exists k.\ y = z + k \wedge k>0$ **by** (*rule-tac* $x=y - z$ **in** *exI*, *auto*)
then obtain k **where** *kp*: $k>0$ **and** *yz*:$y = z + k$ **by** *blast*
from p **have** $(w * z + x * z) + w*k = (w * z + x*z) + x*k$ **by** (*simp add*: *yz 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 $(w * y + x * z = w * z + x * y) = (w = x \vee y = z)$ **by** *auto*
qed

declaration ⟪ *normalizer-funs* @{*thm natgb.semiringb-axioms'*} ⟫

locale *fieldgb* = *ringb* + *gb-field*
begin

declare *gb-field-axioms'* [*normalizer del*]

lemmas *fieldgb-axioms'* = *fieldgb-axioms* [*normalizer*
semiring ops: *semiring-ops*
semiring rules: *semiring-rules*
ring ops: *ring-ops*
ring rules: *ring-rules*
idom rules: *noteq-reduce add-scale-eq-noteq*
ideal rules: *subr0-iff add-r0-iff*]

end

lemmas *bool-simps* = *simp-thms*(*1*−*34*)
lemma *dnf*:
$(P \,\&\, (Q \mid R)) = ((P\&Q) \mid (P\&R))$ $((Q \mid R) \,\&\, P) = ((Q\&P) \mid (R\&P))$
$(P \wedge Q) = (Q \wedge P)$ $(P \vee Q) = (Q \vee P)$
by *blast*+

lemmas *weak-dnf-simps* = *dnf bool-simps*

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

lemma *PFalse*:

$P \equiv False \Longrightarrow \neg P$
$\neg P \Longrightarrow (P \equiv False)$
by *auto*

use *Tools/Groebner-Basis/groebner.ML*

method-setup *algebra* =
⟪
let
fun keyword k = Scan.lift (Args.$$$ k −− Args.colon) >> K ()
val addN = add
val delN = del
val any-keyword = keyword addN || keyword delN
val thms = Scan.repeat (Scan.unless any-keyword Attrib.multi-thm) >> flat;
in
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

28 Arith-Tools: Setup of arithmetic tools

theory *Arith-Tools*
imports *Groebner-Basis*
uses
~~/src/Provers/Arith/cancel-numeral-factor.ML
~~/src/Provers/Arith/extract-common-term.ML
int-factor-simprocs.ML
nat-simprocs.ML
begin

28.1 Simprocs for the Naturals

declaration ⟪ *K nat-simprocs-setup* ⟫

28.1.1 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* (*Int.pred* *v*)::*int*) = (*number-of* *v* = (*0*::*nat*))
apply (*subgoal-tac neg* (*number-of* (*Int.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* (*Int.pred* *v*))
apply (*subst Suc-diff-eq-diff-pred*)
apply *simp*
apply (*simp del*: *nat-numeral-1-eq-1*)
apply (*auto simp only*: *diff-nat-number-of less-0-number-of* [*symmetric*]
                  neg-number-of-pred-iff-0)
done

lemma *diff-Suc-eq-diff-pred*: *m* − *Suc* *n* = (*m* − *1*) − *n*
by (*simp add*: *numerals split add*: *nat-diff-split*)

28.1.2 For *nat-case* and *nat-rec*

lemma *nat-case-number-of* [*simp*]:
    nat-case *a* *f* (*number-of* *v*) =
      (*let* *pv* = *number-of* (*Int.pred* *v*) *in*
       if neg pv then a else f (*nat pv*))
by (*simp split add*: *nat.split add*: *Let-def neg-number-of-pred-iff-0*)

lemma *nat-case-add-eq-if* [*simp*]:
    nat-case *a* *f* ((*number-of* *v*) + *n*) =
      (*let* *pv* = *number-of* (*Int.pred* *v*) *in*
       if neg pv then nat-case a f n else f (*nat pv* + *n*))
apply (*subst add-eq-if*)
apply (*simp split add*: *nat.split*
          del: *nat-numeral-1-eq-1*
          add: *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* (*Int.pred* *v*) *in*
       if neg pv then a else f (*nat pv*) (*nat-rec a f* (*nat pv*)))
apply (*case-tac* (*number-of* *v*) ::*nat*)
apply (*simp-all* (*no-asm-simp*) *add*: *Let-def neg-number-of-pred-iff-0*)
apply (*simp split add*: *split-if-asm*)

done

lemma *nat-rec-add-eq-if* [*simp*]:
  nat-rec a f (*number-of v* + *n*) =
    (*let pv* = *number-of* (*Int.pred v*) *in*
     if neg pv then nat-rec a f n
           else f (*nat pv* + *n*) (*nat-rec a f* (*nat pv* + *n*)))
apply (*subst add-eq-if*)
apply (*simp split add*: *nat.split*
      del: *nat-numeral-1-eq-1*
      add: *numeral-1-eq-Suc-0* [*symmetric*] *Let-def neg-imp-number-of-eq-0*
        neg-number-of-pred-iff-0)
done

28.1.3 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* ==> *n* = *0* | *n* = *Suc 0*
by *arith*

lemma *div2-Suc-Suc* [*simp*]: *Suc*(*Suc m*) *div 2* = *Suc* (*m div 2*)
by *arith*

lemma *add-self-div-2* [*simp*]: (*m* + *m*) *div 2* = (*m*::*nat*)
by (*simp add*: *nat-mult-2* [*symmetric*])

lemma *mod2-Suc-Suc* [*simp*]: *Suc*(*Suc*(*m*)) *mod 2* = *m mod 2*
apply (*subgoal-tac m mod 2* < *2*)
apply (*erule less-2-cases* [*THEN disjE*])
apply (*simp-all* (*no-asm-simp*) *add*: *Let-def mod-Suc nat-1*)
done

lemma *mod2-gr-0* [*simp*]: !!*m*::*nat*. (*0* < *m mod 2*) = (*m mod 2* = *1*)
apply (*subgoal-tac m mod 2* < *2*)
apply (*force simp del*: *mod-less-divisor*, *simp*)
done

Removal of Small Numerals: 0, 1 and (in additive positions) 2

lemma *add-2-eq-Suc* [*simp*]: *2 + n = Suc (Suc n)*
by *simp*

lemma *add-2-eq-Suc'* [*simp*]: *n + 2 = Suc (Suc n)*
by *simp*

Can be used to eliminate long strings of Sucs, but not by default

lemma *Suc3-eq-add-3*: *Suc (Suc (Suc n)) = 3 + n*
by *simp*

These lemmas collapse some needless occurrences of Suc: at least three Sucs, since two and fewer are rewritten back to Suc again! We already have some rules to simplify operands smaller than 3.

lemma *div-Suc-eq-div-add3* [*simp*]: *m div (Suc (Suc (Suc n))) = m div (3+n)*
by (*simp add*: *Suc3-eq-add-3*)

lemma *mod-Suc-eq-mod-add3* [*simp*]: *m mod (Suc (Suc (Suc n))) = m mod (3+n)*
by (*simp add*: *Suc3-eq-add-3*)

lemma *Suc-div-eq-add3-div*: *(Suc (Suc (Suc m))) div n = (3+m) div n*
by (*simp add*: *Suc3-eq-add-3*)

lemma *Suc-mod-eq-add3-mod*: *(Suc (Suc (Suc m))) mod n = (3+m) mod n*
by (*simp add*: *Suc3-eq-add-3*)

lemmas *Suc-div-eq-add3-div-number-of =*
Suc-div-eq-add3-div [*of - number-of v*, *standard*]
declare *Suc-div-eq-add3-div-number-of* [*simp*]

lemmas *Suc-mod-eq-add3-mod-number-of =*
Suc-mod-eq-add3-mod [*of - number-of v*, *standard*]
declare *Suc-mod-eq-add3-mod-number-of* [*simp*]

28.1.4 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-idom*,*number-ring*}
shows (*1* < − *b*) = (*b* < −*1*)
by *auto*

lemma *le-minus-iff-1* [*simp*,*noatp*]:
fixes *b*::'*b*::{*ordered-idom*,*number-ring*}
shows (*1* ≤ − *b*) = (*b* ≤ −*1*)
by *auto*

lemma *equation-minus-iff-1* [*simp*,*noatp*]:
fixes *b*::'*b*::*number-ring*
shows (*1* = − *b*) = (*b* = −*1*)
by (*subst equation-minus-iff*, *auto*)

lemma *minus-less-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::{*ordered-idom*,*number-ring*}
shows (− *a* < *1*) = (−*1* < *a*)
by *auto*

lemma *minus-le-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::{*ordered-idom*,*number-ring*}
shows (− *a* ≤ *1*) = (−*1* ≤ *a*)
by *auto*

lemma *minus-equation-iff-1* [*simp*,*noatp*]:
fixes *a*::'*b*::*number-ring*
shows (− *a* = *1*) = (*a* = −*1*)
by (*subst minus-equation-iff*, *auto*)

Cancellation of constant factors in comparisons (< and ≤)

lemmas *mult-less-cancel-left-number-of* =
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-of1* [*simp*] = *le-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-le-eq-number-of1* [*simp*] = *divide-le-eq* [*of - number-of w*, *standard*]
lemmas *less-divide-eq-number-of1* [*simp*] = *less-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-less-eq-number-of1* [*simp*] = *divide-less-eq* [*of - number-of w*, *standard*]
lemmas *eq-divide-eq-number-of1* [*simp*] = *eq-divide-eq* [*of - - number-of w*, *standard*]
lemmas *divide-eq-eq-number-of1* [*simp*] = *divide-eq-eq* [*of - number-of w*, *standard*]

28.1.5 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*};
⟫

28.2 Groebner Bases for fields

interpretation *class-fieldgb*:
fieldgb[*op + op * op ^ 0*::'*a*::{*field*,*recpower*,*number-ring*} *1 op − uminus op / inverse*] **apply** (*unfold-locales*) **by** (*simp-all add*: *divide-inverse*)

lemma *divide-Numeral1*: (*x*::'*a*::{*field*,*number-ring*}) / *Numeral1 = x* **by** *simp*
lemma *divide-Numeral0*: (*x*::'*a*::{*field*,*number-ring*, *division-by-zero*}) / *Numeral0 = 0*
by *simp*
lemma *mult-frac-frac*: ((*x*::'*a*::{*field*,*division-by-zero*}) / *y*) * (*z* / *w*) = (*x**z*) / (*y**w*)
by *simp*
lemma *mult-frac-num*: ((*x*::'*a*::{*field*, *division-by-zero*}) / *y*) * *z* = (*x**z*) / *y*
by *simp*
lemma *mult-num-frac*: ((*x*::'*a*::{*field*, *division-by-zero*}) / *y*) * *z* = (*x**z*) / *y*
by *simp*

lemma *Numeral1-eq1-nat*: (*1*::*nat*) = *Numeral1* **by** *simp*

lemma *add-frac-num*: *y*≠ *0* ⟹ (*x*::'*a*::{*field*, *division-by-zero*}) / *y* + *z* = (*x* + *z***y*) / *y*
by (*simp add*: *add-divide-distrib*)
lemma *add-num-frac*: *y*≠ *0* ⟹ *z* + (*x*::'*a*::{*field*, *division-by-zero*}) / *y* = (*x* + *z***y*) / *y*
by (*simp add*: *add-divide-distrib*)

ML⟪
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])
                 @ @{thms arith-simps} @ nat-arith @ @{thms 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.fieldgb-axioms'}
   {is-const = K numeral-is-const,
    dest-const = K dest-const,
    mk-const = mk-const,
    conv = K (K comp-conv)}
end;
⟩⟩
```

declaration⟪ *fieldgb-declaration* ⟫
end

29 SetInterval: Set intervals

theory *SetInterval*
imports *Int*
begin

context *ord*
begin
definition
lessThan :: *'a* => *'a set* ((*1*{..<-})) **where**
{..<*u*} == {*x*. *x* < *u*}

definition
atMost :: *'a* => *'a set* ((*1*{..-})) **where**
{..*u*} == {*x*. *x* ≤ *u*}

definition
greaterThan :: *'a* => *'a set* ((*1*{-<..})) **where**
{*l*<..} == {*x*. *l*<*x*}

definition
atLeast :: *'a* => *'a set* ((*1*{-..})) **where**
{*l*..} == {*x*. *l*≤*x*}

definition
greaterThanLessThan :: *'a* => *'a* => *'a set* ((*1*{-<..<-})) **where**
{*l*<..<*u*} == {*l*<..} *Int* {..<*u*}

definition
atLeastLessThan :: *'a* => *'a* => *'a set* ((*1*{-..<-})) **where**
{*l*..<*u*} == {*l*..} *Int* {..<*u*}

definition
greaterThanAtMost :: *'a* => *'a* => *'a set* ((*1*{-<..-})) **where**
{*l*<..*u*} == {*l*<..} *Int* {..*u*}

definition
atLeastAtMost :: *'a* => *'a* => *'a set* ((*1*{-..-})) **where**
{*l*..*u*} == {*l*..} *Int* {..*u*}

end

A note of warning when using {..<*n*} on type *nat*: it is equivalent to {*0*..<*n*} but some lemmas involving {*m*..<*n*} may not exist in {..<*n*}-form as well.

syntax

```
  @UNION-le   :: nat => nat => 'b set => 'b set      ((3UN -<=-./ -) 10)
  @UNION-less :: nat => nat => 'b set => 'b set      ((3UN -<-./ -) 10)
  @INTER-le   :: nat => nat => 'b set => 'b set      ((3INT -<=-./ -) 10)
  @INTER-less :: nat => nat => 'b set => 'b set      ((3INT -<-./ -) 10)

syntax (input)
  @UNION-le   :: nat => nat => 'b set => 'b set      ((3⋃ -≤-./ -) 10)
  @UNION-less :: nat => nat => 'b set => 'b set      ((3⋃ -<-./ -) 10)
  @INTER-le   :: nat => nat => 'b set => 'b set      ((3⋂ -≤-./ -) 10)
  @INTER-less :: nat => nat => 'b set => 'b set      ((3⋂ -<-./ -) 10)

syntax (xsymbols)
  @UNION-le   :: nat ⇒ nat => 'b set => 'b set       ((3⋃(00_ ≤ _)/ -) 10)
  @UNION-less :: nat ⇒ nat => 'b set => 'b set       ((3⋃(00_ < _)/ -) 10)
  @INTER-le   :: nat ⇒ nat => 'b set => 'b set       ((3⋂(00_ ≤ _)/ -) 10)
  @INTER-less :: nat ⇒ nat => 'b set => 'b set       ((3⋂(00_ < _)/ -) 10)

translations
  UN i<=n. A  == UN i:{..n}. A
  UN i<n. A   == UN i:{..<n}. A
  INT i<=n. A == INT i:{..n}. A
  INT i<n. A  == INT i:{..<n}. A
```

29.1 Various equivalences

```
lemma (in ord) lessThan-iff [iff]: (i: lessThan k) = (i<k)
by (simp add: lessThan-def)

lemma Compl-lessThan [simp]:
    !!k:: 'a::linorder. -lessThan k = atLeast k
apply (auto simp add: lessThan-def atLeast-def)
done

lemma single-Diff-lessThan [simp]: !!k:: 'a::order. {k} - lessThan k = {k}
by auto

lemma (in ord) greaterThan-iff [iff]: (i: greaterThan k) = (k<i)
by (simp add: greaterThan-def)

lemma Compl-greaterThan [simp]:
    !!k:: 'a::linorder. -greaterThan k = atMost k
  by (auto simp add: greaterThan-def atMost-def)

lemma Compl-atMost [simp]: !!k:: 'a::linorder. -atMost k = greaterThan k
apply (subst Compl-greaterThan [symmetric])
apply (rule double-complement)
done

lemma (in ord) atLeast-iff [iff]: (i: atLeast k) = (k<=i)
```

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

lemma *Compl-atLeast* [*simp*]:
  !!*k*:: *'a*::*linorder*. −*atLeast k* = *lessThan k*
 by (*auto simp add*: *lessThan-def atLeast-def*)

lemma (**in** *ord*) *atMost-iff* [*iff*]: (*i*: *atMost k*) = (*i*<=*k*)
by (*simp add*: *atMost-def*)

lemma *atMost-Int-atLeast*: !!*n*:: *'a*::*order*. *atMost n Int atLeast n* = {*n*}
by (*blast intro*: *order-antisym*)

29.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-subset-iff* [*iff*]:
    (*atLeast x* ⊆ *atLeast y*) = (*y* ≤ (*x*::*'a*::*order*))
by (*blast intro*: *order-trans*)

lemma *atLeast-eq-iff* [*iff*]:
    (*atLeast x* = *atLeast y*) = (*x* = (*y*::*'a*::*linorder*))
by (*blast intro*: *order-antisym order-trans*)

lemma *greaterThan-subset-iff* [*iff*]:
    (*greaterThan x* ⊆ *greaterThan y*) = (*y* ≤ (*x*::*'a*::*linorder*))
apply (*auto simp add*: *greaterThan-def*)
 apply (*subst linorder-not-less* [*symmetric*], *blast*)
done

lemma *greaterThan-eq-iff* [*iff*]:
    (*greaterThan x* = *greaterThan y*) = (*x* = (*y*::*'a*::*linorder*))
apply (*rule iffI*)
 apply (*erule equalityE*)
 apply (*simp-all add*: *greaterThan-subset-iff*)
done

lemma *atMost-subset-iff* [*iff*]: (*atMost x* ⊆ *atMost y*) = (*x* ≤ (*y*::*'a*::*order*))
by (*blast intro*: *order-trans*)

lemma *atMost-eq-iff* [*iff*]: (*atMost x* = *atMost y*) = (*x* = (*y*::*'a*::*linorder*))
by (*blast intro*: *order-antisym order-trans*)

lemma *lessThan-subset-iff* [*iff*]:
    (*lessThan x* ⊆ *lessThan y*) = (*x* ≤ (*y*::*'a*::*linorder*))
apply (*auto simp add*: *lessThan-def*)
 apply (*subst linorder-not-less* [*symmetric*], *blast*)
done

lemma *lessThan-eq-iff* [*iff*]:
    (*lessThan x* = *lessThan y*) = (*x* = (*y*::*'a*::*linorder*))

apply (*rule iffI*)
apply (*erule equalityE*)
apply (*simp-all add*: *lessThan-subset-iff*)
done

29.3 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

29.3.1 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* ≤ *m* ==> {*m*..<*n*} = {}
by (*auto simp add*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-empty*[*simp*]:*l* ≤ *k* ==> {*k*<..*l*} = {}
by(*auto simp*:*greaterThanAtMost-def greaterThan-def atMost-def*)

lemma *greaterThanLessThan-empty*[*simp*]:*l* ≤ *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

29.4 Intervals of natural numbers

29.4.1 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*

29.4.2 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*

29.4.3 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*

29.4.4 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*

29.4.5 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*)

29.4.6 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:
{*m*..<*Suc n*} = (*if m* ≤ *n then insert n* {*m*..<*n*} *else* {})
by (*auto simp add*: *atLeastLessThan-def*)

lemma *atLeastLessThan-singleton* [*simp*]: {*m*..<*Suc m*} = {*m*}
by (*auto simp add*: *atLeastLessThan-def*)

lemma *atLeastLessThanSuc-atLeastAtMost*: {*l*..<*Suc u*} = {*l*..*u*}
by (*simp add*: *lessThan-Suc-atMost atLeastAtMost-def atLeastLessThan-def*)

lemma *atLeastSucAtMost-greaterThanAtMost*: {*Suc l*..*u*} = {*l*<..*u*}
by (*simp add*: *atLeast-Suc-greaterThan atLeastAtMost-def*
greaterThanAtMost-def)

lemma *atLeastSucLessThan-greaterThanLessThan*: {*Suc l*..<*u*} = {*l*<..<*u*}

by (*simp add*: *atLeast-Suc-greaterThan atLeastLessThan-def*
greaterThanLessThan-def)

lemma *atLeastAtMostSuc-conv*: $m \leq Suc\ n \Longrightarrow \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$
by (*auto simp add*: *atLeastAtMost-def*)

29.4.7 Image

lemma *image-add-atLeastAtMost*:
$(\%n{::}nat.\ n+k)\ `\ \{i..j\} = \{i+k..j+k\}$ (**is** *?A* = *?B*)
proof
show *?A* $\subseteq$ *?B* **by** *auto*
next
show *?B* $\subseteq$ *?A*
proof
fix *n* **assume** *a*: *n* : *?B*
hence *n* − *k* : $\{i..j\}$ **by** *auto*
moreover have *n* = (*n* − *k*) + *k* **using** *a* **by** *auto*
ultimately show *n* : *?A* **by** *blast*
qed
qed

lemma *image-add-atLeastLessThan*:
$(\%n{::}nat.\ n+k)\ `\ \{i..<j\} = \{i+k..<j+k\}$ (**is** *?A* = *?B*)
proof
show *?A* $\subseteq$ *?B* **by** *auto*
next
show *?B* $\subseteq$ *?A*
proof
fix *n* **assume** *a*: *n* : *?B*
hence *n* − *k* : $\{i..<j\}$ **by** *auto*
moreover have *n* = (*n* − *k*) + *k* **using** *a* **by** *auto*
ultimately show *n* : *?A* **by** *blast*
qed
qed

corollary *image-Suc-atLeastAtMost*[*simp*]:
Suc ` $\{i..j\} = \{Suc\ i..Suc\ j\}$
using *image-add-atLeastAtMost*[**where** *k*=*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

29.4.8 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*} ⟹ 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

29.4.9 Cardinality

lemma *card-lessThan* [*simp*]: *card* {..<*u*} = *u*
by (*induct u*, *simp-all add*: *lessThan-Suc*)

lemma *card-atMost* [*simp*]: *card* {..*u*} = *Suc u*
by (*simp add*: *lessThan-Suc-atMost* [*THEN sym*])

lemma *card-atLeastLessThan* [*simp*]: *card* {*l*..<*u*} = *u* − *l*
apply (*subgoal-tac card* {*l*..<*u*} = *card* {..<*u*−*l*})
apply (*erule ssubst*, *rule card-lessThan*)
apply (*subgoal-tac* (%*x*. *x* + *l*) ' {..<*u*−*l*} = {*l*..<*u*})
apply (*erule subst*)
apply (*rule card-image*)
apply (*simp add*: *inj-on-def*)
apply (*auto simp add*: *image-def atLeastLessThan-def lessThan-def*)
apply (*rule-tac x* = *x* − *l* **in** *exI*)
apply *arith*
done

lemma *card-atLeastAtMost* [*simp*]: *card* {*l*..*u*} = *Suc u* − *l*
by (*subst atLeastLessThanSuc-atLeastAtMost* [*THEN sym*], *simp*)

lemma *card-greaterThanAtMost* [*simp*]: *card* {*l*<..*u*} = *u* − *l*
by (*subst atLeastSucAtMost-greaterThanAtMost* [*THEN sym*], *simp*)

lemma *card-greaterThanLessThan* [*simp*]: *card* {*l*<..<*u*} = *u* − *Suc l*
by (*subst atLeastSucLessThan-greaterThanLessThan* [*THEN sym*], *simp*)

lemma *ex-bij-betw-nat-finite*:
finite M ⟹ ∃*h*. *bij-betw h* {*0*..<*card M*} *M*
apply(*drule finite-imp-nat-seg-image-inj-on*)
apply(*auto simp*:*atLeast0LessThan*[*symmetric*] *lessThan-def*[*symmetric*] *card-image bij-betw-def*)
done

lemma *ex-bij-betw-finite-nat*:
finite M ⟹ ∃*h*. *bij-betw h M* {*0*..<*card M*}
by (*blast dest*: *ex-bij-betw-nat-finite bij-betw-inv*)

29.5 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*)

29.5.1 Finiteness

lemma *image-atLeastZeroLessThan-int*: $0 \leq u \Longrightarrow$
$\{(0::int)..<u\} = int\ `\ \{..<nat\ u\}$
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* $0 \leq u$)
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*)

29.5.2 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*)

29.6 Lemmas useful with the summation operator setsum

For examples, see Algebra/poly/UnivPoly2.thy

29.6.1 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*

29.6.2 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*

29.6.3 Some Differences

lemma *ivl-diff*[*simp*]:
$i \leq n \Longrightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$
by(*auto*)

29.6.4 Some Subset Conditions

lemma *ivl-subset* [*simp*,*noatp*]:
$(\{i..<j\} \subseteq \{m..<n\}) = (j \leq i \mid m \leq i \ \& \ j \leq (n::'a::linorder))$
apply(*auto simp*:*linorder-not-le*)
apply(*rule ccontr*)
apply(*insert linorder-le-less-linear*[*of i n*])
apply(*clarsimp simp*:*linorder-not-le*)
apply(*fastsimp*)
done

29.7 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*∑*-* ≤ *-* -) [*0*,*0*,*10*] *10*)

translations
∑ *x*=*a*..*b*. *t* == *setsum* (%*x*. *t*) {*a*..*b*}
∑ *x*=*a*..<*b*. *t* == *setsum* (%*x*. *t*) {*a*..<*b*}
∑ *i*≤*n*. *t* == *setsum* (λ*i*. *t*) {..*n*}
∑ *i*<*n*. *t* == *setsum* (λ*i*. *t*) {..<*n*}

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	LaTeX
∑ *x*∈{*a*..*b*}. *e*	∑ *x* = *a*..*b*. *e*	$\sum_{x=a}^{b} e$
∑ *x*∈{*a*..<*b*}. *e*	∑ *x* = *a*..<*b*. *e*	$\sum_{x=a}^{<b} e$
∑ *x*∈{..*b*}. *e*	∑ *x*≤*b*. *e*	$\sum_{x \le b} e$
∑ *x*∈{..<*b*}. *e*	∑ *x*<*b*. *e*	$\sum_{x < b} e$

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to *latex-sum* (e.g. via *mode* = *latex-sum* in antiquotations). It is not the default LaTeX output because it only works well with italic-style formulae, not tt-style.

Note that for uniformity on *nat* it is better to use ∑ *x* = *0*..<*n*. *e* rather than ∑ *x*<*n*. *e*: *setsum* may not provide all lemmas available for {*m*..<*n*} also in the special form for {..<*n*}.

This congruence rule should be used for sums over intervals as the standard theorem *setsum-cong* does not work well with the simplifier who adds the unsimplified premise *x* ∈ *B* to the context.

lemma *setsum-ivl-cong*:
⟦*a* = *c*; *b* = *d*; !!*x*. ⟦ *c* ≤ *x*; *x* < *d* ⟧ ⟹ *f x* = *g x* ⟧ ⟹
setsum f {*a*..<*b*} = *setsum g* {*c*..<*d*}
by(*rule setsum-cong*, *simp-all*)

lemma *setsum-atMost-Suc*[*simp*]: (∑ *i* ≤ *Suc n*. *f i*) = (∑ *i* ≤ *n*. *f i*) + *f*(*Suc n*)
by (*simp add*:*atMost-Suc add-ac*)

lemma *setsum-lessThan-Suc*[*simp*]: (∑ *i* < *Suc n*. *f i*) = (∑ *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 ≤ n*; *n ≤ p* ⟧ ⟹
setsum f {m..<n} + setsum f {n..<p} = setsum f {m..<p::*nat}*
by (*simp add*:*setsum-Un-disjoint*[*symmetric*] *ivl-disj-int ivl-disj-un*)

lemma *setsum-diff-nat-ivl*:
fixes *f* :: *nat* ⇒ *'a*::*ab-group-add*
shows ⟦ *m ≤ n*; *n ≤ p* ⟧ ⟹
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

29.8 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 (∑ *x*∈{*m..n*}. *P x*) = *P m* + (∑ *x*∈{*m<..n*}. *P x*) (**is** *?lhs = ?rhs*)
proof −
from *mn*
have {*m..n*} = {*m*} ∪ {*m<..n*}
by (*auto intro*: *ivl-disj-un-singleton*)
hence *?lhs* = (∑ *x*∈{*m*} ∪ {*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 $\ldots = (\sum x{\in}\{0..m - 1\}.\ P\ x)$
by (*cases m*) (*auto simp add*: *atLeastLessThanSuc-atLeastAtMost*)
also
have $\ldots = P\ 0 + (\sum x{\in}\{0{<}..m - 1\}.\ P\ x)$
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

29.9 The formula for geometric sums

lemma *geometric-sum*:
$x \mathbin{\sim=} 1 \Longrightarrow (\sum i{=}0..{<}n.\ x\ \hat{}\ i) =$
$(x\ \hat{}\ n - 1)\ /\ (x - 1{::}'a{::}\{field, recpower\})$
by (*induct n*) (*simp-all add*:*field-simps power-Suc*)

29.10 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-nat i* **and** *?n* = *of-nat n*
have

```
    (∑ i∈{..<n}. a+?I i*d) =
     ((∑ i∈{..<n}. a) + (∑ i∈{..<n}. ?I i*d))
    by (rule setsum-addf)
  also from ngt1 have ... = ?n*a + (∑ i∈{..<n}. ?I i*d) by simp
  also from ngt1 have ... = (?n*a + d*(∑ i∈{1..<n}. ?I i))
    by (simp add: setsum-right-distrib setsum-head-upt mult-ac)
  also have (1+1)*... = (1+1)*?n*a + d*(1+1)*(∑ i∈{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)*(∑ i∈{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 ¬(n > 1)
  hence n = 1 ∨ n = 0 by auto
  thus ?thesis by (auto simp: mult-ac right-distrib)
qed

lemma arith-series-nat:
  Suc (Suc 0) * (∑ i∈{..<n}. a+i*d) = n * (a + (a+(n - 1)*d))
proof -
  have
    ((1::nat) + 1) * (∑ i∈{..<n::nat}. a + of-nat(i)*d) =
    of-nat(n) * (a + (a + of-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) * (∑ i∈{..<n}. a + of-nat i * d) =
  of-nat n * (a + (a + of-nat(n - 1)*d))
proof -
  have
    ((1::int) + 1) * (∑ i∈{..<n}. a + of-nat i * d) =
    of-nat(n) * (a + (a + of-nat(n - 1)*d))
    by (rule arith-series-general)
  thus ?thesis by simp
qed

lemma sum-diff-distrib:
  fixes P::nat⇒nat
  shows
  ∀x. Q x ≤ P x ⟹
  (∑ x<n. P x) - (∑ x<n. Q x) = (∑ x<n. P x - Q x)
proof (induct n)
```

case *0* **show** *?case* **by** *simp*
next
case (*Suc n*)

let *?lhs* = $(\sum x{<}n.\ P\ x) - (\sum x{<}n.\ Q\ x)$
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

end

30 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/qelim.ML
(*Tools/Qelim/cooper.ML*)
(*Tools/Qelim/presburger.ML*)
begin

setup *CooperData.setup*

30.1 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*)+

30.2 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*

30.3 Cooper's Theorem $-\infty$ and $+\infty$ Version

30.3.1 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*

30.3.2 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** $?L = (?R1 \vee ?R2)$)
proof−
{assume *?R2* **hence** *?L* **by** *blast***}**
moreover
{assume *H*:*?R1* **hence** *?L* **using** *minusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] **by** *simp***}**

moreover
{ fix *x*
assume *P*: *P x* **and** *H*: ¬ *?R2*
{fix *y* **assume** ¬ (∃*j*∈{*1..D*}. ∃ *b*∈*B*. *P* (*b* + *j*)) **and** *P*: *P y*
hence ~(*EX* (*j*::*int*) : {*1..D*}. *EX* (*b*::*int*) : *B*. *y* = *b*+*j*) **by** *auto*
with *nb P* **have** *P* (*y* − *D*) **by** *auto* **}**
hence *ALL x.*~(*EX* (*j*::*int*) : {*1..D*}. *EX* (*b*::*int*) : *B*. *P*(*b*+*j*)) −−> *P* (*x*) −−> *P* (*x* − *D*) **by** *blast*
with *H P* **have** *th*: ∀ *x*. *P x* ⟶ *P* (*x* − *D*) **by** *auto*
from *p1* **obtain** *z* **where** *z*: *ALL x*. *x* < *z* −−> (*P x* = *P*′ *x*) **by** *blast*
let *?y* = *x* − (|*x* − *z*| + *1*)∗*D*
have *zp*: *0* <= (|*x* − *z*| + *1*) **by** *arith*
from *dp* **have** *yz*: *?y* < *z* **using** *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

30.3.3 The +∞ Version

lemma *plusinfinity*:
assumes *dpos*: (*0*::*int*) < *d* **and**
P1eqP1: ∀ *x k*. *P*′ *x* = *P*′(*x* − *k*∗*d*) **and** *ePeqP1*: ∃ *z*. ∀ *x*>*z*. *P x* = *P*′ *x*
shows (∃ *x*. *P*′ *x*) ⟶ (∃ *x*. *P x*)
proof
assume *eP1*: *EX x*. *P*′ *x*
then obtain *x* **where** *P1*: *P*′ *x* **..**
from *ePeqP1* **obtain** *z* **where** *P1eqP*: ∀ *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 ... = *P*(*?w*) **using** *w P1eqP* **by** *blast*
finally have *P ?w* **using** *P1* **by** *blast*
thus *EX x*. *P x* **..**
qed

lemma *incr-mult-lemma*:
assumes *dpos*: (*0*::*int*) < *d* **and** *plus*: *ALL x*::*int*. *P x* ⟶ *P*(*x* + *d*) **and** *knneg*: *0* <= *k*
shows *ALL x*. *P x* ⟶ *P*(*x* + *k*∗*d*)
using *knneg*
proof (*induct rule*:*int-ge-induct*)
case *base* **thus** *?case* **by** *simp*
next
case (*step i*)

```
 {fix x
  have P x ⟶ P (x + i * d) using step.hyps by blast
  also have ... ⟶ P(x + (i + 1) * d) using plus[THEN spec, of x + i * d]
    by (simp add:int-distrib zadd-ac)
  ultimately have P x ⟶ P(x + (i + 1) * d) by blast}
 thus ?case ..
qed

lemma cppi:
 assumes dp: 0 < D and p1:∃ z. ∀ x> z. P x = P' x
 and nb:∀ x.(∀ j∈ {1..D}. ∀(b::int) ∈ A. x ≠ b - j) --> P (x) --> P (x +
D)
 and pd: ∀ x k. P' x= P' (x-k*D)
 shows (∃ x. P x) = ((∃ j∈ {1..D} . P' j) | (∃ j ∈ {1..D}.∃ b∈ A. P (b - j)))
(is ?L = (?R1 ∨ ?R2))
proof-
 {assume ?R2 hence ?L by blast}
 moreover
 {assume H:?R1 hence ?L using plusinfinity[OF dp pd p1] periodic-finite-ex[OF
dp pd] by simp}
 moreover
 { fix x
  assume P: P x and H: ¬ ?R2
  {fix y assume ¬ (∃j∈{1..D}. ∃ b∈A. P (b - j)) and P: P y
   hence ~(EX (j::int) : {1..D}. EX (b::int) : A. y = b - j) by auto
   with nb P have P (y + D) by auto }
  hence ALL x.~(EX (j::int) : {1..D}. EX (b::int) : A. P(b-j)) --> P (x)
--> P (x + D) by blast
  with H P have th: ∀ x. P x ⟶ P (x + D) by auto
  from p1 obtain z where z: ALL x. x > z --> (P x = P' x) by blast
  let ?y = x + (|x - z| + 1)*D
  have zp: 0 <= (|x - z| + 1) by arith
  from dp have yz: ?y > z using incr-lemma[OF dp] by simp
  from z[rule-format, OF yz] incr-mult-lemma[OF dp th zp, rule-format, OF P]
have th2: P' ?y by auto
  with periodic-finite-ex[OF dp pd]
  have ?R1 by blast}
 ultimately show ?thesis by blast
qed

lemma simp-from-to: {i..j::int} = (if j < i then {} else insert i {i+1..j})
apply(simp add:atLeastAtMost-def atLeast-def atMost-def)
apply(fastsimp)
done

theorem unity-coeff-ex: (∃ (x::'a::{semiring-0,Divides.div}). P (l * x)) ≡ (∃ x. l
dvd (x + 0) ∧ 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*) ≠ *0*
shows ((*m*::*int*) *dvd t*) ≡ (*k*∗*m dvd k*∗*t*)
using *not0* **by** (*simp add*: *dvd-def*)

lemma *uminus-dvd-conv*: (*d dvd* (*t*::*int*)) ≡ (−*d dvd t*) (*d dvd* (*t*::*int*)) ≡ (*d dvd* −*t*)
by *simp-all*

Theorems for transforming predicates on nat to predicates on *int*

lemma *all-nat*: (∀ *x*::*nat*. *P x*) = (∀ *x*::*int*. *0* <= *x* ⟶ *P* (*nat x*))
by (*simp split add*: *split-nat*)

lemma *ex-nat*: (∃ *x*::*nat*. *P x*) = (∃ *x*::*int*. *0* <= *x* ∧ *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* ≤ *x* ⟶ *P* (*int x* − *int y*)) ∧ (*x* < *y* ⟶ *P 0*))
by (*case-tac y* ≤ *x*, *simp-all add*: *zdiff-int*)

lemma *number-of1*: (*0*::*int*) <= *number-of n* ⟹ (*0*::*int*) <= *number-of* (*Int.Bit0 n*) ∧ (*0*::*int*) <= *number-of* (*Int.Bit1 n*)
by *simp*
lemma *number-of2*: (*0*::*int*) <= *Numeral0* **by** *simp*
lemma *Suc-plus1*: *Suc n* = *n* + *1* **by** *simp*

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*: (*0* <= *x* ⟹ *P* = *P'*) ⟹ (*0* <= (*x*::*int*) ⟶ *P*) = (*0* <= *x* ⟶ *P'*) **by** *simp*

theorem *conj-le-cong*: (*0* <= *x* ⟹ *P* = *P'*) ⟹ (*0* <= (*x*::*int*) ∧ *P*) = (*0* <= *x* ∧ *P'*)
by (*simp cong*: *conj-cong*)
lemma *int-eq-number-of-eq*:

(((*number-of v*)::*int*) = (*number-of w*)) = *iszero* ((*number-of* (*v* + (*uminus w*)))::*int*)
by *simp*

lemma *mod-eq0-dvd-iff*[*presburger*]: (*m*::*nat*) *mod n* = *0* ⟷ *n dvd m*
unfolding *dvd-eq-mod-eq-0*[*symmetric*] **..**

lemma *zmod-eq0-zdvd-iff*[*presburger*]: (*m*::*int*) *mod n* = *0* ⟷ *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) ⟷ ¬ 2 dvd m  by presburger
lemma [presburger]: m mod 2 = Suc 0 ⟷ ¬ 2 dvd m  by presburger
lemma [presburger]: m mod (Suc (Suc 0)) = (1::nat) ⟷ ¬ 2 dvd m  by presburger
lemma [presburger]: m mod (Suc (Suc 0)) = Suc 0 ⟷ ¬ 2 dvd m  by presburger
lemma [presburger]: m mod 2 = (1::int) ⟷ ¬ 2 dvd m  by presburger


lemma zdvd-period:
  fixes a d :: int
  assumes advdd: a dvd d
  shows a dvd (x + t) ⟷ a dvd ((x + c * d) + t)
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

end
```

31 Refute: Refute

```
theory Refute
imports Inductive
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

32 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

33 Recdef: TFL: recursive function definitions

theory *Recdef*
imports *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

* This form avoids giant explosions in proofs. NOTE USE OF ==

lemma *def-wfrec*: [| *f==wfrec r H*; *wf(r)* |] ==> *f(a) = H* (*cut f r a*) *a*
apply *auto*
apply (*blast intro*: *wfrec*)
done

lemma *tfl-wf-induct*: *ALL R. wf R* −−>
(*ALL P.* (*ALL x.* (*ALL y.* (*y*,*x*):*R* −−> *P y*) −−> *P x*) −−> (*ALL x. P x*))
apply *clarify*
apply (*rule-tac r = R* **and** *P = P* **and** *a = x* **in** *wf-induct*, *assumption*, *blast*)
done

lemma *tfl-cut-apply*: $ALL\ f\ R.\ (x,a){:}R\ {-}{-}{>}\ (cut\ f\ R\ a)(x) = f(x)$
apply *clarify*
apply (*rule cut-apply*, *assumption*)
done

lemma *tfl-wfrec*:
$ALL\ M\ R\ f.\ (f{=}wfrec\ R\ M)\ {-}{-}{>}\ wf\ R\ {-}{-}{>}\ (ALL\ x.\ f\ x = M\ (cut\ f\ R\ x)\ x)$
apply *clarify*
apply (*erule wfrec*)
done

lemma *tfl-eq-True*: $(x = True)\ {-}{-}{>}\ x$
by *blast*

lemma *tfl-rev-eq-mp*: $(x = y)\ {-}{-}{>}\ y\ {-}{-}{>}\ x$
by *blast*

lemma *tfl-simp-thm*: $(x\ {-}{-}{>}\ y)\ {-}{-}{>}\ (x = x')\ {-}{-}{>}\ (x'\ {-}{-}{>}\ y)$
by *blast*

lemma *tfl-P-imp-P-iff-True*: $P\ {=}{=}{>}\ P = True$
by *blast*

lemma *tfl-imp-trans*: $(A\ {-}{-}{>}\ B)\ {=}{=}{>}\ (B\ {-}{-}{>}\ C)\ {=}{=}{>}\ (A\ {-}{-}{>}\ C)$
by *blast*

lemma *tfl-disj-assoc*: $(a \vee b) \vee c == a \vee (b \vee c)$
by *simp*

lemma *tfl-disjE*: $P \vee Q\ {=}{=}{>}\ P\ {-}{-}{>}\ R\ {=}{=}{>}\ Q\ {-}{-}{>}\ R\ {=}{=}{>}\ R$
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

34 Datatype: Analogues of the Cartesian Product and Disjoint Sum for Datatypes

theory *Datatype*
imports *Finite-Set Wellfounded*
begin

lemma *size-bool* [*code func*]:
size (*b*::*bool*) = *0* **by** (*cases b*) *auto*

declare *prod.size* [*noatp*]

typedef (*Node*)
('*a*,'*b*) *node* = {*p*. *EX f x k*. *p* = (*f*::*nat*=>'*b*+*nat*, *x*::'*a*+*nat*) & *f k* = *Inr 0*}
— it is a subtype of (*nat*=>'*b*+*nat*) * ('*a*+*nat*)
by *auto*

Datatypes will be represented by sets of type *node*

types '*a item* = ('*a*, *unit*) *node set*
('*a*, '*b*) *dtree* = ('*a*, '*b*) *node set*

consts
Push :: [('*b* + *nat*), *nat* => ('*b* + *nat*)] => (*nat* => ('*b* + *nat*))

Push-Node :: [('*b* + *nat*), ('*a*, '*b*) *node*] => ('*a*, '*b*) *node*
ndepth :: ('*a*, '*b*) *node* => *nat*

Atom :: $('a + nat) \Rightarrow ('a, 'b)\ dtree$
Leaf :: $'a \Rightarrow ('a, 'b)\ dtree$
Numb :: $nat \Rightarrow ('a, 'b)\ dtree$
Scons :: $[('a, 'b)\ dtree, ('a, 'b)\ dtree] \Rightarrow ('a, 'b)\ dtree$
In0 :: $('a, 'b)\ dtree \Rightarrow ('a, 'b)\ dtree$
In1 :: $('a, 'b)\ dtree \Rightarrow ('a, 'b)\ dtree$
Lim :: $('b \Rightarrow ('a, 'b)\ dtree) \Rightarrow ('a, 'b)\ dtree$

ntrunc :: $[nat, ('a, 'b)\ dtree] \Rightarrow ('a, 'b)\ dtree$

uprod :: $[('a, 'b)\ dtree\ set, ('a, 'b)\ dtree\ set] \Rightarrow ('a, 'b)\ dtree\ set$
usum :: $[('a, 'b)\ dtree\ set, ('a, 'b)\ dtree\ set] \Rightarrow ('a, 'b)\ dtree\ set$

Split :: $[[('a, 'b)\ dtree, ('a, 'b)\ dtree] \Rightarrow 'c, ('a, 'b)\ dtree] \Rightarrow 'c$
Case :: $[[('a, 'b)\ dtree] \Rightarrow 'c, [('a, 'b)\ dtree] \Rightarrow 'c, ('a, 'b)\ dtree] \Rightarrow 'c$

dprod :: $[(('a, 'b)\ dtree * ('a, 'b)\ dtree)set, (('a, 'b)\ dtree * ('a, 'b)\ dtree)set] \Rightarrow (('a, 'b)\ dtree * ('a, 'b)\ dtree)set$
dsum :: $[(('a, 'b)\ dtree * ('a, 'b)\ dtree)set, (('a, 'b)\ dtree * ('a, 'b)\ dtree)set] \Rightarrow (('a, 'b)\ dtree * ('a, 'b)\ dtree)set$

defs

Push-Node-def: *Push-Node* $==$ $(\%n\ x.\ Abs\text{-}Node\ (apfst\ (Push\ n)\ (Rep\text{-}Node\ x)))$

Push-def: *Push* $==$ $(\%b\ h.\ nat\text{-}case\ b\ h)$

Atom-def: *Atom* $==$ $(\%x.\ \{Abs\text{-}Node((\%k.\ Inr\ 0, x))\})$
Scons-def: *Scons M N* $==$ $(Push\text{-}Node\ (Inr\ 1)\ `\ M)\ Un\ (Push\text{-}Node\ (Inr\ (Suc\ 1))\ `\ N)$

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* $\{z.\ ?\ x.\ z = Push\text{-}Node\ (Inl\ x)\ `\ (f\ x)\}$

ndepth-def: $ndepth(n) == (\%(f,x).\ LEAST\ k.\ f\ k = Inr\ 0)\ (Rep\text{-}Node\ n)$
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: $Case\ c\ d\ M == THE\ u.\ (EX\ x\ .\ M = In0(x)\ \&\ u = c(x))$
$\quad | (EX\ y\ .\ M = In1(y)\ \&\ u = d(y))$

dprod-def: $dprod\ r\ s == UN\ (x,x'){:}r.\ UN\ (y,y'){:}s.\ \{(Scons\ x\ y,\ Scons\ x'\ y')\}$

dsum-def: $dsum\ r\ s == (UN\ (x,x'){:}r.\ \{(In0(x),In0(x'))\})\ Un$
$\quad (UN\ (y,y'){:}s.\ \{(In1(y),In1(y'))\})$

lemma *apfst-convE*:
$[|\ q = apfst\ f\ p;\ !!x\ y.\ [|\ p = (x,y);\ q = (f(x),y)\ |] ==> R\ |] ==> R$
by (*force simp add*: *apfst-def*)

lemma *Push-inject1*: $Push\ i\ f = Push\ j\ g\ ==> i{=}j$
apply (*simp add*: *Push-def expand-fun-eq*)
apply (*drule-tac x=0* **in** *spec*, *simp*)
done

lemma *Push-inject2*: $Push\ i\ f = Push\ j\ g\ ==> f{=}g$
apply (*auto simp add*: *Push-def expand-fun-eq*)
apply (*drule-tac x=Suc x* **in** *spec*, *simp*)
done

lemma *Push-inject*:
$[|\ Push\ i\ f = Push\ j\ g;\ [|\ i{=}j;\ f{=}g\ |] ==> P\ |] ==> P$
by (*blast dest*: *Push-inject1 Push-inject2*)

lemma *Push-neq-K0*: $Push\ (Inr\ (Suc\ k))\ f = (\%z.\ Inr\ 0) ==> P$
by (*auto simp add*: *Push-def expand-fun-eq split*: *nat.split-asm*)

lemmas *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*, *standard*]

lemma *Node-K0-I*: $(\%k.\ Inr\ 0,\ a) : Node$
by (*simp add*: *Node-def*)

lemma *Node-Push-I*: p: $Node \Longrightarrow apfst\ (Push\ i)\ p : Node$
apply (*simp add*: *Node-def Push-def*)
apply (*fast intro*!: *apfst-conv nat-case-Suc* [*THEN trans*])
done

34.1 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: $Scons\ M\ N \neq Atom(a)$
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* $\neq$ *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* $\neq$ *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*) $\neq$ *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*)

34.2 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*

35 Datatypes

35.1 Representing sums

rep-datatype *sum*
  distinct *Inl-not-Inr Inr-not-Inl*
  inject *Inl-eq Inr-eq*
  induction *sum-induct*

lemma *sum-case-KK*[*simp*]: *sum-case* (%*x*. *a*) (%*x*. *a*) = (%*x*. *a*)
  by (*rule ext*) (*simp split*: *sum.split*)

lemma *surjective-sum*: *sum-case* (%*x*::'*a*. *f* (*Inl x*)) (%*y*::'*b*. *f* (*Inr y*)) *s* = *f*(*s*)
  apply (*rule-tac s* = *s* **in** *sumE*)
   apply (*erule ssubst*)
   apply (*rule sum.cases*(*1*))
  apply (*erule ssubst*)
  apply (*rule sum.cases*(*2*))
  done

lemma *sum-case-weak-cong*: *s* = *t* ==> *sum-case f g s* = *sum-case f g t*
  — Prevents simplification of *f* and *g*: much faster.
  by *simp*

lemma *sum-case-inject*:
  sum-case f1 f2 = *sum-case g1 g2* ==> (*f1* = *g1* ==> *f2* = *g2* ==> *P*) ==> *P*
proof −
  assume *a*: *sum-case f1 f2* = *sum-case g1 g2*
  assume *r*: *f1* = *g1* ==> *f2* = *g2* ==> *P*
  show *P*
    apply (*rule r*)
     apply (*rule ext*)
     apply (*cut-tac x* = *Inl x* **in** *a* [*THEN fun-cong*], *simp*)
    apply (*rule ext*)
    apply (*cut-tac x* = *Inr x* **in** *a* [*THEN fun-cong*], *simp*)
    done
qed

constdefs
  Suml :: ('*a* => '*c*) => '*a* + '*b* => '*c*

Suml == (%*f*. *sum-case f arbitrary*)

Sumr :: ('*b* => '*c*) => '*a* + '*b* => '*c*
Sumr == *sum-case arbitrary*

lemma *Suml-inject*: *Suml f* = *Suml g* ==> *f* = *g*
by (*unfold Suml-def*) (*erule sum-case-inject*)

lemma *Sumr-inject*: *Sumr f* = *Sumr g* ==> *f* = *g*
by (*unfold Sumr-def*) (*erule sum-case-inject*)

hide (**open**) *const Suml Sumr*

35.2 The option datatype

datatype '*a option* = *None* | *Some* '*a*

lemma *not-None-eq* [*iff*]: (*x* ~= *None*) = (*EX y*. *x* = *Some y*)
by (*induct x*) *auto*

lemma *not-Some-eq* [*iff*]: (*ALL y*. *x* ~= *Some y*) = (*x* = *None*)
by (*induct x*) *auto*

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

lemma *option-caseE*:
assumes *c*: (*case x of None* => *P* | *Some y* => *Q y*)
obtains
(*None*) *x* = *None* **and** *P*
| (*Some*) *y* **where** *x* = *Some y* **and** *Q y*
using *c* **by** (*cases x*) *simp-all*

lemma *insert-None-conv-UNIV*: *insert None* (*range Some*) = *UNIV*
by (*rule set-ext*, *case-tac x*) *auto*

instance *option* :: (*finite*) *finite*
by *default* (*simp add*: *insert-None-conv-UNIV* [*symmetric*])

35.2.1 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*

declaration ⟪ *fn* - =>
Classical.map-cs (*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*

definition
option-map :: ('*a* ⇒ '*b*) ⇒ '*a option* ⇒ '*b option*
where
[*code func del*]: *option-map* = (%*f y*. *case y of None* => *None* | *Some x* => *Some* (*f x*))

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*)

35.2.2 Code generator setup

definition
is-none :: '*a option* ⇒ *bool* **where**
is-none-none [*code post*, *symmetric*, *code inline*]: *is-none x* ⟷ *x* = *None*

lemma *is-none-code* [*code*]:
  shows *is-none None* ⟷ *True*
    and *is-none* (*Some x*) ⟷ *False*
  unfolding *is-none-none* [*symmetric*] **by** *simp-all*

hide (**open**) *const is-none*

code-type *option*
  (*SML - option*)
  (*OCaml - option*)
  (*Haskell Maybe -*)

code-const *None* **and** *Some*
  (*SML NONE* **and** *SOME*)
  (*OCaml None* **and** *Some -*)
  (*Haskell Nothing* **and** *Just*)

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

code-const *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

36 Extraction: Program extraction for HOL

theory *Extraction*
imports *Datatype*
uses *Tools/rewrite-hol-proof.ML*
begin

36.1 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
```

36.2 Type of extracted program

extract-type

$typeof\ (Trueprop\ P) \equiv typeof\ P$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('Q))$

$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE(Null))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('P \Rightarrow 'Q))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$typeof\ (\forall x.\ P\ x) \equiv Type\ (TYPE(Null))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$typeof\ (\forall x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$typeof\ (\exists x{::}'a.\ P\ x) \equiv Type\ (TYPE('a))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$typeof\ (\exists x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

36.3 Realizability

realizability

$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv \neg\ realizes\ Null\ P$

typeof P ≡ Type (TYPE('P)) ⟹
realizes t (¬ P) ≡ (∀ x::'P. ¬ realizes x P)

typeof (P::bool) ≡ Type (TYPE(Null)) ⟹
typeof Q ≡ Type (TYPE(Null)) ⟹
realizes t (P = Q) ≡ realizes Null P = realizes Null Q

(realizes t (P = Q)) ≡ (realizes t ((P ⟶ Q) ∧ (Q ⟶ P)))

36.4 Computational content of basic inference rules

theorem *disjE-realizer*:
assumes *r*: *case x of Inl p ⇒ P p | Inr q ⇒ Q q*
and *r1*: ⋀*p*. *P p ⟹ R (f p)* **and** *r2*: ⋀*q*. *Q q ⟹ R (g q)*
shows *R (case x of Inl p ⇒ f p | Inr q ⇒ g q)*
proof *(cases x)*
case *Inl*
with *r* **show** *?thesis* **by** *simp (rule r1)*
next
case *Inr*
with *r* **show** *?thesis* **by** *simp (rule r2)*
qed

theorem *disjE-realizer2*:
assumes *r*: *case x of None ⇒ P | Some q ⇒ Q q*
and *r1*: *P ⟹ R f* **and** *r2*: ⋀*q*. *Q q ⟹ R (g q)*
shows *R (case x of None ⇒ f | Some q ⇒ g q)*
proof *(cases x)*
case *None*
with *r* **show** *?thesis* **by** *simp (rule r1)*
next
case *Some*
with *r* **show** *?thesis* **by** *simp (rule r2)*
qed

theorem *disjE-realizer3*:
assumes *r*: *case x of Left ⇒ P | Right ⇒ Q*
and *r1*: *P ⟹ R f* **and** *r2*: *Q ⟹ R g*
shows *R (case x of Left ⇒ f | Right ⇒ g)*
proof *(cases x)*
case *Left*
with *r* **show** *?thesis* **by** *simp (rule r1)*
next
case *Right*
with *r* **show** *?thesis* **by** *simp (rule r2)*
qed

theorem *conjI-realizer*:

$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (fst\ (p,\ q)) \wedge Q\ (snd\ (p,\ q))$
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 $(cases\ p)$ $(simp\ add\text{: }Let\text{-}def)$

theorem *exE-realizer'*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q$ **by** $(cases\ p)$ *simp*

realizers
$impI\ (P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ P\ Q\ pq\ (h\text{: }\text{-}).\ allI \cdot \text{-} \bullet (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (h \cdot x))$

$impI\ (P)$: *Null*
$\Lambda\ P\ Q\ (h\text{: }\text{-}).\ allI \cdot \text{-} \bullet (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (h \cdot x))$

$impI\ (Q)$: $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ impI \cdot \text{-} \cdot \text{-}$

impI: *Null impI*

$mp\ (P,\ Q)$: $\lambda pq.\ pq$
$\Lambda\ P\ Q\ pq\ (h\text{: }\text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (spec \cdot \text{-} \cdot p \bullet h)$

$mp\ (P)$: *Null*
$\Lambda\ P\ Q\ (h\text{: }\text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (spec \cdot \text{-} \cdot p \bullet h)$

$mp\ (Q)$: $\lambda q.\ q\ \Lambda\ P\ Q\ q.\ mp \cdot \text{-} \cdot \text{-}$

mp: *Null mp*

$allI\ (P)$: $\lambda p.\ p\ \Lambda\ P\ p.\ allI \cdot \text{-}$

allI: *Null allI*

$spec\ (P)$: $\lambda x\ p.\ p\ x\ \Lambda\ P\ x\ p.\ spec \cdot \text{-} \cdot x$

spec: *Null spec*

$exI\ (P)$: $\lambda x\ p.\ (x,\ p)\ \Lambda\ P\ x\ p.\ exI\text{-}realizer \cdot P \cdot p \cdot x$

exI: $\lambda x.\ x\ \Lambda\ P\ x\ (h\text{: }\text{-}).\ h$

$exE\ (P,\ Q)$: $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
$\Lambda\ P\ Q\ p\ (h\text{: }\text{-})\ pq.\ exE\text{-}realizer \cdot P \cdot p \cdot Q \cdot pq \bullet h$

exE (*P*): *Null*
  Λ *P Q p*. *exE-realizer'* · - · - · -

exE (*Q*): λ*x pq*. *pq x*
  Λ *P Q x* (*h1*: -) *pq* (*h2*: -). *h2* · *x* • *h1*

exE: *Null*
  Λ *P Q x* (*h1*: -) (*h2*: -). *h2* · *x* • *h1*

conjI (*P*, *Q*): *Pair*
  Λ *P Q p* (*h*: -) *q*. *conjI-realizer* · *P* · *p* · *Q* · *q* • *h*

conjI (*P*): λ*p*. *p*
  Λ *P Q p*. *conjI* · - · -

conjI (*Q*): λ*q*. *q*
  Λ *P Q* (*h*: -) *q*. *conjI* · - · - • *h*

conjI: *Null conjI*

conjunct1 (*P*, *Q*): *fst*
  Λ *P Q pq*. *conjunct1* · - · -

conjunct1 (*P*): λ*p*. *p*
  Λ *P Q p*. *conjunct1* · - · -

conjunct1 (*Q*): *Null*
  Λ *P Q q*. *conjunct1* · - · -

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*
  Λ *P Q pq*. *conjunct2* · - · -

conjunct2 (*P*): *Null*
  Λ *P Q p*. *conjunct2* · - · -

conjunct2 (*Q*): λ*p*. *p*
  Λ *P Q p*. *conjunct2* · - · -

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*
  Λ *P Q p*. *iffD2* · - · - • (*sum.cases-1* · *P* · - · *p*)

disjI1 (*P*): *Some*
  Λ *P Q p*. *iffD2* · - · - • (*option.cases-2* · - · *P* · *p*)

disjI1 (*Q*): *None*

$\Lambda\ P\ Q.\ \mathit{iffD2} \cdot - \cdot - \cdot - \bullet (\mathit{option.cases\text{-}1} \cdot - \cdot - \cdot -)$

disjI1: *Left*
$\Lambda\ P\ Q.\ \mathit{iffD2} \cdot - \cdot - \cdot - \bullet (\mathit{sumbool.cases\text{-}1} \cdot - \cdot - \cdot -)$

disjI2 $(P,\ Q)$: *Inr*
$\Lambda\ Q\ P\ q.\ \mathit{iffD2} \cdot - \cdot - \cdot - \bullet (\mathit{sum.cases\text{-}2} \cdot - \cdot - \cdot Q \cdot q)$

disjI2 (P): *None*
$\Lambda\ Q\ P.\ \mathit{iffD2} \cdot - \cdot - \cdot - \bullet (\mathit{option.cases\text{-}1} \cdot - \cdot - \cdot -)$

disjI2 (Q): *Some*
$\Lambda\ Q\ P\ q.\ \mathit{iffD2} \cdot - \cdot - \cdot - \bullet (\mathit{option.cases\text{-}2} \cdot - \cdot - \cdot Q \cdot q)$

disjI2: *Right*
$\Lambda\ Q\ P.\ \mathit{iffD2} \cdot - \cdot - \cdot - \bullet (\mathit{sumbool.cases\text{-}2} \cdot - \cdot - \cdot -)$

disjE $(P,\ Q,\ R)$: $\lambda pq\ pr\ qr.$
$(\mathit{case}\ pq\ \mathit{of}\ \mathit{Inl}\ p \Rightarrow pr\ p \mid \mathit{Inr}\ q \Rightarrow qr\ q)$
$\Lambda\ P\ Q\ R\ pq\ (h1\text{:}\ -)\ pr\ (h2\text{:}\ -)\ qr.$
$\mathit{disjE\text{-}realizer} \cdot - \cdot - \cdot - \cdot pq \cdot R \cdot pr \cdot qr \bullet h1 \bullet h2$

disjE $(Q,\ R)$: $\lambda pq\ pr\ qr.$
$(\mathit{case}\ pq\ \mathit{of}\ \mathit{None} \Rightarrow pr \mid \mathit{Some}\ q \Rightarrow qr\ q)$
$\Lambda\ P\ Q\ R\ pq\ (h1\text{:}\ -)\ pr\ (h2\text{:}\ -)\ qr.$
$\mathit{disjE\text{-}realizer2} \cdot - \cdot - \cdot - \cdot pq \cdot R \cdot pr \cdot qr \bullet h1 \bullet h2$

disjE $(P,\ R)$: $\lambda pq\ pr\ qr.$
$(\mathit{case}\ pq\ \mathit{of}\ \mathit{None} \Rightarrow qr \mid \mathit{Some}\ p \Rightarrow pr\ p)$
$\Lambda\ P\ Q\ R\ pq\ (h1\text{:}\ -)\ pr\ (h2\text{:}\ -)\ qr\ (h3\text{:}\ -).$
$\mathit{disjE\text{-}realizer2} \cdot - \cdot - \cdot - \cdot pq \cdot R \cdot qr \cdot pr \bullet h1 \bullet h3 \bullet h2$

disjE (R): $\lambda pq\ pr\ qr.$
$(\mathit{case}\ pq\ \mathit{of}\ \mathit{Left} \Rightarrow pr \mid \mathit{Right} \Rightarrow qr)$
$\Lambda\ P\ Q\ R\ pq\ (h1\text{:}\ -)\ pr\ (h2\text{:}\ -)\ qr.$
$\mathit{disjE\text{-}realizer3} \cdot - \cdot - \cdot - \cdot pq \cdot R \cdot pr \cdot qr \bullet h1 \bullet h2$

disjE $(P,\ Q)$: *Null*
$\Lambda\ P\ Q\ R\ pq.\ \mathit{disjE\text{-}realizer} \cdot - \cdot - \cdot - \cdot pq \cdot (\lambda x.\ R) \cdot - \cdot - \cdot -$

disjE (Q): *Null*
$\Lambda\ P\ Q\ R\ pq.\ \mathit{disjE\text{-}realizer2} \cdot - \cdot - \cdot - \cdot pq \cdot (\lambda x.\ R) \cdot - \cdot - \cdot -$

disjE (P): *Null*
$\Lambda\ P\ Q\ R\ pq\ (h1\text{:}\ -)\ (h2\text{:}\ -)\ (h3\text{:}\ -).$
$\mathit{disjE\text{-}realizer2} \cdot - \cdot - \cdot - \cdot pq \cdot (\lambda x.\ R) \cdot - \cdot - \cdot - \bullet h1 \bullet h3 \bullet h2$

disjE: *Null*
$\Lambda\ P\ Q\ R\ pq.\ \mathit{disjE\text{-}realizer3} \cdot - \cdot - \cdot - \cdot pq \cdot (\lambda x.\ R) \cdot - \cdot - \cdot -$

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*): *λp*. *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*): *λs 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*): *λp*. *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*): *λp*. *p*
  Λ *P Q p* (*h*: -). *mp* · - · - • (*conjunct2* · - · - • *h*)

iffD2 (*Q*): *Null*
  Λ *P Q q1* (*h*: -) *q2*.

$mp \cdot - \cdot - \cdot \bullet (spec \cdot - \cdot q2 \bullet (conjunct2 \cdot - \cdot - \cdot \bullet h))$

iffD2: *Null iffD2*

iffI (P, Q): *Pair*
$\Lambda\ P\ Q\ pq\ (h1 : -)\ qp\ (h2 : -).\ conjI\text{-}realizer \cdot$
$(\lambda pq.\ \forall x.\ P\ x \longrightarrow Q\ (pq\ x)) \cdot pq \cdot$
$(\lambda qp.\ \forall x.\ Q\ x \longrightarrow P\ (qp\ x)) \cdot qp \bullet$
$(allI \cdot - \bullet (\Lambda\ x.\ impI \cdot - \cdot - \cdot \bullet (h1 \cdot x))) \bullet$
$(allI \cdot - \bullet (\Lambda\ x.\ impI \cdot - \cdot - \cdot \bullet (h2 \cdot x)))$

iffI (P): $\lambda p.\ p$
$\Lambda\ P\ Q\ (h1 : -)\ p\ (h2 : -).\ conjI \cdot - \cdot - \cdot \bullet$
$(allI \cdot - \bullet (\Lambda\ x.\ impI \cdot - \cdot - \cdot \bullet (h1 \cdot x))) \bullet$
$(impI \cdot - \cdot - \cdot \bullet h2)$

iffI (Q): $\lambda q.\ q$
$\Lambda\ P\ Q\ q\ (h1 : -)\ (h2 : -).\ conjI \cdot - \cdot - \cdot \bullet$
$(impI \cdot - \cdot - \cdot \bullet h1) \bullet$
$(allI \cdot - \bullet (\Lambda\ x.\ impI \cdot - \cdot - \cdot \bullet (h2 \cdot x)))$

iffI: *Null iffI*

end

37 Relation-Power: Powers of Relations and Functions

theory *Relation-Power*
imports *Power Transitive-Closure*
begin

instance
fun :: (*type*, *type*) *power* **..**
— only type $'a \Rightarrow 'a$ should be in class *power*!

overloading
relpow $\equiv$ *power* :: $('a \times 'a)\ set \Rightarrow nat \Rightarrow ('a \times 'a)\ set$ (**unchecked**)
begin

$R\ \hat{}\ n = R\ O\ ...\ O\ R$, the n-fold composition of R

primrec *relpow* **where**
$(R :: ('a \times 'a)\ set)\ \hat{}\ 0 = Id$
$|\ (R :: ('a \times 'a)\ set)\ \hat{}\ Suc\ n = R\ O\ (R\ \hat{}\ n)$

end

overloading
funpow ≡ *power* :: $('a \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a \Rightarrow 'a$ (**unchecked**)
begin

$f \hat{} n = f\ o\ ...\ o\ f$, the n-fold composition of f

primrec *funpow* **where**
$(f :: 'a \Rightarrow 'a) \hat{} 0 = id$
| $(f :: 'a \Rightarrow 'a) \hat{} Suc\ n = f\ o\ (f \hat{} n)$

end

WARNING: due to the limits of Isabelle's type classes, exponentiation on functions and relations has too general a domain, namely $('a \times 'b)\ set$ and $'a \Rightarrow 'b$. Explicit type constraints may therefore be necessary. For example, $range\ (f \hat{} n) = A$ and $Range\ (R \hat{} n) = B$ need constraints.

Circumvent this problem for code generation:

primrec
fun-pow :: $nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$
where
fun-pow 0 f = *id*
| *fun-pow* (*Suc n*) *f* = *f o fun-pow n f*

lemma *funpow-fun-pow* [*code inline*]: $f \hat{} n = fun\text{-}pow\ n\ f$
unfolding *funpow-def fun-pow-def* **..**

lemma *funpow-add*: $f \hat{} (m+n) = f\hat{}m\ o\ f\hat{}n$
by (*induct m*) *simp-all*

lemma *funpow-swap1*: $f((f\hat{}n)\ x) = (f\hat{}n)(f\ x)$
proof −
have $f((f\hat{}n)\ x) = (f\hat{}(n+1))\ x$ **by** *simp*
also have ... $= (f\hat{}n\ o\ f\hat{}1)\ x$ **by** (*simp only*: *funpow-add*)
also have ... $= (f\hat{}n)(f\ x)$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *rel-pow-1* [*simp*]:
fixes $R :: ('a*'a)set$
shows $R\hat{}1 = R$
by *simp*

lemma *rel-pow-0-I*: $(x,x) : R\hat{}0$
by *simp*

lemma *rel-pow-Suc-I*: [| $(x,y) : R\hat{}n$; $(y,z):R$ |] ==> $(x,z):R\hat{}(Suc\ n)$
by *auto*

lemma *rel-pow-Suc-I2*:
  $(x,\ y) : R \Longrightarrow (y,\ z) : R\hat{}n \Longrightarrow (x,z) : R\hat{}(Suc\ n)$
 apply (*induct n arbitrary*: *z*)
  apply *simp*
 apply *fastsimp*
 done

lemma *rel-pow-0-E*: [| $(x,y) : R\hat{}0$; $x=y$ ==> P |] ==> P
 by *simp*

lemma *rel-pow-Suc-E*:
  [| $(x,z) : R\hat{}(Suc\ n)$; !!y. [| $(x,y) : R\hat{}n$; $(y,z) : R$ |] ==> P |] ==> P
 by *auto*

lemma *rel-pow-E*:
  [| $(x,z) : R\hat{}n$; [| $n=0$; $x = z$ |] ==> P;
      !!$y\ m$. [| $n = Suc\ m$; $(x,y) : R\hat{}m$; $(y,z) : R$ |] ==> P
   |] ==> P
 by (*cases n*) *auto*

lemma *rel-pow-Suc-D2*:
  $(x,\ z) : R\hat{}(Suc\ n) \Longrightarrow (\exists\, y.\ (x,y) : R\ \&\ (y,z) : R\hat{}n)$
 apply (*induct n arbitrary*: *x z*)
  apply (*blast intro*: *rel-pow-0-I elim*: *rel-pow-0-E rel-pow-Suc-E*)
 apply (*blast intro*: *rel-pow-Suc-I elim*: *rel-pow-0-E rel-pow-Suc-E*)
 done

lemma *rel-pow-Suc-D2'*:
  $\forall x\ y\ z.\ (x,y) : R\hat{}n\ \&\ (y,z) : R$ --> $(\exists\, w.\ (x,w) : R\ \&\ (w,z) : R\hat{}n)$
 by (*induct n*) (*simp-all*, *blast*)

lemma *rel-pow-E2*:
  [| $(x,z) : R\hat{}n$; [| $n=0$; $x = z$ |] ==> P;
      !!$y\ m$. [| $n = Suc\ m$; $(x,y) : R$; $(y,z) : R\hat{}m$ |] ==> P
   |] ==> P
 apply (*case-tac n*, *simp*)
 apply (*cut-tac n=nat* **and** *R=R* **in** *rel-pow-Suc-D2'*, *simp*, *blast*)
 done

lemma *rtrancl-imp-UN-rel-pow*: !!p. p:$R\hat{}*$ ==> p : $(UN\ n.\ R\hat{}n)$
 apply (*simp only*: *split-tupled-all*)
 apply (*erule rtrancl-induct*)
  apply (*blast intro*: *rel-pow-0-I rel-pow-Suc-I*)+
 done

lemma *rel-pow-imp-rtrancl*: !!p. p:$R\hat{}n$ ==> p:$R\hat{}*$
 apply (*simp only*: *split-tupled-all*)
 apply (*induct n*)

apply (*blast intro*: *rtrancl-refl elim*: *rel-pow-0-E*)
apply (*blast elim*: *rel-pow-Suc-E intro*: *rtrancl-into-rtrancl*)
done

lemma *rtrancl-is-UN-rel-pow*: $R\hat{}* = (UN\ n.\ R\hat{}n)$
by (*blast intro*: *rtrancl-imp-UN-rel-pow rel-pow-imp-rtrancl*)

lemma *trancl-power*:
$x \in r\hat{}+ = (\exists n > 0.\ x \in r\hat{}n)$
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*:
$!!r::('a * 'a)set.\ single\text{-}valued\ r ==> single\text{-}valued\ (r\hat{}n)$
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

38 Hilbert-Choice: Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*
imports *Nat Wellfounded*
uses (*Tools/meson.ML*) (*Tools/specification-package.ML*)
begin

38.1 Hilbert's epsilon

axiomatization
  Eps :: (*'a* => *bool*) => *'a*
where
  someI: *P x* ==> *P* (*Eps P*)

syntax (*epsilon*)
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3ϵ -./ -*) [*0*, *10*] *10*)
syntax (*HOL*)
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3@ -./ -*) [*0*, *10*] *10*)
syntax
  -Eps :: [*pttrn*, *bool*] => *'a* ((*3SOME -./ -*) [*0*, *10*] *10*)
translations
  SOME x. P == *CONST Eps* (%*x. P*)

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

constdefs
  inv :: (*'a* => *'b*) => (*'b* => *'a*)
  inv(*f* :: *'a* => *'b*) == %*y. SOME x. f x* = *y*

  Inv :: *'a set* => (*'a* => *'b*) => (*'b* => *'a*)
  Inv A f == %*x. SOME y. y* ∈ *A* & *f y* = *x*

38.2 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

38.3 Axiom of Choice, Proved Using the Description Operator

Used in *Tools/meson.ML*

lemma *choice*: ∀ *x*. ∃ *y*. *Q x y* ==> ∃ *f*. ∀ *x*. *Q x* (*f x*)
by (*fast elim*: *someI*)

lemma *bchoice*: ∀ *x*∈*S*. ∃ *y*. *Q x y* ==> ∃ *f*. ∀ *x*∈*S*. *Q x* (*f x*)
by (*fast elim*: *someI*)

38.4 Function Inverse

lemma *inv-id* [*simp*]: *inv id* = *id*
by (*simp add*: *inv-def id-def*)

A one-to-one function has an inverse.

lemma *inv-f-f* [*simp*]: *inj f* ==> *inv f* (*f x*) = *x*
by (*simp add*: *inv-def inj-eq*)

lemma *inv-f-eq*: [| *inj f*; *f x* = *y* |] ==> *inv f y* = *x*
apply (*erule subst*)
apply (*erule inv-f-f*)
done

lemma *inj-imp-inv-eq*: [| *inj f*; ∀ *x*. *f*(*g x*) = *x* |] ==> *inv f* = *g*
by (*blast intro*: *ext inv-f-eq*)

But is it useful?

lemma *inj-transfer*:
assumes *injf*: *inj f* **and** *minor*: !!*y*. *y* ∈ *range*(*f*) ==> *P*(*inv f y*)
shows *P x*
proof −
have *f x* ∈ *range f* **by** *auto*
hence *P*(*inv f* (*f x*)) **by** (*rule minor*)
thus *P x* **by** (*simp add*: *inv-f-f* [*OF injf*])
qed

lemma *inj-iff*: (*inj f*) = (*inv f o f* = *id*)
apply (*simp add*: *o-def expand-fun-eq*)
apply (*blast intro*: *inj-on-inverseI inv-f-f*)
done

lemma *inv-o-cancel*[*simp*]: *inj f* ==> *inv f o f* = *id*
by (*simp add*: *inj-iff*)

lemma *o-inv-o-cancel*[*simp*]: *inj f* ==> *g o inv f o f* = *g*
by (*simp add*: *o-assoc*[*symmetric*])

lemma *inv-image-cancel*[*simp*]:
inj f ==> *inv f* ‘ *f* ‘ *S* = *S*
by (*simp add*: *image-compose*[*symmetric*])

lemma *inj-imp-surj-inv*: *inj f* ==> *surj* (*inv f*)
by (*blast intro*: *surjI inv-f-f*)

lemma *f-inv-f*: *y* ∈ *range*(*f*) ==> *f*(*inv f y*) = *y*
apply (*simp add*: *inv-def*)
apply (*fast intro*: *someI*)
done

lemma *surj-f-inv-f*: *surj f* ==> *f*(*inv f y*) = *y*
by (*simp add*: *f-inv-f surj-range*)

lemma *inv-injective*:

assumes *eq*: *inv f x = inv f y*
and *x*: *x*: *range f*
and *y*: *y*: *range f*
shows *x=y*
proof −
have *f (inv f x) = f (inv f y)* **using** *eq* **by** *simp*
thus *?thesis* **by** (*simp add*: *f-inv-f x y*)
qed

lemma *inj-on-inv*: *A <= range(f) ==> inj-on (inv f) A*
by (*fast intro*: *inj-onI elim*: *inv-injective injD*)

lemma *surj-imp-inj-inv*: *surj f ==> inj (inv f)*
by (*simp add*: *inj-on-inv surj-range*)

lemma *surj-iff*: *(surj f) = (f o inv f = id)*
apply (*simp add*: *o-def expand-fun-eq*)
apply (*blast intro*: *surjI surj-f-inv-f*)
done

lemma *surj-imp-inv-eq*: [| *surj f*; ∀ *x*. *g(f x) = x* |] ==> *inv f = g*
apply (*rule ext*)
apply (*drule-tac x = inv f x* **in** *spec*)
apply (*simp add*: *surj-f-inv-f*)
done

lemma *bij-imp-bij-inv*: *bij f ==> bij (inv f)*
by (*simp add*: *bij-def inj-imp-surj-inv surj-imp-inj-inv*)

lemma *inv-equality*: [| !!*x*. *g (f x) = x*; !!*y*. *f (g y) = y* |] ==> *inv f = g*
apply (*rule ext*)
apply (*auto simp add*: *inv-def*)
done

lemma *inv-inv-eq*: *bij f ==> inv (inv f) = f*
apply (*rule inv-equality*)
apply (*auto simp add*: *bij-def surj-f-inv-f*)
done

lemma *o-inv-distrib*: [| *bij f*; *bij g* |] ==> *inv (f o g) = inv g o inv f*
apply (*rule inv-equality*)
apply (*auto simp add*: *bij-def surj-f-inv-f*)
done

lemma *image-surj-f-inv-f*: *surj f ==> f ‘ (inv f ‘ A) = A*
by (*simp add*: *image-eq-UN surj-f-inv-f*)

lemma *image-inv-f-f*: *inj f ==> (inv f) ' (f ' A) = A*
by (*simp add*: *image-eq-UN*)

lemma *inv-image-comp*: *inj f ==> inv f ' (f'X) = X*
by (*auto simp add*: *image-def*)

lemma *bij-image-Collect-eq*: *bij f ==> f ' Collect P = {y. P (inv f y)}*
apply *auto*
apply (*force simp add*: *bij-is-inj*)
apply (*blast intro*: *bij-is-surj* [*THEN surj-f-inv-f*, *symmetric*])
done

lemma *bij-vimage-eq-inv-image*: *bij f ==> f -' A = inv f ' A*
apply (*auto simp add*: *bij-is-surj* [*THEN surj-f-inv-f*])
apply (*blast intro*: *bij-is-inj* [*THEN inv-f-f*, *symmetric*])
done

38.5 Inverse of a PI-function (restricted domain)

lemma *Inv-f-f*: [| *inj-on f A*; *x ∈ A* |] ==> *Inv A f (f x) = x*
apply (*simp add*: *Inv-def inj-on-def*)
apply (*blast intro*: *someI2*)
done

lemma *f-Inv-f*: *y ∈ f'A ==> f (Inv A f y) = y*
apply (*simp add*: *Inv-def*)
apply (*fast intro*: *someI2*)
done

lemma *Inv-injective*:
assumes *eq*: *Inv A f x = Inv A f y*
and *x*: *x*: *f'A*
and *y*: *y*: *f'A*
shows *x=y*
proof −
have *f (Inv A f x) = f (Inv A f y)* **using** *eq* **by** *simp*
thus *?thesis* **by** (*simp add*: *f-Inv-f x y*)
qed

lemma *inj-on-Inv*: *B <= f'A ==> inj-on (Inv A f) B*
apply (*rule inj-onI*)
apply (*blast intro*: *inj-onI dest*: *Inv-injective injD*)
done

lemma *Inv-mem*: [| *f ' A = B*; *x ∈ B* |] ==> *Inv A f x ∈ A*
apply (*simp add*: *Inv-def*)
apply (*fast intro*: *someI2*)
done

lemma *Inv-f-eq*: [| *inj-on f A*; *f x* = *y*; *x* ∈ *A* |] ==> *Inv A f y* = *x*
apply (*erule subst*)
apply (*erule Inv-f-f*, *assumption*)
done

lemma *Inv-comp*:
[| *inj-on f* (*g* ' *A*); *inj-on g A*; *x* ∈ *f* ' *g* ' *A* |] ==>
Inv A (*f o g*) *x* = (*Inv A g o Inv* (*g* ' *A*) *f*) *x*
apply *simp*
apply (*rule Inv-f-eq*)
apply (*fast intro*: *comp-inj-on*)
apply (*simp add*: *f-Inv-f Inv-mem*)
apply (*simp add*: *Inv-mem*)
done

lemma *bij-betw-Inv*: *bij-betw f A B* ⟹ *bij-betw* (*Inv A f*) *B A*
apply (*auto simp add*: *bij-betw-def inj-on-Inv Inv-mem*)
apply (*simp add*: *image-compose* [*symmetric*] *o-def*)
apply (*simp add*: *image-def Inv-f-f*)
done

38.6 Other Consequences of Hilbert's Epsilon

Hilbert's Epsilon and the *split* Operator

Looping simprule

lemma *split-paired-Eps*: (*SOME x*. *P x*) = (*SOME* (*a*,*b*). *P*(*a*,*b*))
by *simp*

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

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

A relation is wellfounded iff it has no infinite descending chain

lemma *wf-iff-no-infinite-down-chain*:
wf r = (~(∃*f*. ∀*i*. (*f*(*Suc i*),*f i*) ∈ *r*))
apply (*simp only*: *wf-eq-minimal*)
apply (*rule iffI*)
apply (*rule notI*)
apply (*erule exE*)
apply (*erule-tac x* = {*w*. ∃*i*. *w*=*f i*} **in** *allE*, *blast*)
apply (*erule contrapos-np*, *simp*, *clarify*)
apply (*subgoal-tac* ∀*n*. *nat-rec x* (%*i y*. @*z*. *z*:*Q* & (*z*,*y*) :*r*) *n* ∈ *Q*)
apply (*rule-tac x* = *nat-rec x* (%*i y*. @*z*. *z*:*Q* & (*z*,*y*) :*r*) **in** *exI*)
apply (*rule allI*, *simp*)
apply (*rule someI2-ex*, *blast*, *blast*)

apply (*rule allI*)
apply (*induct-tac n*, *simp-all*)
apply (*rule someI2-ex*, *blast+*)
done

A dynamically-scoped fact for TFL

lemma *tfl-some*: $\forall P\ x.\ P\ x\ \longrightarrow\ P\ (Eps\ P)$
by (*blast intro*: *someI*)

38.7 Least value operator

constdefs
LeastM :: [*'a* => *'b*::*ord*, *'a* => *bool*] => *'a*
LeastM m P == *SOME x*. *P x* & (∀ *y*. *P y* −−> *m x* <= *m y*)

syntax
-*LeastM* :: [*pttrn*, *'a* => *'b*::*ord*, *bool*] => *'a* (*LEAST - WRT -*. - [*0*, *4*, *10*] *10*)
translations
LEAST x WRT m. *P* == *LeastM m* (%*x*. *P*)

lemma *LeastMI2*:
P x ==> (!!*y*. *P y* ==> *m x* <= *m y*)
==> (!!*x*. *P x* ==> ∀ *y*. *P y* −−> *m x* ≤ *m y* ==> *Q x*)
==> *Q* (*LeastM m P*)
apply (*simp add*: *LeastM-def*)
apply (*rule someI2-ex*, *blast*, *blast*)
done

lemma *LeastM-equality*:
P k ==> (!!*x*. *P x* ==> *m k* <= *m x*)
==> *m* (*LEAST x WRT m*. *P x*) = (*m k*::*'a*::*order*)
apply (*rule LeastMI2*, *assumption*, *blast*)
apply (*blast intro*!: *order-antisym*)
done

lemma *wf-linord-ex-has-least*:
wf r ==> ∀ *x y*. ((*x*,*y*):*r*^+) = ((*y*,*x*)~:*r*^*) ==> *P k*
==> ∃ *x*. *P x* & (!*y*. *P y* −−> (*m x*,*m y*):*r*^*)
apply (*drule wf-trancl* [*THEN wf-eq-minimal* [*THEN iffD1*]])
apply (*drule-tac x* = *m*'*Collect P* **in** *spec*, *force*)
done

lemma *ex-has-least-nat*:
P k ==> ∃ *x*. *P x* & (∀ *y*. *P y* −−> *m x* <= (*m y*::*nat*))
apply (*simp only*: *pred-nat-trancl-eq-le* [*symmetric*])
apply (*rule wf-pred-nat* [*THEN wf-linord-ex-has-least*])
apply (*simp add*: *less-eq linorder-not-le pred-nat-trancl-eq-le*, *assumption*)
done

lemma *LeastM-nat-lemma*:
P k ==> P (LeastM m P) & (∀ y. P y --> m (LeastM m P) <= (m y::nat))
apply (*simp add*: *LeastM-def*)
apply (*rule someI-ex*)
apply (*erule ex-has-least-nat*)
done

lemmas *LeastM-natI = LeastM-nat-lemma* [*THEN conjunct1*, *standard*]

lemma *LeastM-nat-le*: *P x ==> m (LeastM m P) <= (m x::nat)*
by (*rule LeastM-nat-lemma* [*THEN conjunct2*, *THEN spec*, *THEN mp*], *assumption*, *assumption*)

38.8 Greatest value operator

constdefs
GreatestM :: ['*a* => '*b*::*ord*, '*a* => *bool*] => '*a*
GreatestM m P == SOME x. P x & (∀ y. P y --> m y <= m x)

Greatest :: ('*a*::*ord* => *bool*) => '*a* (**binder** *GREATEST* *10*)
Greatest == GreatestM (%x. x)

syntax
-GreatestM :: [*pttrn*, '*a*=>'*b*::*ord*, *bool*] => '*a*
(*GREATEST - WRT -. - [0, 4, 10] 10*)

translations
GREATEST x WRT m. P == GreatestM m (%x. P)

lemma *GreatestMI2*:
P x ==> (!!y. P y ==> m y <= m x)
==> (!!x. P x ==> ∀ y. P y --> m y ≤ m x ==> Q x)
==> Q (GreatestM m P)
apply (*simp add*: *GreatestM-def*)
apply (*rule someI2-ex*, *blast*, *blast*)
done

lemma *GreatestM-equality*:
P k ==> (!!x. P x ==> m x <= m k)
==> m (GREATEST x WRT m. P x) = (m k::'a::order)
apply (*rule-tac m = m* **in** *GreatestMI2*, *assumption*, *blast*)
apply (*blast intro*!: *order-antisym*)
done

lemma *Greatest-equality*:
P (k::'a::order) ==> (!!x. P x ==> x <= k) ==> (GREATEST x. P x) = k
apply (*simp add*: *Greatest-def*)
apply (*erule GreatestM-equality*, *blast*)

done

lemma *ex-has-greatest-nat-lemma*:
P k ==> ∀ x. P x --> (∃ y. P y & ~ ((m y::nat) <= m x))
==> ∃ y. P y & ~ (m y < m k + n)
apply (*induct n*, *force*)
apply (*force simp add*: *le-Suc-eq*)
done

lemma *ex-has-greatest-nat*:
P k ==> ∀ y. P y --> m y < b
==> ∃ x. P x & (∀ y. P y --> (m y::nat) <= m x)
apply (*rule ccontr*)
apply (*cut-tac P = P* **and** *n = b - m k* **in** *ex-has-greatest-nat-lemma*)
apply (*subgoal-tac* [*3*] *m k <= b*, *auto*)
done

lemma *GreatestM-nat-lemma*:
P k ==> ∀ y. P y --> m y < b
==> P (GreatestM m P) & (∀ y. P y --> (m y::nat) <= m (GreatestM m P))
apply (*simp add*: *GreatestM-def*)
apply (*rule someI-ex*)
apply (*erule ex-has-greatest-nat*, *assumption*)
done

lemmas *GreatestM-natI = GreatestM-nat-lemma* [*THEN conjunct1*, *standard*]

lemma *GreatestM-nat-le*:
P x ==> ∀ y. P y --> m y < b
==> (m x::nat) <= m (GreatestM m P)
apply (*blast dest*: *GreatestM-nat-lemma* [*THEN conjunct2*, *THEN spec*, *of P*])
done

Specialization to *GREATEST*.

lemma *GreatestI*: *P (k::nat) ==> ∀ y. P y --> y < b ==> P (GREATEST x. P x)*
apply (*simp add*: *Greatest-def*)
apply (*rule GreatestM-natI*, *auto*)
done

lemma *Greatest-le*:
P x ==> ∀ y. P y --> y < b ==> (x::nat) <= (GREATEST x. P x)
apply (*simp add*: *Greatest-def*)
apply (*rule GreatestM-nat-le*, *auto*)
done

38.9 The Meson proof procedure

38.9.1 Negation Normal Form

de Morgan laws

lemma *meson-not-conjD*: $\sim(P\&Q) ==> \sim P \mid \sim Q$
and *meson-not-disjD*: $\sim(P|Q) ==> \sim P \ \& \ \sim Q$
and *meson-not-notD*: $\sim\sim P ==> P$
and *meson-not-allD*: $!!P.\ \sim(\forall x.\ P(x)) ==> \exists x.\ \sim P(x)$
and *meson-not-exD*: $!!P.\ \sim(\exists x.\ P(x)) ==> \forall x.\ \sim P(x)$
by *fast+*

Removal of $--->$ and $<->$ (positive and negative occurrences)

lemma *meson-imp-to-disjD*: $P--->Q ==> \sim P \mid Q$
and *meson-not-impD*: $\sim(P--->Q) ==> P \ \& \ \sim Q$
and *meson-iff-to-disjD*: $P=Q ==> (\sim P \mid Q) \ \& \ (\sim Q \mid P)$
and *meson-not-iffD*: $\sim(P=Q) ==> (P \mid Q) \ \& \ (\sim P \mid \sim Q)$
— Much more efficient than $P \wedge \neg\ Q \vee Q \wedge \neg\ P$ for computing CNF
and *meson-not-refl-disj-D*: $x \sim= x \mid P ==> P$
by *fast+*

38.9.2 Pulling out the existential quantifiers

Conjunction

lemma *meson-conj-exD1*: $!!P\ Q.\ (\exists x.\ P(x)) \ \& \ Q ==> \exists x.\ P(x) \ \& \ Q$
and *meson-conj-exD2*: $!!P\ Q.\ P \ \& \ (\exists x.\ Q(x)) ==> \exists x.\ P \ \& \ Q(x)$
by *fast+*

Disjunction

lemma *meson-disj-exD*: $!!P\ Q.\ (\exists x.\ P(x)) \mid (\exists x.\ Q(x)) ==> \exists x.\ P(x) \mid Q(x)$
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants
and *meson-disj-exD1*: $!!P\ Q.\ (\exists x.\ P(x)) \mid Q ==> \exists x.\ P(x) \mid Q$
and *meson-disj-exD2*: $!!P\ Q.\ P \mid (\exists x.\ Q(x)) ==> \exists x.\ P \mid Q(x)$
by *fast+*

38.9.3 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+*

38.10 Lemmas for Meson, the Model Elimination Procedure

Generation of contrapositives

Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

lemma *make-neg-rule*: $^{\sim}P|Q \Longrightarrow ((^{\sim}P \Longrightarrow P) \Longrightarrow Q)$
by *blast*

Version for Plaisted's "Postive refinement" of the Meson procedure

lemma *make-refined-neg-rule*: $^{\sim}P|Q \Longrightarrow (P \Longrightarrow Q)$
by *blast*

P should be a literal

lemma *make-pos-rule*: $P|Q \Longrightarrow ((P \Longrightarrow ^{\sim}P) \Longrightarrow Q)$
by *blast*

Versions of *make-neg-rule* and *make-pos-rule* that don't insert new assumptions, for ordinary resolution.

lemmas *make-neg-rule'* = *make-refined-neg-rule*

lemma *make-pos-rule'*: $[|P|Q; {}^{\sim}P|] \Longrightarrow Q$
by *blast*

Generation of a goal clause – put away the final literal

lemma *make-neg-goal*: $^{\sim}P \Longrightarrow ((^{\sim}P \Longrightarrow P) \Longrightarrow False)$
by *blast*

lemma *make-pos-goal*: $P \Longrightarrow ((P \Longrightarrow ^{\sim}P) \Longrightarrow False)$
by *blast*

38.10.1 Lemmas for Forward Proof

There is a similarity to congruence rules

lemma *conj-forward*: $[|\ P'\&Q';\ \ P' \Longrightarrow P;\ \ Q' \Longrightarrow Q\ |] \Longrightarrow P\&Q$
by *blast*

lemma *disj-forward*: $[|\ P'|Q';\ \ P' \Longrightarrow P;\ \ Q' \Longrightarrow Q\ |] \Longrightarrow P|Q$
by *blast*

lemma *disj-forward2*:
$[|\ P'|Q';\ \ P' \Longrightarrow P;\ \ [|\ Q';\ P \Longrightarrow False\ |] \Longrightarrow Q\ |] \Longrightarrow P|Q$
apply *blast*
done

lemma *all-forward*: $[|\ \forall x.\ P'(x);\ \ !!x.\ P'(x) \Longrightarrow P(x)\ |] \Longrightarrow \forall x.\ P(x)$
by *blast*

lemma *ex-forward*: $[|\ \exists x.\ P'(x);\ \ !!x.\ P'(x) \Longrightarrow P(x)\ |] \Longrightarrow \exists x.\ P(x)$

by *blast*

Many of these bindings are used by the ATP linkup, and not just by legacy proof scripts.

ML

```
⟪
val inv-def = thm inv-def;
val Inv-def = thm Inv-def;

val someI = thm someI;
val someI-ex = thm someI-ex;
val someI2 = thm someI2;
val someI2-ex = thm someI2-ex;
val some-equality = thm some-equality;
val some1-equality = thm some1-equality;
val some-eq-ex = thm some-eq-ex;
val some-eq-trivial = thm some-eq-trivial;
val some-sym-eq-trivial = thm some-sym-eq-trivial;
val choice = thm choice;
val bchoice = thm bchoice;
val inv-id = thm inv-id;
val inv-f-f = thm inv-f-f;
val inv-f-eq = thm inv-f-eq;
val inj-imp-inv-eq = thm inj-imp-inv-eq;
val inj-transfer = thm inj-transfer;
val inj-iff = thm inj-iff;
val inj-imp-surj-inv = thm inj-imp-surj-inv;
val f-inv-f = thm f-inv-f;
val surj-f-inv-f = thm surj-f-inv-f;
val inv-injective = thm inv-injective;
val inj-on-inv = thm inj-on-inv;
val surj-imp-inj-inv = thm surj-imp-inj-inv;
val surj-iff = thm surj-iff;
val surj-imp-inv-eq = thm surj-imp-inv-eq;
val bij-imp-bij-inv = thm bij-imp-bij-inv;
val inv-equality = thm inv-equality;
val inv-inv-eq = thm inv-inv-eq;
val o-inv-distrib = thm o-inv-distrib;
val image-surj-f-inv-f = thm image-surj-f-inv-f;
val image-inv-f-f = thm image-inv-f-f;
val inv-image-comp = thm inv-image-comp;
val bij-image-Collect-eq = thm bij-image-Collect-eq;
val bij-vimage-eq-inv-image = thm bij-vimage-eq-inv-image;
val Inv-f-f = thm Inv-f-f;
val f-Inv-f = thm f-Inv-f;
val Inv-injective = thm Inv-injective;
val inj-on-Inv = thm inj-on-Inv;
val split-paired-Eps = thm split-paired-Eps;
val Eps-split = thm Eps-split;
```

val Eps-split-eq = thm Eps-split-eq;
val wf-iff-no-infinite-down-chain = thm wf-iff-no-infinite-down-chain;
val Inv-mem = thm Inv-mem;
val Inv-f-eq = thm Inv-f-eq;
val Inv-comp = thm Inv-comp;
val tfl-some = thm tfl-some;
val make-neg-rule = thm make-neg-rule;
val make-refined-neg-rule = thm make-refined-neg-rule;
val make-pos-rule = thm make-pos-rule;
val make-neg-rule' = thm make-neg-rule';
val make-pos-rule' = thm make-pos-rule';
val make-neg-goal = thm make-neg-goal;
val make-pos-goal = thm make-pos-goal;
val conj-forward = thm conj-forward;
val disj-forward = thm disj-forward;
val disj-forward2 = thm disj-forward2;
val all-forward = thm all-forward;
val ex-forward = thm ex-forward;
⟩⟩

38.11 Meson package

use *Tools/meson.ML*

setup *Meson.setup*

38.12 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

39 ATP-Linkup: The Isabelle-ATP Linkup

theory *ATP-Linkup*
imports *Record Presburger SAT Recdef Extraction Relation-Power Hilbert-Choice*

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*

39.1 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 *ResReconstruct.setup* — Config parameters
setup *ResAxioms.setup* — Sledgehammer

39.2 The Metis prover

use *Tools/metis-tools.ML*
setup *MetisTools.setup*

setup ⟨⟨
Theory.at-end ResAxioms.clause-cache-endtheory
⟩⟩

end

40 List: The datatype of finite lists

theory *List*
imports *ATP-Linkup*
uses *Tools/string-syntax.ML*
begin

datatype *'a list* =
  Nil ([])
| *Cons 'a 'a list* (**infixr** # *65*)

40.1 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*) = (*if xs*=[] *then x else last xs*)

primrec
butlast []= []
butlast(*x*#*xs*) = (*if xs*=[] *then* [] *else x*#*butlast xs*)

primrec
set [] = {}
set (*x*#*xs*) = *insert x* (*set xs*)

primrec
map f [] = []
map f (*x*#*xs*) = *f*(*x*)#*map f xs*

primrec
append :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixr** @ *65*)
where
append-Nil:[] @ *ys* = *ys*
| *append-Cons*: (*x*#*xs*) @ *ys* = *x* # *xs* @ *ys*

primrec
rev([]) = []
rev(*x*#*xs*) = *rev*(*xs*) @ [*x*]

primrec
filter P [] = []
filter P (*x*#*xs*) = (*if P x then x*#*filter P xs else filter P xs*)

primrec
foldl-Nil:*foldl f a* [] = *a*
foldl-Cons: *foldl f a* (*x*#*xs*) = *foldl f* (*f a x*) *xs*

primrec
foldr f [] *a* = *a*
foldr f (*x*#*xs*) *a* = *f x* (*foldr f xs a*)

primrec
concat([]) = []
concat(*x*#*xs*) = *x* @ *concat*(*xs*)

primrec
listsum [] = *0*
listsum (*x* # *xs*) = *x* + *listsum xs*

primrec
drop-Nil:*drop n* [] = []
drop-Cons: *drop n* (*x*#*xs*) = (*case n of 0* => *x*#*xs* | *Suc*(*m*) => *drop m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
take-Nil:*take n* [] = []
take-Cons: *take n* (*x*#*xs*) = (*case n of 0* => [] | *Suc*(*m*) => *x* # *take m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n* = *0* and *n* = *Suc k*

primrec
nth-Cons:(*x*#*xs*)!*n* = (*case n of 0* => *x* | (*Suc k*) => *xs*!*k*)
— Warning: simpset does not contain this definition, but separate theorems for

$n = 0$ and $n = Suc\ k$

primrec
[][i:=v] = []
(x#xs)[i:=v] = (*case* i *of* 0 => v # xs | Suc j => x # xs[j:=v])

primrec
takeWhile P [] = []
takeWhile P (x#xs) = (*if* P x *then* x#*takeWhile* P xs *else* [])

primrec
dropWhile P [] = []
dropWhile P (x#xs) = (*if* P x *then dropWhile* P xs *else* x#xs)

primrec
zip xs [] = []
zip-Cons: *zip* xs (y#ys) = (*case* xs *of* [] => [] | z#zs => (z,y)#*zip* zs ys)
— Warning: simpset does not contain this definition, but separate theorems for xs = [] and xs = z # zs

primrec
upt-0: [i..<0] = []
upt-Suc: [i..<(Suc j)] = (*if* i <= j *then* [i..<j] @ [j] *else* [])

primrec
distinct [] = *True*
distinct (x#xs) = (x ~: *set* xs ∧ *distinct* xs)

primrec
remdups [] = []
remdups (x#xs) = (*if* x : *set* xs *then remdups* xs *else* x # *remdups* xs)

primrec
remove1 x [] = []
remove1 x (y#xs) = (*if* x=y *then* xs *else* y # *remove1* x xs)

primrec
replicate-0: *replicate* 0 x = []
replicate-Suc: *replicate* (Suc n) x = x # *replicate* n x

definition
rotate1 :: 'a *list* ⇒ 'a *list* **where**
rotate1 xs = (*case* xs *of* [] ⇒ [] | x#xs ⇒ xs @ [x])

definition
rotate :: *nat* ⇒ 'a *list* ⇒ 'a *list* **where**
rotate n = *rotate1* ^ n

definition

list-all2 :: ('*a* => '*b* => *bool*) => '*a list* => '*b list* => *bool* **where**
[*code func del*]: *list-all2 P xs ys* =
(*length xs* = *length ys* ∧ (∀(*x*, *y*) ∈ *set* (*zip xs ys*). *P x y*))

definition
sublist :: '*a list* => *nat set* => '*a list* **where**
sublist xs A = *map fst* (*filter* (λ*p*. *snd p* ∈ *A*) (*zip xs* [*0*..<*size xs*]))

primrec
splice [] *ys* = *ys*
splice (*x*#*xs*) *ys* = (*if ys*=[] *then x*#*xs else x* # *hd ys* # *splice xs* (*tl ys*))
— Warning: simpset does not contain the second eqn but a derived one.

Figure 1 shows charachteristic examples that should give an intuitive understanding of the above functions.

The following simple sort functions are intended for proofs, not for efficient implementations.

context *linorder*
begin

fun *sorted* :: '*a list* ⇒ *bool* **where**
sorted [] ⟷ *True* |
sorted [*x*] ⟷ *True* |
sorted (*x*#*y*#*zs*) ⟷ *x* <= *y* ∧ *sorted* (*y*#*zs*)

primrec *insort* :: '*a* ⇒ '*a list* ⇒ '*a list* **where**
insort x [] = [*x*] |
insort x (*y*#*ys*) = (*if x* <= *y then* (*x*#*y*#*ys*) *else y*#(*insort x ys*))

primrec *sort* :: '*a list* ⇒ '*a list* **where**
sort [] = [] |
sort (*x*#*xs*) = *insort x* (*sort xs*)

end

40.1.1 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: [(*x*,*y*). *x* ← *xs*, *y* ← *ys*, *x* ≠ *y*], the list of all pairs of distinct elements from *xs* and *ys*. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle's set comprehension): [*e*. *x* ← *xs*, ...] rather than `[e| x <- xs, ...]`.

The qualifiers after the dot are

generators *p* ← *xs*, where *p* is a pattern and *xs* an expression of list type, or

guards *b*, where *b* is a boolean expression.

$[a, b] @ [c, d] = [a, b, c, d]$
length $[a, b, c] = 3$
set $[a, b, c] = \{a, b, c\}$
map f $[a, b, c] = [f\ a, f\ b, f\ c]$
rev $[a, b, c] = [c, b, a]$
hd $[a, b, c, d] = a$
tl $[a, b, c, d] = [b, c, d]$
last $[a, b, c, d] = d$
butlast $[a, b, c, d] = [a, b, c]$
filter $(\lambda n{::}nat.\ n{<}2)\ [0,2,1] = [0,1]$
concat $[[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]$
foldl f x $[a, b, c] = f\ (f\ (f\ x\ a)\ b)\ c$
foldr f $[a, b, c]\ x = f\ a\ (f\ b\ (f\ c\ x))$
zip $[a, b, c]\ [x, y, z] = [(a, x), (b, y), (c, z)]$
zip $[a, b]\ [x, y, z] = [(a, x), (b, y)]$
splice $[a, b, c]\ [x, y, z] = [a, x, b, y, c, z]$
splice $[a, b, c, d]\ [x, y] = [a, x, b, y, c, d]$
take 2 $[a, b, c, d] = [a, b]$
take 6 $[a, b, c, d] = [a, b, c, d]$
drop 2 $[a, b, c, d] = [c, d]$
drop 6 $[a, b, c, d] = []$
takeWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [1, 2]$
dropWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [3, 0]$
distinct $[2, 0, 1]$
remdups $[2, 0, 2, 1, 2] = [0, 1, 2]$
remove1 2 $[2, 0, 2, 1, 2] = [0, 2, 1, 2]$
$[a, b, c, d]\ !\ 2 = c$
$[a, b, c, d][2 := x] = [a, b, x, d]$
sublist $[a, b, c, d, e]\ \{0, 2, 3\} = [a, c, d]$
rotate1 $[a, b, c, d] = [b, c, d, a]$
rotate 3 $[a, b, c, d] = [d, a, b, c]$
replicate 4 a $= [a, a, a, a]$
$[2..{<}5] = [2, 3, 4]$
listsum $[1, 2, 3] = 6$

Figure 1: Characteristic examples

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
⟫
```

40.1.2 [] 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*

40.1.3 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: *length* $(xs\ @\ ys) = length\ xs + length\ ys$
by (*induct xs*) *auto*

lemma *length-map* [*simp*]: *length* $(map\ f\ xs) = length\ xs$
by (*induct xs*) *auto*

lemma *length-rev* [*simp*]: *length* $(rev\ xs) = length\ xs$
by (*induct xs*) *auto*

lemma *length-tl* [*simp*]: *length* (*tl xs*) = *length xs* − *1*
by (*cases xs*) *auto*

lemma *length-0-conv* [*iff*]: (*length xs* = *0*) = (*xs* = [])
by (*induct xs*) *auto*

lemma *length-greater-0-conv* [*iff*]: (*0* < *length xs*) = (*xs* ≠ [])
by (*induct xs*) *auto*

lemma *length-pos-if-in-set*: *x* : *set xs* ⟹ *length xs* > *0*
by *auto*

lemma *length-Suc-conv*:
(*length xs* = *Suc n*) = (∃ *y ys*. *xs* = *y* # *ys* ∧ *length ys* = *n*)
by (*induct xs*) *auto*

lemma *Suc-length-conv*:
(*Suc n* = *length xs*) = (∃ *y ys*. *xs* = *y* # *ys* ∧ *length ys* = *n*)
apply (*induct xs*, *simp*, *simp*)
apply *blast*
done

lemma *impossible-Cons*: *length xs* <= *length ys* ==> *xs* = *x* # *ys* = *False*
by (*induct xs*) *auto*

lemma *list-induct2* [*consumes 1*, *case-names Nil Cons*]:
length xs = *length ys* ⟹ *P* [] [] ⟹
(⋀*x xs y ys*. *length xs* = *length ys* ⟹ *P xs ys* ⟹ *P* (*x*#*xs*) (*y*#*ys*))
⟹ *P xs ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs ys*) **then show** *?case* **by** (*cases ys*) *simp-all*
qed

lemma *list-induct3* [*consumes 2*, *case-names Nil Cons*]:
length xs = *length ys* ⟹ *length ys* = *length zs* ⟹ *P* [] [] [] ⟹
(⋀*x xs y ys z zs*. *length xs* = *length ys* ⟹ *length ys* = *length zs* ⟹ *P xs ys zs*
⟹ *P* (*x*#*xs*) (*y*#*ys*) (*z*#*zs*))
⟹ *P xs ys zs*
proof (*induct xs arbitrary*: *ys zs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs ys zs*) **then show** *?case* **by** (*cases ys*, *simp-all*)
(*cases zs*, *simp-all*)
qed

lemma *list-induct2'*:

```
  ⟦ P [] [];
  ⋀x xs. P (x#xs) [];
  ⋀y ys. P [] (y#ys);
   ⋀x xs y ys. P xs ys  ⟹ P (x#xs) (y#ys) ⟧
 ⟹ P xs ys
by (induct xs arbitrary: ys) (case-tac x, auto)+

lemma neq-if-length-neq: length xs ≠ length ys ⟹ (xs = ys) == False
by (rule Eq-FalseI) auto

simproc-setup list-neq ((xs::'a list) = ys) = ⟪
(*
Reduces xs=ys to False if xs and ys cannot be of the same length.
This is the case if the atomic sublists of one are a submultiset
of those of the other list and there are fewer Cons's in one than the other.
*)

let

fun len (Const(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;
⟫
```

40.1.4 @ – 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*]: $(xs\ @\ ys = []) = (xs = [] \wedge ys = [])$
by (*induct xs*) *auto*

lemma *Nil-is-append-conv* [*iff*]: $([] = xs\ @\ ys) = (xs = [] \wedge ys = [])$
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*]:
$length\ xs = length\ ys \vee length\ us = length\ vs$
$\Longrightarrow (xs@us = ys@vs) = (xs{=}ys \wedge us{=}vs)$
apply (*induct xs arbitrary*: *ys*)
apply (*case-tac ys*, *simp*, *force*)
apply (*case-tac ys*, *force*, *simp*)
done

lemma *append-eq-append-conv2*: $(xs\ @\ ys = zs\ @\ ts) =$
$(\exists us.\ xs = zs\ @\ us \wedge us\ @\ ys = ts \mid xs\ @\ us = zs \wedge ys = us@\ ts)$
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*]: $xs \neq [] \Longrightarrow hd\ xs \ \# \ tl\ xs = xs$
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*]: $xs \neq [] \Longrightarrow hd\ (xs \ @ \ ys) = hd\ xs$
by (*simp add*: *hd-append split*: *list.split*)

lemma *tl-append*: $tl\ (xs \ @ \ ys) = (case\ xs\ of\ [] \Rightarrow tl\ ys \mid z\#zs \Rightarrow zs \ @ \ ys)$
by (*simp split*: *list.split*)

lemma *tl-append2* [*simp*]: $xs \neq [] \Longrightarrow tl\ (xs \ @ \ ys) = tl\ xs \ @ \ ys$
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*: $xs = ys \Longrightarrow xs = [] \ @ \ ys$
by *simp*

lemma *Cons-eq-appendI*:
$[|\ x \ \# \ xs1 = ys;\ xs = xs1 \ @ \ zs\ |] \Longrightarrow x \ \# \ xs = ys \ @ \ zs$
by (*drule sym*) *simp*

lemma *append-eq-appendI*:
$[|\ xs \ @ \ xs1 = zs;\ ys = xs1 \ @ \ us\ |] \Longrightarrow xs \ @ \ ys = zs \ @ \ us$
by (*drule sym*) *simp*

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

ML ⟪
local

```
fun last (cons as Const(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];
⟩
```

40.1.5 *map*

lemma *map-ext*: $(!!x.\ x : set\ xs \longrightarrow f\ x = g\ x) \Longrightarrow map\ f\ xs = map\ g\ xs$
by (*induct xs*) *simp-all*

lemma *map-ident* [*simp*]: $map\ (\lambda x.\ x) = (\lambda xs.\ xs)$
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*]:
$xs = ys \Longrightarrow (!!x.\ x : set\ ys \Longrightarrow f\ x = g\ x) \Longrightarrow map\ f\ xs = map\ g\ ys$
— 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*:
$(map\ f\ xs = y\#ys) = (\exists z\ zs.\ xs = z\#zs \wedge f\ z = y \wedge map\ f\ zs = ys)$
by (*cases xs*) *auto*

lemma *Cons-eq-map-conv*:
$(x\#xs = map\ f\ ys) = (\exists z\ zs.\ ys = z\#zs \wedge x = f\ z \wedge xs = map\ f\ zs)$
by (*cases ys*) *auto*

lemmas *map-eq-Cons-D* = *map-eq-Cons-conv* [*THEN iffD1*]
lemmas *Cons-eq-map-D* = *Cons-eq-map-conv* [*THEN iffD1*]
declare *map-eq-Cons-D* [*dest*!] *Cons-eq-map-D* [*dest*!]

lemma *ex-map-conv*:
$(EX\ xs.\ ys = map\ f\ xs) = (ALL\ y : set\ ys.\ EX\ x.\ y = f\ x)$
by(*induct ys*, *auto simp add*: *Cons-eq-map-conv*)

lemma *map-eq-imp-length-eq*:
assumes $map\ f\ xs = map\ f\ ys$
shows $length\ xs = length\ ys$

using *assms* **proof** (*induct ys arbitrary*: *xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons y ys*) **then obtain** *z zs* **where** *xs*: *xs* = *z* # *zs* **by** *auto*
from *Cons xs* **have** *map f zs* = *map f ys* **by** *simp*
moreover with *Cons* **have** *length zs* = *length ys* **by** *blast*
with *xs* **show** *?case* **by** *simp*
qed

lemma *map-inj-on*:
[| *map f xs* = *map f ys*; *inj-on f* (*set xs Un set ys*) |]
==> *xs* = *ys*
apply(*frule map-eq-imp-length-eq*)
apply(*rotate-tac* −*1*)
apply(*induct rule*:*list-induct2*)
apply *simp*
apply(*simp*)
apply (*blast intro*:*sym*)
done

lemma *inj-on-map-eq-map*:
inj-on f (*set xs Un set ys*) ⟹ (*map f xs* = *map f ys*) = (*xs* = *ys*)
by(*blast dest*:*map-inj-on*)

lemma *map-injective*:
map f xs = *map f ys* ==> *inj f* ==> *xs* = *ys*
by (*induct ys arbitrary*: *xs*) (*auto dest*!:*injD*)

lemma *inj-map-eq-map*[*simp*]: *inj f* ⟹ (*map f xs* = *map f ys*) = (*xs* = *ys*)
by(*blast dest*:*map-injective*)

lemma *inj-mapI*: *inj f* ==> *inj* (*map f*)
by (*iprover dest*: *map-injective injD intro*: *inj-onI*)

lemma *inj-mapD*: *inj* (*map f*) ==> *inj f*
apply (*unfold inj-on-def*, *clarify*)
apply (*erule-tac x* = [*x*] **in** *ballE*)
apply (*erule-tac x* = [*y*] **in** *ballE*, *simp*, *blast*)
apply *blast*
done

lemma *inj-map*[*iff*]: *inj* (*map f*) = *inj f*
by (*blast dest*: *inj-mapD intro*: *inj-mapI*)

lemma *inj-on-mapI*: *inj-on f* (⋃(*set* ‘ *A*)) ⟹ *inj-on* (*map f*) *A*
apply(*rule inj-onI*)
apply(*erule map-inj-on*)
apply(*blast intro*:*inj-onI dest*:*inj-onD*)
done

lemma *map-idI*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = x) \Longrightarrow map\ f\ xs = xs$
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*)

40.1.6 *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*

40.1.7 *set*

lemma *finite-set* [*iff*]: *finite* (*set xs*)
by (*induct xs*) *auto*

lemma *set-append* [*simp*]: *set* (*xs @ ys*) = (*set xs ∪ set ys*)
by (*induct xs*) *auto*

lemma *hd-in-set*[*simp*]: *xs ≠ [] ⟹ hd xs : set xs*
by(*cases xs*) *auto*

lemma *set-subset-Cons*: *set xs ⊆ set* (*x # xs*)
by *auto*

lemma *set-ConsD*: *y ∈ set* (*x # xs*) ⟹ *y=x ∨ y ∈ set xs*
by *auto*

lemma *set-empty* [*iff*]: (*set xs* = {}) = (*xs* = [])
by (*induct xs*) *auto*

lemma *set-empty2*[*iff*]: ({} = *set xs*) = (*xs* = [])
by(*induct xs*) *auto*

lemma *set-rev* [*simp*]: *set* (*rev xs*) = *set xs*
by (*induct xs*) *auto*

lemma *set-map* [*simp*]: *set* (*map f xs*) = *f'*(*set xs*)
by (*induct xs*) *auto*

lemma *set-filter* [*simp*]: *set* (*filter P xs*) = {*x. x : set xs ∧ P x*}
by (*induct xs*) *auto*

lemma *set-upt* [*simp*]: *set*[*i..<j*] = {*k. i ≤ k ∧ k < j*}
apply (*induct j*, *simp-all*)
apply (*erule ssubst*, *auto*)
done

lemma *split-list*: $x : set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs$
proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case *Cons* **thus** *?case* **by** (*auto intro*: *Cons-eq-appendI*)
qed

lemma *in-set-conv-decomp*: $x \in set\ xs \longleftrightarrow (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs)$
  by (*auto elim*: *split-list*)

lemma *split-list-first*: $x : set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ ys$
proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*Cons a xs*)
  show *?case*
  proof *cases*
    assume $x = a$ **thus** *?case* **using** *Cons* **by** *fastsimp*
  next
    assume $x \neq a$ **thus** *?case* **using** *Cons* **by**(*fastsimp intro*!: *Cons-eq-appendI*)
  qed
qed

lemma *in-set-conv-decomp-first*:
  $(x : set\ xs) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ ys)$
  by (*auto dest*!: *split-list-first*)

lemma *split-list-last*: $x : set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ zs$
proof (*induct xs rule*:*rev-induct*)
  case *Nil* **thus** *?case* **by** *simp*
next
  case (*snoc a xs*)
  show *?case*
  proof *cases*
    assume $x = a$ **thus** *?case* **using** *snoc* **by** *simp* (*metis ex-in-conv set-empty2*)
  next
    assume $x \neq a$ **thus** *?case* **using** *snoc* **by** *fastsimp*
  qed
qed

lemma *in-set-conv-decomp-last*:
  $(x : set\ xs) = (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ zs)$
  by (*auto dest*!: *split-list-last*)

lemma *split-list-prop*: $\exists\, x \in set\ xs.\ P\ x \Longrightarrow \exists\, ys\ x\ zs.\ xs = ys\ @\ x\ \#\ zs\ \&\ P\ x$
proof (*induct xs*)
  case *Nil* **thus** *?case* **by** *simp*
next

case *Cons* **thus** *?case*
by(*simp add*:*Bex-def*)(*metis append-Cons append.simps*(*1*))
qed

lemma *split-list-propE*:
assumes $\exists\, x \in set\ xs.\ P\ x$
obtains *ys x zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** $P\ x$
using *split-list-prop* [*OF assms*] **by** *blast*

lemma *split-list-first-prop*:
$\exists\, x \in set\ xs.\ P\ x \Longrightarrow$
$\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, y \in set\ ys.\ \neg\ P\ y)$
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof *cases*
assume *P x*
thus *?thesis* **by** *simp*
(*metis Un-upper1 contra-subsetD in-set-conv-decomp-first self-append-conv2 set-append*)
next
assume $\neg\ P\ x$
hence $\exists\, x{\in}set\ xs.\ P\ x$ **using** *Cons*(*2*) **by** *simp*
thus *?thesis* **using** ‹$\neg\ P\ x$› *Cons*(*1*) **by** (*metis append-Cons set-ConsD*)
qed
qed

lemma *split-list-first-propE*:
assumes $\exists\, x \in set\ xs.\ P\ x$
obtains *ys x zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** $P\ x$ **and** $\forall\, y \in set\ ys.\ \neg\ P\ y$
using *split-list-first-prop* [*OF assms*] **by** *blast*

lemma *split-list-first-prop-iff*:
$(\exists\, x \in set\ xs.\ P\ x) \longleftrightarrow$
$(\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, y \in set\ ys.\ \neg\ P\ y))$
by (*rule*, *erule split-list-first-prop*) *auto*

lemma *split-list-last-prop*:
$\exists\, x \in set\ xs.\ P\ x \Longrightarrow$
$\exists\, ys\ x\ zs.\ xs = ys@x\#zs \wedge P\ x \wedge (\forall\, z \in set\ zs.\ \neg\ P\ z)$
proof(*induct xs rule*:*rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc x xs*)
show *?case*
proof *cases*
assume *P x* **thus** *?thesis* **by** (*metis emptyE set-empty*)

next
assume ¬ *P x*
hence ∃ *x*∈*set xs*. *P x* **using** *snoc*(*2*) **by** *simp*
thus *?thesis* **using** ‹¬ *P x*› *snoc*(*1*) **by** *fastsimp*
qed
qed

lemma *split-list-last-propE*:
assumes ∃ *x* ∈ *set xs*. *P x*
obtains *ys x zs* **where** *xs* = *ys* @ *x* # *zs* **and** *P x* **and** ∀ *z* ∈ *set zs*. ¬ *P z*
using *split-list-last-prop* [*OF assms*] **by** *blast*

lemma *split-list-last-prop-iff*:
(∃ *x* ∈ *set xs*. *P x*) ⟷
(∃ *ys x zs*. *xs* = *ys*@*x*#*zs* ∧ *P x* ∧ (∀ *z* ∈ *set zs*. ¬ *P z*))
by (*metis split-list-last-prop* [**where** *P*=*P*] *in-set-conv-decomp*)

lemma *finite-list*: *finite A* ==> *EX xs*. *set xs* = *A*
by (*erule finite-induct*)
(*auto simp add*: *set.simps*(*2*) [*symmetric*] *simp del*: *set.simps*(*2*))

lemma *card-length*: *card* (*set xs*) ≤ *length xs*
by (*induct xs*) (*auto simp add*: *card-insert-if*)

lemma *set-minus-filter-out*:
set xs − {*y*} = *set* (*filter* (λ*x*. ¬ (*x* = *y*)) *xs*)
by (*induct xs*) *auto*

40.1.8 *filter*

lemma *filter-append* [*simp*]: *filter P* (*xs* @ *ys*) = *filter P xs* @ *filter P ys*
by (*induct xs*) *auto*

lemma *rev-filter*: *rev* (*filter P xs*) = *filter P* (*rev xs*)
by (*induct xs*) *simp-all*

lemma *filter-filter* [*simp*]: *filter P* (*filter Q xs*) = *filter* (λ*x*. *Q x* ∧ *P x*) *xs*
by (*induct xs*) *auto*

lemma *length-filter-le* [*simp*]: *length* (*filter P xs*) ≤ *length xs*
by (*induct xs*) (*auto simp add*: *le-SucI*)

lemma *sum-length-filter-compl*:
length(*filter P xs*) + *length*(*filter* (%*x*. ~*P x*) *xs*) = *length xs*
by(*induct xs*) *simp-all*

lemma *filter-True* [*simp*]: ∀ *x* ∈ *set xs*. *P x* ==> *filter P xs* = *xs*
by (*induct xs*) *auto*

lemma *filter-False* [*simp*]: $\forall x \in set\ xs.\ \neg\ P\ x \Longrightarrow filter\ P\ xs = []$
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*]: $set\ (filter\ P\ xs) \leq set\ xs$
by *auto*

lemma *length-filter-less*:
$[\![\ x : set\ xs;\ \sim\ P\ x\ ]\!] \Longrightarrow length(filter\ P\ xs) < length\ xs$
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case*
apply (*auto split*:*split-if-asm*)
using *length-filter-le*[*of P xs*] **apply** *arith*
done
qed

lemma *length-filter-conv-card*:
$length(filter\ p\ xs) = card\{i.\ i < length\ xs\ \&\ p(xs!i)\}$
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
let *?S* $= \{i.\ i < length\ xs\ \&\ p(xs!i)\}$
have *fin*: *finite ?S* **by**(*fast intro*: *bounded-nat-set-is-finite*)
show *?case* (**is** *?l* = *card ?S'*)
proof (*cases*)
assume *p x*
hence *eq*: *?S'* = *insert 0* (*Suc* ' *?S*)
by(*auto simp*: *image-def split*:*nat.split dest*:*gr0-implies-Suc*)
have *length* (*filter p* (*x # xs*)) = *Suc*(*card ?S*)

```
    using Cons ‹p x› by simp
  also have ... = Suc(card(Suc ` ?S)) using fin
    by (simp add: card-image inj-Suc)
  also have ... = card ?S' using eq fin
    by (simp add:card-insert-if) (simp add:image-def)
  finally show ?thesis .
 next
  assume ¬ p x
  hence eq: ?S' = Suc ` ?S
    by(auto simp add: image-def split:nat.split elim:lessE)
  have length (filter p (x # xs)) = card ?S
    using Cons ‹¬ p x› by simp
  also have ... = card(Suc ` ?S) using fin
    by (simp add: card-image inj-Suc)
  also have ... = card ?S' using eq fin
    by (simp add:card-insert-if)
  finally show ?thesis .
 qed
qed

lemma Cons-eq-filterD:
 x#xs = filter P ys ⟹
 ∃us vs. ys = us @ x # vs ∧ (∀u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs
 (is - ⟹ ∃us vs. ?P ys us vs)
proof(induct ys)
 case Nil thus ?case by simp
next
 case (Cons y ys)
 show ?case (is ∃x. ?Q x)
 proof cases
  assume Py: P y
  show ?thesis
  proof cases
   assume x = y
   with Py Cons.prems have ?Q [] by simp
   then show ?thesis ..
  next
   assume x ≠ y
   with Py Cons.prems show ?thesis by simp
  qed
 next
  assume ¬ P y
  with Cons obtain us vs where ?P (y#ys) (y#us) vs by fastsimp
  then have ?Q (y#us) by simp
  then show ?thesis ..
 qed
qed

lemma filter-eq-ConsD:
```

filter P ys = x#xs ⟹
∃ us vs. ys = us @ x # vs ∧ (∀ u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs
by(*rule Cons-eq-filterD*) *simp*

lemma *filter-eq-Cons-iff*:
(filter P ys = x#xs) =
(∃ us vs. ys = us @ x # vs ∧ (∀ u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs)
by(*auto dest*:*filter-eq-ConsD*)

lemma *Cons-eq-filter-iff*:
(x#xs = filter P ys) =
(∃ us vs. ys = us @ x # vs ∧ (∀ u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs)
by(*auto dest*:*Cons-eq-filterD*)

lemma *filter-cong*[*fundef-cong*, *recdef-cong*]:
xs = ys ⟹ (⋀x. x ∈ set ys ⟹ P x = Q x) ⟹ filter P xs = filter Q ys
apply *simp*
apply(*erule thin-rl*)
by (*induct ys*) *simp-all*

40.1.9 List partitioning

primrec *partition* :: *('a ⇒ bool) ⇒'a list ⇒ 'a list × 'a list* **where**
partition P [] = ([], [])
| partition P (x # xs) =
(let (yes, no) = partition P xs
in if P x then (x # yes, no) else (yes, x # no))

lemma *partition-filter1*:
fst (partition P xs) = filter P xs
by (*induct xs*) (*auto simp add*: *Let-def split-def*)

lemma *partition-filter2*:
snd (partition P xs) = filter (Not o P) xs
by (*induct xs*) (*auto simp add*: *Let-def split-def*)

lemma *partition-P*:
assumes *partition P xs = (yes, no)*
shows *(∀ p ∈ set yes. P p) ∧ (∀ p ∈ set no. ¬ P p)*
proof −
from *assms* **have** *yes = fst (partition P xs)* **and** *no = snd (partition P xs)*
by *simp-all*
then show *?thesis* **by** (*simp-all add*: *partition-filter1 partition-filter2*)
qed

lemma *partition-set*:
assumes *partition P xs = (yes, no)*
shows *set yes ∪ set no = set xs*
proof −

from *assms* **have** *yes* = *fst* (*partition P xs*) **and** *no* = *snd* (*partition P xs*)
by *simp-all*
then show *?thesis* **by** (*auto simp add*: *partition-filter1 partition-filter2*)
qed

40.1.10 *concat*

lemma *concat-append* [*simp*]: *concat* (*xs* @ *ys*) = *concat xs* @ *concat ys*
by (*induct xs*) *auto*

lemma *concat-eq-Nil-conv* [*simp*]: (*concat xss* = []) = (∀ *xs* ∈ *set xss*. *xs* = [])
by (*induct xss*) *auto*

lemma *Nil-eq-concat-conv* [*simp*]: ([] = *concat xss*) = (∀ *xs* ∈ *set xss*. *xs* = [])
by (*induct xss*) *auto*

lemma *set-concat* [*simp*]: *set* (*concat xs*) = (*UN x*:*set xs*. *set x*)
by (*induct xs*) *auto*

lemma *concat-map-singleton*[*simp*]: *concat*(*map* (%*x*. [*f x*]) *xs*) = *map f xs*
by (*induct xs*) *auto*

lemma *map-concat*: *map f* (*concat xs*) = *concat* (*map* (*map f*) *xs*)
by (*induct xs*) *auto*

lemma *filter-concat*: *filter p* (*concat xs*) = *concat* (*map* (*filter p*) *xs*)
by (*induct xs*) *auto*

lemma *rev-concat*: *rev* (*concat xs*) = *concat* (*map rev* (*rev xs*))
by (*induct xs*) *auto*

40.1.11 *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* ≠ [] ⟹ *hd xs* = *xs*!*0*
by(*cases xs*) *simp-all*

lemma *list-eq-iff-nth-eq*:
(*xs* = *ys*) = (*length xs* = *length ys* ∧ (*ALL i*<*length xs*. *xs*!*i* = *ys*!*i*))
apply(*induct xs arbitrary*: *ys*)
apply *force*
apply(*case-tac ys*)
apply *simp*
apply(*simp add*:*nth-Cons split*:*nat.split*)**apply** *blast*
done

lemma *set-conv-nth*: *set xs* = {*xs*!*i* | *i*. *i* < *length xs*}
apply (*induct xs*, *simp*, *simp*)
apply *safe*
apply (*metis nat-case-0 nth.simps zero-less-Suc*)
apply (*metis less-Suc-eq-0-disj nth-Cons-Suc*)
apply (*case-tac i*, *simp*)
apply (*metis diff-Suc-Suc nat-case-Suc nth.simps zero-less-diff*)
done

lemma *in-set-conv-nth*: (*x* ∈ *set xs*) = (∃ *i* < *length xs*. *xs*!*i* = *x*)
by(*auto simp*:*set-conv-nth*)

lemma *list-ball-nth*: [| *n* < *length xs*; !*x* : *set xs*. *P x*|] ==> *P*(*xs*!*n*)
by (*auto simp add*: *set-conv-nth*)

lemma *nth-mem* [*simp*]: *n* < *length xs* ==> *xs*!*n* : *set xs*
by (*auto simp add*: *set-conv-nth*)

lemma *all-nth-imp-all-set*:
[| !*i* < *length xs*. *P*(*xs*!*i*); *x* : *set xs*|] ==> *P x*
by (*auto simp add*: *set-conv-nth*)

lemma *all-set-conv-all-nth*:
(∀ *x* ∈ *set xs*. *P x*) = (∀ *i*. *i* < *length xs* −−> *P* (*xs* ! *i*))
by (*auto simp add*: *set-conv-nth*)

lemma *rev-nth*:
n < *size xs* ⟹ *rev xs* ! *n* = *xs* ! (*length xs* − *Suc n*)

proof (*induct xs arbitrary*: *n*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
hence *n*: *n* < *Suc* (*length xs*) **by** *simp*
moreover
{ assume *n* < *length xs*
with *n* **obtain** *n'* **where** *length xs* − *n* = *Suc n'*
by (*cases length xs* − *n*, *auto*)
moreover
then have *length xs* − *Suc n* = *n'* **by** *simp*
ultimately
have *xs* ! (*length xs* − *Suc n*) = (*x* # *xs*) ! (*length xs* − *n*) **by** *simp*
}
ultimately
show *?case* **by** (*clarsimp simp add*: *Cons nth-append*)
qed

40.1.12 *list-update*

lemma *length-list-update* [*simp*]: *length*(*xs*[*i*:=*x*]) = *length xs*
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *nth-list-update*:
i < *length xs*==> (*xs*[*i*:=*x*])!*j* = (*if i* = *j then x else xs*!*j*)
by (*induct xs arbitrary*: *i j*) (*auto simp add*: *nth-Cons split*: *nat.split*)

lemma *nth-list-update-eq* [*simp*]: *i* < *length xs* ==> (*xs*[*i*:=*x*])!*i* = *x*
by (*simp add*: *nth-list-update*)

lemma *nth-list-update-neq* [*simp*]: *i* ≠ *j* ==> *xs*[*i*:=*x*]!*j* = *xs*!*j*
by (*induct xs arbitrary*: *i j*) (*auto simp add*: *nth-Cons split*: *nat.split*)

lemma *list-update-id*[*simp*]: *xs*[*i* := *xs*!*i*] = *xs*
by (*induct xs arbitrary*: *i*) (*simp-all split*:*nat.splits*)

lemma *list-update-beyond*[*simp*]: *length xs* ≤ *i* ⟹ *xs*[*i*:=*x*] = *xs*
apply (*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
done

lemma *list-update-same-conv*:
i < *length xs* ==> (*xs*[*i* := *x*] = *xs*) = (*xs*!*i* = *x*)
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *list-update-append1*:
i < *size xs* ⟹ (*xs* @ *ys*)[*i*:=*x*] = *xs*[*i*:=*x*] @ *ys*

apply (*induct xs arbitrary*: *i*, *simp*)
apply(*simp split*:*nat.split*)
done

lemma *list-update-append*:
(*xs* @ *ys*) [*n*:= *x*] =
(*if n* < *length xs then xs*[*n*:= *x*] @ *ys else xs* @ (*ys* [*n*−*length xs*:= *x*]))
by (*induct xs arbitrary*: *n*) (*auto split*:*nat.splits*)

lemma *list-update-length* [*simp*]:
(*xs* @ *x* # *ys*)[*length xs* := *y*] = (*xs* @ *y* # *ys*)
by (*induct xs*, *auto*)

lemma *update-zip*:
length xs = *length ys* ==>
(*zip xs ys*)[*i*:=*xy*] = *zip* (*xs*[*i*:=*fst xy*]) (*ys*[*i*:=*snd xy*])
by (*induct ys arbitrary*: *i xy xs*) (*auto*, *case-tac xs*, *auto split*: *nat.split*)

lemma *set-update-subset-insert*: *set*(*xs*[*i*:=*x*]) <= *insert x* (*set xs*)
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *set-update-subsetI*: [| *set xs* <= *A*; *x*:*A* |] ==> *set*(*xs*[*i* := *x*]) <= *A*
by (*blast dest*!: *set-update-subset-insert* [*THEN subsetD*])

lemma *set-update-memI*: *n* < *length xs* ⟹ *x* ∈ *set* (*xs*[*n* := *x*])
by (*induct xs arbitrary*: *n*) (*auto split*:*nat.splits*)

lemma *list-update-overwrite*:
xs [*i* := *x*, *i* := *y*] = *xs* [*i* := *y*]
apply (*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
done

lemma *list-update-swap*:
i ≠ *i*′ ⟹ *xs* [*i* := *x*, *i*′ := *x*′] = *xs* [*i*′ := *x*′, *i* := *x*]
apply (*induct xs arbitrary*: *i i*′)
apply *simp*
apply (*case-tac i*, *case-tac i*′)
apply *auto*
apply (*case-tac i*′)
apply *auto*
done

40.1.13 *last* and *butlast*

lemma *last-snoc* [*simp*]: *last* (*xs* @ [*x*]) = *x*
by (*induct xs*) *auto*

lemma *butlast-snoc* [*simp*]: *butlast* (*xs* @ [*x*]) = *xs*
by (*induct xs*) *auto*

lemma *last-ConsL*: *xs* = [] ⟹ *last*(*x*#*xs*) = *x*
by(*simp add*:*last.simps*)

lemma *last-ConsR*: *xs* ≠ [] ⟹ *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* = [] ⟹ *last*(*xs* @ *ys*) = *last xs*
by(*simp add*:*last-append*)

lemma *last-appendR*[*simp*]: *ys* ≠ [] ⟹ *last*(*xs* @ *ys*) = *last ys*
by(*simp add*:*last-append*)

lemma *hd-rev*: *xs* ≠ [] ⟹ *hd*(*rev xs*) = *last xs*
by(*rule rev-exhaust*[*of xs*]) *simp-all*

lemma *last-rev*: *xs* ≠ [] ⟹ *last*(*rev xs*) = *hd xs*
by(*cases xs*) *simp-all*

lemma *last-in-set*[*simp*]: *as* ≠ [] ⟹ *last as* ∈ *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* ⟹ *last* (*drop n xs*) = *last xs*
apply (*induct xs arbitrary*: *n*)
apply *simp*

apply (*auto split*:*nat.split*)
done

lemma *last-conv-nth*: *xs*≠[] ⟹ *last xs* = *xs*!(*length xs* − *1*)
by(*induct xs*)(*auto simp*:*neq-Nil-conv*)

lemma *butlast-conv-take*: *butlast xs* = *take* (*length xs* − *1*) *xs*
by (*induct xs*, *simp*, *case-tac xs*, *simp-all*)

40.1.14 *take* and *drop*

lemma *take-0* [*simp*]: *take 0 xs* = []
by (*induct xs*) *auto*

lemma *drop-0* [*simp*]: *drop 0 xs* = *xs*
by (*induct xs*) *auto*

lemma *take-Suc-Cons* [*simp*]: *take* (*Suc n*) (*x* # *xs*) = *x* # *take n xs*
by *simp*

lemma *drop-Suc-Cons* [*simp*]: *drop* (*Suc n*) (*x* # *xs*) = *drop n xs*
by *simp*

declare *take-Cons* [*simp del*] **and** *drop-Cons* [*simp del*]

lemma *take-Suc*: *xs* ~= [] ==> *take* (*Suc n*) *xs* = *hd xs* # *take n* (*tl xs*)
by(*clarsimp simp add*:*neq-Nil-conv*)

lemma *drop-Suc*: *drop* (*Suc n*) *xs* = *drop n* (*tl xs*)
by(*cases xs*, *simp-all*)

lemma *take-tl*: *take n* (*tl xs*) = *tl* (*take* (*Suc n*) *xs*)
by (*induct xs arbitrary*: *n*) *simp-all*

lemma *drop-tl*: *drop n* (*tl xs*) = *tl*(*drop n xs*)
by(*induct xs arbitrary*: *n*, *simp-all add*:*drop-Cons drop-Suc split*:*nat.split*)

lemma *tl-take*: *tl* (*take n xs*) = *take* (*n* − *1*) (*tl xs*)
by (*cases n*, *simp*, *cases xs*, *auto*)

lemma *tl-drop*: *tl* (*drop n xs*) = *drop n* (*tl xs*)
by (*simp only*: *drop-tl*)

lemma *nth-via-drop*: *drop n xs* = *y*#*ys* ⟹ *xs*!*n* = *y*
apply (*induct xs arbitrary*: *n*, *simp*)
apply(*simp add*:*drop-Cons nth-Cons split*:*nat.splits*)
done

lemma *take-Suc-conv-app-nth*:

i < *length xs* ⟹ *take* (*Suc i*) *xs* = *take i xs* @ [*xs*!*i*]
apply (*induct xs arbitrary*: *i*, *simp*)
apply (*case-tac i*, *auto*)
done

lemma *drop-Suc-conv-tl*:
i < *length xs* ⟹ (*xs*!*i*) # (*drop* (*Suc i*) *xs*) = *drop i xs*
apply (*induct xs arbitrary*: *i*, *simp*)
apply (*case-tac i*, *auto*)
done

lemma *length-take* [*simp*]: *length* (*take n xs*) = *min* (*length xs*) *n*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *length-drop* [*simp*]: *length* (*drop n xs*) = (*length xs* − *n*)
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-all* [*simp*]: *length xs* <= *n* ==> *take n xs* = *xs*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-all* [*simp*]: *length xs* <= *n* ==> *drop n xs* = []
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-append* [*simp*]:
take n (*xs* @ *ys*) = (*take n xs* @ *take* (*n* − *length xs*) *ys*)
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-append* [*simp*]:
drop n (*xs* @ *ys*) = *drop n xs* @ *drop* (*n* − *length xs*) *ys*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-take* [*simp*]: *take n* (*take m xs*) = *take* (*min n m*) *xs*
apply (*induct m arbitrary*: *xs n*, *auto*)
apply (*case-tac xs*, *auto*)
apply (*case-tac n*, *auto*)
done

lemma *drop-drop* [*simp*]: *drop n* (*drop m xs*) = *drop* (*n* + *m*) *xs*
apply (*induct m arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *take-drop*: *take n* (*drop m xs*) = *drop m* (*take* (*n* + *m*) *xs*)
apply (*induct m arbitrary*: *xs n*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *drop-take*: *drop n* (*take m xs*) = *take* (*m*−*n*) (*drop n xs*)
apply(*induct xs arbitrary*: *m n*)

apply *simp*
apply(*simp add*: *take-Cons drop-Cons split*:*nat.split*)
done

lemma *append-take-drop-id* [*simp*]: *take n xs* @ *drop n xs* = *xs*
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *take-eq-Nil*[*simp*]: (*take n xs* = []) = (*n* = *0* ∨ *xs* = [])
apply(*induct xs arbitrary*: *n*)
apply *simp*
apply(*simp add*:*take-Cons split*:*nat.split*)
done

lemma *drop-eq-Nil*[*simp*]: (*drop n xs* = []) = (*length xs* <= *n*)
apply(*induct xs arbitrary*: *n*)
apply *simp*
apply(*simp add*:*drop-Cons split*:*nat.split*)
done

lemma *take-map*: *take n* (*map f xs*) = *map f* (*take n xs*)
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *drop-map*: *drop n* (*map f xs*) = *map f* (*drop n xs*)
apply (*induct n arbitrary*: *xs*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *rev-take*: *rev* (*take i xs*) = *drop* (*length xs* − *i*) (*rev xs*)
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *auto*)
done

lemma *rev-drop*: *rev* (*drop i xs*) = *take* (*length xs* − *i*) (*rev xs*)
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *auto*)
done

lemma *nth-take* [*simp*]: *i* < *n* ==> (*take n xs*)!*i* = *xs*!*i*
apply (*induct xs arbitrary*: *i n*, *auto*)
apply (*case-tac n*, *blast*)
apply (*case-tac i*, *auto*)
done

lemma *nth-drop* [*simp*]:
n + *i* <= *length xs* ==> (*drop n xs*)!*i* = *xs*!(*n* + *i*)

apply (*induct n arbitrary*: *xs i*, *auto*)
apply (*case-tac xs*, *auto*)
done

lemma *butlast-take*:
n <= length xs ==> butlast (*take n xs*) = *take* (*n − 1*) *xs*
by (*simp add*: *butlast-conv-take min-max.inf-absorb1 min-max.inf-absorb2*)

lemma *butlast-drop*: *butlast* (*drop n xs*) = *drop n* (*butlast xs*)
by (*simp add*: *butlast-conv-take drop-take*)

lemma *take-butlast*: *n < length xs ==> take n* (*butlast xs*) = *take n xs*
by (*simp add*: *butlast-conv-take min-max.inf-absorb1*)

lemma *drop-butlast*: *drop n* (*butlast xs*) = *butlast* (*drop n xs*)
by (*simp add*: *butlast-conv-take drop-take*)

lemma *hd-drop-conv-nth*: ⟦ *xs* ≠ []; *n* < *length xs* ⟧ ⟹ *hd*(*drop n xs*) = *xs*!*n*
by(*simp add*: *hd-conv-nth*)

lemma *set-take-subset*: *set*(*take n xs*) ⊆ *set xs*
by(*induct xs arbitrary*: *n*)(*auto simp*:*take-Cons split*:*nat.split*)

lemma *set-drop-subset*: *set*(*drop n xs*) ⊆ *set xs*
by(*induct xs arbitrary*: *n*)(*auto simp*:*drop-Cons split*:*nat.split*)

lemma *in-set-takeD*: *x* : *set*(*take n xs*) ⟹ *x* : *set xs*
using *set-take-subset* **by** *fast*

lemma *in-set-dropD*: *x* : *set*(*drop n xs*) ⟹ *x* : *set xs*
using *set-drop-subset* **by** *fast*

lemma *append-eq-conv-conj*:
(*xs* @ *ys* = *zs*) = (*xs* = *take* (*length xs*) *zs* ∧ *ys* = *drop* (*length xs*) *zs*)
apply (*induct xs arbitrary*: *zs*, *simp*, *clarsimp*)
apply (*case-tac zs*, *auto*)
done

lemma *take-add*:
i+j ≤ *length*(*xs*) ⟹ *take* (*i+j*) *xs* = *take i xs* @ *take j* (*drop i xs*)
apply (*induct xs arbitrary*: *i*, *auto*)
apply (*case-tac i*, *simp-all*)
done

lemma *append-eq-append-conv-if*:
(xs_1 @ xs_2 = ys_1 @ ys_2) =
(*if size* xs_1 ≤ *size* ys_1
then xs_1 = *take* (*size* xs_1) ys_1 ∧ xs_2 = *drop* (*size* xs_1) ys_1 @ ys_2
else take (*size* ys_1) xs_1 = ys_1 ∧ *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

40.1.15 *takeWhile* and *dropWhile*

lemma *takeWhile-dropWhile-id* [*simp*]: $takeWhile\ P\ xs\ @\ dropWhile\ P\ xs = xs$
by (*induct xs*) *auto*

lemma *takeWhile-append1* [*simp*]:
[| *x*:*set xs*; ~*P*(*x*)|] ==> *takeWhile P* (*xs* @ *ys*) = *takeWhile P xs*
by (*induct xs*) *auto*

lemma *takeWhile-append2* [*simp*]:
(!!*x*. *x* : *set xs* ==> *P x*) ==> *takeWhile P* (*xs* @ *ys*) = *xs* @ *takeWhile P ys*
by (*induct xs*) *auto*

lemma *takeWhile-tail*: ¬ *P x* ==> *takeWhile P* (*xs* @ (*x*#*l*)) = *takeWhile P xs*
by (*induct xs*) *auto*

lemma *dropWhile-append1* [*simp*]:
[| *x* : *set xs*; ~*P*(*x*)|] ==> *dropWhile P* (*xs* @ *ys*) = (*dropWhile P xs*)@*ys*
by (*induct xs*) *auto*

lemma *dropWhile-append2* [*simp*]:
(!!*x*. *x*:*set xs* ==> *P*(*x*)) ==> *dropWhile P* (*xs* @ *ys*) = *dropWhile P ys*
by (*induct xs*) *auto*

lemma *set-takeWhileD*: *x* : *set* (*takeWhile P xs*) ==> *x* : *set xs* ∧ *P x*
by (*induct xs*) (*auto split*: *split-if-asm*)

lemma *takeWhile-eq-all-conv*[*simp*]:
(*takeWhile P xs* = *xs*) = (∀ *x* ∈ *set xs*. *P x*)
by(*induct xs*, *auto*)

lemma *dropWhile-eq-Nil-conv*[*simp*]:
(*dropWhile P xs* = []) = (∀ *x* ∈ *set xs*. *P x*)
by(*induct xs*, *auto*)

lemma *dropWhile-eq-Cons-conv*:
(*dropWhile P xs* = *y*#*ys*) = (*xs* = *takeWhile P xs* @ *y* # *ys* & ¬ *P y*)
by(*induct xs*, *auto*)

The following two lemmmas could be generalized to an arbitrary property.

lemma *takeWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
takeWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *rev* (*tl* (*dropWhile* (λ*y*. *y* ≠ *x*) *xs*))
by(*induct xs*) (*auto simp*: *takeWhile-tail*[**where** *l*=[]])

lemma *dropWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ ⟹
dropWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *x* # *rev* (*takeWhile* (λ*y*. *y* ≠ *x*) *xs*)
apply(*induct xs*)
apply *simp*
apply *auto*
apply(*subst dropWhile-append2*)
apply *auto*
done

lemma *takeWhile-not-last*:
⟦ *xs* ≠ []; *distinct xs*⟧ ⟹ *takeWhile* (λ*y*. *y* ≠ *last xs*) *xs* = *butlast xs*
apply(*induct xs*)
apply *simp*
apply(*case-tac xs*)
apply(*auto*)
done

lemma *takeWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| *l* = *k*; !!*x*. *x* : *set l* ==> *P x* = *Q x* |]
==> *takeWhile P l* = *takeWhile Q k*
by (*induct k arbitrary*: *l*) (*simp-all*)

lemma *dropWhile-cong* [*fundef-cong*, *recdef-cong*]:
[| *l* = *k*; !!*x*. *x* : *set l* ==> *P x* = *Q x* |]
==> *dropWhile P l* = *dropWhile Q k*
by (*induct k arbitrary*: *l*, *simp-all*)

40.1.16 *zip*

lemma *zip-Nil* [*simp*]: *zip* [] *ys* = []
by (*induct ys*) *auto*

lemma *zip-Cons-Cons* [*simp*]: *zip* (*x* # *xs*) (*y* # *ys*) = (*x*, *y*) # *zip xs ys*
by *simp*

declare *zip-Cons* [*simp del*]

lemma *zip-Cons1*:
zip (*x*#*xs*) *ys* = (*case ys of* [] ⇒ [] | *y*#*ys* ⇒ (*x*,*y*)#*zip xs ys*)
by(*auto split*:*list.split*)

lemma *length-zip* [*simp*]:
length (*zip xs ys*) = *min* (*length xs*) (*length ys*)
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *zip-append1*:
zip (*xs* @ *ys*) *zs* =
zip xs (*take* (*length xs*) *zs*) @ *zip ys* (*drop* (*length xs*) *zs*)
by (*induct xs zs rule*:*list-induct2'*) *auto*

lemma *zip-append2*:
zip xs (*ys* @ *zs*) =
zip (*take* (*length ys*) *xs*) *ys* @ *zip* (*drop* (*length ys*) *xs*) *zs*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *zip-append* [*simp*]:
[| *length xs* = *length us*; *length ys* = *length vs* |] ==>
zip (*xs*@*ys*) (*us*@*vs*) = *zip xs us* @ *zip ys vs*

by (*simp add*: *zip-append1*)

lemma *zip-rev*:
length xs = *length ys* ==> *zip* (*rev xs*) (*rev ys*) = *rev* (*zip xs ys*)
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *map-zip-map*:
map f (*zip* (*map g xs*) *ys*) = *map* (%(*x*,*y*). *f*(*g x*, *y*)) (*zip xs ys*)
apply(*induct xs arbitrary*:*ys*) **apply** *simp*
apply(*case-tac ys*)
apply *simp-all*
done

lemma *map-zip-map2*:
map f (*zip xs* (*map g ys*)) = *map* (%(*x*,*y*). *f*(*x*, *g y*)) (*zip xs ys*)
apply(*induct xs arbitrary*:*ys*) **apply** *simp*
apply(*case-tac ys*)
apply *simp-all*
done

lemma *nth-zip* [*simp*]:
[| *i* < *length xs*; *i* < *length ys*|] ==> (*zip xs ys*)!*i* = (*xs*!*i*, *ys*!*i*)
apply (*induct ys arbitrary*: *i xs*, *simp*)
apply (*case-tac xs*)
apply (*simp-all add*: *nth.simps split*: *nat.split*)
done

lemma *set-zip*:
set (*zip xs ys*) = {(*xs*!*i*, *ys*!*i*) | *i*. *i* < *min* (*length xs*) (*length ys*)}
by (*simp add*: *set-conv-nth cong*: *rev-conj-cong*)

lemma *zip-update*:
length xs = *length ys* ==> *zip* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*]) = (*zip xs ys*)[*i*:=(*x*,*y*)]
by (*rule sym*, *simp add*: *update-zip*)

lemma *zip-replicate* [*simp*]:
zip (*replicate i x*) (*replicate j y*) = *replicate* (*min i j*) (*x*,*y*)
apply (*induct i arbitrary*: *j*, *auto*)
apply (*case-tac j*, *auto*)
done

lemma *take-zip*:
take n (*zip xs ys*) = *zip* (*take n xs*) (*take n ys*)
apply (*induct n arbitrary*: *xs ys*)
apply *simp*
apply (*case-tac xs*, *simp*)
apply (*case-tac ys*, *simp-all*)
done

lemma *drop-zip*:
drop n (zip xs ys) = zip (drop n xs) (drop n ys)
apply (*induct n arbitrary*: *xs ys*)
apply *simp*
apply (*case-tac xs*, *simp*)
apply (*case-tac ys*, *simp-all*)
done

lemma *set-zip-leftD*:
(x,y)∈ set (zip xs ys) ⟹ x ∈ set xs
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *set-zip-rightD*:
(x,y)∈ set (zip xs ys) ⟹ y ∈ set ys
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *in-set-zipE*:
(x,y) : set(zip xs ys) ⟹ (⟦ x : set xs; y : set ys ⟧ ⟹ R) ⟹ R
by(*blast dest*: *set-zip-leftD set-zip-rightD*)

40.1.17 *list-all2*

lemma *list-all2-lengthD* [*intro?*]:
list-all2 P xs ys ==> length xs = length ys
by (*simp add*: *list-all2-def*)

lemma *list-all2-Nil* [*iff*, *code*]: *list-all2 P [] ys = (ys = [])*
by (*simp add*: *list-all2-def*)

lemma *list-all2-Nil2* [*iff*, *code*]: *list-all2 P xs [] = (xs = [])*
by (*simp add*: *list-all2-def*)

lemma *list-all2-Cons* [*iff*, *code*]:
list-all2 P (x # xs) (y # ys) = (P x y ∧ list-all2 P xs ys)
by (*auto simp add*: *list-all2-def*)

lemma *list-all2-Cons1*:
list-all2 P (x # xs) ys = (∃ z zs. ys = z # zs ∧ P x z ∧ list-all2 P xs zs)
by (*cases ys*) *auto*

lemma *list-all2-Cons2*:
list-all2 P xs (y # ys) = (∃ z zs. xs = z # zs ∧ P z y ∧ list-all2 P zs ys)
by (*cases xs*) *auto*

lemma *list-all2-rev* [*iff*]:
list-all2 P (rev xs) (rev ys) = list-all2 P xs ys
by (*simp add*: *list-all2-def zip-rev cong*: *conj-cong*)

lemma *list-all2-rev1*:

list-all2 P (rev xs) ys = list-all2 P xs (rev ys)
by (*subst list-all2-rev* [*symmetric*]) *simp*

lemma *list-all2-append1*:
list-all2 P (xs @ ys) zs =
(EX us vs. zs = us @ vs ∧ length us = length xs ∧ length vs = length ys ∧
list-all2 P xs us ∧ list-all2 P ys vs)
apply (*simp add*: *list-all2-def zip-append1*)
apply (*rule iffI*)
apply (*rule-tac x = take (length xs) zs* **in** *exI*)
apply (*rule-tac x = drop (length xs) zs* **in** *exI*)
apply (*force split*: *nat-diff-split simp add*: *min-def*, *clarify*)
apply (*simp add*: *ball-Un*)
done

lemma *list-all2-append2*:
list-all2 P xs (ys @ zs) =
(EX us vs. xs = us @ vs ∧ length us = length ys ∧ length vs = length zs ∧
list-all2 P us ys ∧ list-all2 P vs zs)
apply (*simp add*: *list-all2-def zip-append2*)
apply (*rule iffI*)
apply (*rule-tac x = take (length ys) xs* **in** *exI*)
apply (*rule-tac x = drop (length ys) xs* **in** *exI*)
apply (*force split*: *nat-diff-split simp add*: *min-def*, *clarify*)
apply (*simp add*: *ball-Un*)
done

lemma *list-all2-append*:
length xs = length ys ⟹
list-all2 P (xs@us) (ys@vs) = (list-all2 P xs ys ∧ list-all2 P us vs)
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *list-all2-appendI* [*intro?*, *trans*]:
⟦ list-all2 P a b; list-all2 P c d ⟧ ⟹ list-all2 P (a@c) (b@d)
by (*simp add*: *list-all2-append list-all2-lengthD*)

lemma *list-all2-conv-all-nth*:
list-all2 P xs ys =
(length xs = length ys ∧ (∀ i < length xs. P (xs!i) (ys!i)))
by (*force simp add*: *list-all2-def set-zip*)

lemma *list-all2-trans*:
assumes *tr*: !!*a b c. P1 a b ==> P2 b c ==> P3 a c*
shows !!*bs cs. list-all2 P1 as bs ==> list-all2 P2 bs cs ==> list-all2 P3 as cs*
(**is** !!*bs cs. PROP ?Q as bs cs*)
proof (*induct as*)
fix *x xs bs* **assume** *I1*: !!*bs cs. PROP ?Q xs bs cs*
show !!*cs. PROP ?Q (x # xs) bs cs*
proof (*induct bs*)

fix *y ys cs* **assume** *I2*: !!*cs*. *PROP ?Q* (*x # xs*) *ys cs*
show *PROP ?Q* (*x # xs*) (*y # ys*) *cs*
by (*induct cs*) (*auto intro*: *tr I1 I2*)
qed *simp*
qed *simp*

lemma *list-all2-all-nthI* [*intro?*]:
length a = length b ⟹ (⋀*n*. *n < length a* ⟹ *P* (*a*!*n*) (*b*!*n*)) ⟹ *list-all2 P a b*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2I*:
∀ *x* ∈ *set* (*zip a b*). *split P x* ⟹ *length a = length b* ⟹ *list-all2 P a b*
by (*simp add*: *list-all2-def*)

lemma *list-all2-nthD*:
⟦ *list-all2 P xs ys*; *p < size xs* ⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-nthD2*:
⟦*list-all2 P xs ys*; *p < size ys*⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
by (*frule list-all2-lengthD*) (*auto intro*: *list-all2-nthD*)

lemma *list-all2-map1*:
list-all2 P (*map f as*) *bs* = *list-all2* (λ*x y*. *P* (*f x*) *y*) *as bs*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-map2*:
list-all2 P as (*map f bs*) = *list-all2* (λ*x y*. *P x* (*f y*)) *as bs*
by (*auto simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-refl* [*intro?*]:
(⋀*x*. *P x x*) ⟹ *list-all2 P xs xs*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-update-cong*:
⟦ *i*<*size xs*; *list-all2 P xs ys*; *P x y* ⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
by (*simp add*: *list-all2-conv-all-nth nth-list-update*)

lemma *list-all2-update-cong2*:
⟦*list-all2 P xs ys*; *P x y*; *i < length ys*⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
by (*simp add*: *list-all2-lengthD list-all2-update-cong*)

lemma *list-all2-takeI* [*simp*,*intro?*]:
list-all2 P xs ys ⟹ *list-all2 P* (*take n xs*) (*take n ys*)
apply (*induct xs arbitrary*: *n ys*)
apply *simp*
apply (*clarsimp simp add*: *list-all2-Cons1*)
apply (*case-tac n*)
apply *auto*

done

lemma *list-all2-dropI* [*simp*,*intro?*]:
list-all2 P as bs $\Longrightarrow$ *list-all2 P* (*drop n as*) (*drop n bs*)
apply (*induct as arbitrary*: *n bs*, *simp*)
apply (*clarsimp simp add*: *list-all2-Cons1*)
apply (*case-tac n*, *simp*, *simp*)
done

lemma *list-all2-mono* [*intro?*]:
list-all2 P xs ys $\Longrightarrow$ ($\bigwedge$*xs ys*. *P xs ys* $\Longrightarrow$ *Q xs ys*) $\Longrightarrow$ *list-all2 Q xs ys*
apply (*induct xs arbitrary*: *ys*, *simp*)
apply (*case-tac ys*, *auto*)
done

lemma *list-all2-eq*:
xs = *ys* $\longleftrightarrow$ *list-all2* (*op* =) *xs ys*
by (*induct xs ys rule*: *list-induct2′*) *auto*

40.1.18 *foldl* and *foldr*

lemma *foldl-append* [*simp*]:
foldl f a (*xs* @ *ys*) = *foldl f* (*foldl f a xs*) *ys*
by (*induct xs arbitrary*: *a*) *auto*

lemma *foldr-append*[*simp*]: *foldr f* (*xs* @ *ys*) *a* = *foldr f xs* (*foldr f ys a*)
by (*induct xs*) *auto*

lemma *foldr-map*: *foldr g* (*map f xs*) *a* = *foldr* (*g o f*) *xs a*
by(*induct xs*) *simp-all*

For efficient code generation: avoid intermediate list.

lemma *foldl-map*[*code unfold*]:
foldl g a (*map f xs*) = *foldl* (%*a x*. *g a* (*f x*)) *a xs*
by(*induct xs arbitrary*:*a*) *simp-all*

lemma *foldl-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f a x* = *g a x* |]
==> *foldl f a l* = *foldl g b k*
by (*induct k arbitrary*: *a b l*) *simp-all*

lemma *foldr-cong* [*fundef-cong*, *recdef-cong*]:
[| *a* = *b*; *l* = *k*; !!*a x*. *x* : *set l* ==> *f x a* = *g x a* |]
==> *foldr f l a* = *foldr g k b*
by (*induct k arbitrary*: *a b l*) *simp-all*

lemma (**in** *semigroup-add*) *foldl-assoc*:
shows *foldl op*+ (*x*+*y*) *zs* = *x* + (*foldl op*+ *y zs*)
by (*induct zs arbitrary*: *y*) (*simp-all add*:*add-assoc*)

lemma (**in** *monoid-add*) *foldl-absorb0*:
shows $x + (foldl\ op+\ 0\ zs) = foldl\ op+\ x\ zs$
by (*induct zs*) (*simp-all add*:*foldl-assoc*)

The "First Duality Theorem" in Bird & Wadler:

lemma *foldl-foldr1-lemma*:
$foldl\ op\ +\ a\ xs = a + foldr\ op\ +\ xs\ (0::'a::monoid\text{-}add)$
by (*induct xs arbitrary*: *a*) (*auto simp*:*add-assoc*)

corollary *foldl-foldr1*:
$foldl\ op\ +\ 0\ xs = foldr\ op\ +\ xs\ (0::'a::monoid\text{-}add)$
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*)

40.1.19 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*

lemma *length-concat*: *length* (*concat xss*) = *listsum* (*map length xss*)
by (*induct xss*) *simp-all*

For efficient code generation — *listsum* is not tail recursive but *foldl* is.

lemma *listsum*[*code unfold*]: *listsum xs* = *foldl* (*op* +) *0 xs*
by(*simp add*:*listsum-foldr foldl-foldr1*)

Some syntactic sugar for summing a function over a list:

syntax
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3SUM -<−-. -*) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-listsum :: *pttrn* => *'a list* => *'b* => *'b* ((*3*$\sum$ -←-. -) [*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-triv*: ($\sum$ *x*←*xs. r*) = *of-nat* (*length xs*) ∗ *r*
by (*induct xs*) (*simp-all add*: *left-distrib*)

lemma *listsum-0* [*simp*]: ($\sum$ *x*←*xs. 0*) = *0*
by (*induct xs*) (*simp-all add*: *left-distrib*)

For non-Abelian groups *xs* needs to be reversed on one side:

lemma *uminus-listsum-map*:
fixes *f* :: *'a* ⇒ *'b*::*ab-group-add*
shows − *listsum* (*map f xs*) = (*listsum* (*map* (*uminus o f*) *xs*))
by (*induct xs*) *simp-all*

40.1.20 *upt*

lemma *upt-rec*[*code*]: [*i*..<*j*] = (*if* *i*<*j* *then* *i*#[*Suc* *i*..<*j*] *else* [])
— simp does not terminate!
by (*induct* *j*) *auto*

lemma *upt-conv-Nil* [*simp*]: *j* <= *i* ==> [*i*..<*j*] = []
by (*subst* *upt-rec*) *simp*

lemma *upt-eq-Nil-conv*[*simp*]: ([*i*..<*j*] = []) = (*j* = *0* ∨ *j* <= *i*)
by(*induct* *j*)*simp-all*

lemma *upt-eq-Cons-conv*:
([*i*..<*j*] = *x*#*xs*) = (*i* < *j* & *i* = *x* & [*i*+*1*..<*j*] = *xs*)
apply(*induct* *j* *arbitrary*: *x* *xs*)
apply *simp*
apply(*clarsimp* *simp* *add*: *append-eq-Cons-conv*)
apply *arith*
done

lemma *upt-Suc-append*: *i* <= *j* ==> [*i*..<(*Suc* *j*)] = [*i*..<*j*]@[*j*]
— Only needed if *upt-Suc* is deleted from the simpset.
by *simp*

lemma *upt-conv-Cons*: *i* < *j* ==> [*i*..<*j*] = *i* # [*Suc* *i*..<*j*]
by (*simp* *add*: *upt-rec*)

lemma *upt-add-eq-append*: *i*<=*j* ==> [*i*..<*j*+*k*] = [*i*..<*j*]@[*j*..<*j*+*k*]
— LOOPS as a simprule, since *j* <= *j*.
by (*induct* *k*) *auto*

lemma *length-upt* [*simp*]: *length* [*i*..<*j*] = *j* − *i*
by (*induct* *j*) (*auto* *simp* *add*: *Suc-diff-le*)

lemma *nth-upt* [*simp*]: *i* + *k* < *j* ==> [*i*..<*j*] ! *k* = *i* + *k*
apply (*induct* *j*)
apply (*auto* *simp* *add*: *less-Suc-eq* *nth-append* *split*: *nat-diff-split*)
done

lemma *hd-upt*[*simp*]: *i* < *j* ⟹ *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 (λ*i*. *xs* ! *i*) [*0*..<*length xs*] = *xs*
by (*rule nth-equalityI*, *auto*)

lemma *list-all2-antisym*:

⟦ (⋀*x y*. ⟦*P x y*; *Q y x*⟧ ⟹ *x* = *y*); *list-all2 P xs ys*; *list-all2 Q ys xs* ⟧
⟹ *xs* = *ys*
apply (*simp add*: *list-all2-conv-all-nth*)
apply (*rule nth-equalityI*, *blast*, *simp*)
done

lemma *take-equalityI*: (∀*i*. *take i xs* = *take i ys*) ==> *xs* = *ys*
— The famous take-lemma.
apply (*drule-tac x* = *max* (*length xs*) (*length ys*) **in** *spec*)
apply (*simp add*: *le-max-iff-disj take-all*)
done

lemma *take-Cons'*:
take n (*x* # *xs*) = (*if n* = *0 then* [] *else x* # *take* (*n* − *1*) *xs*)
by (*cases n*) *simp-all*

lemma *drop-Cons'*:
drop n (*x* # *xs*) = (*if n* = *0 then x* # *xs else drop* (*n* − *1*) *xs*)
by (*cases n*) *simp-all*

lemma *nth-Cons'*: (*x* # *xs*)!*n* = (*if n* = *0 then x else xs*!(*n* − *1*))
by (*cases n*) *simp-all*

lemmas *take-Cons-number-of* = *take-Cons'*[*of number-of v*,*standard*]
lemmas *drop-Cons-number-of* = *drop-Cons'*[*of number-of v*,*standard*]
lemmas *nth-Cons-number-of* = *nth-Cons'*[*of - - number-of v*,*standard*]

declare *take-Cons-number-of* [*simp*]
drop-Cons-number-of [*simp*]
nth-Cons-number-of [*simp*]

40.1.21 *distinct* and *remdups*

lemma *distinct-append* [*simp*]:
distinct (*xs* @ *ys*) = (*distinct xs* ∧ *distinct ys* ∧ *set xs* ∩ *set ys* = {})
by (*induct xs*) *auto*

lemma *distinct-rev*[*simp*]: *distinct*(*rev xs*) = *distinct xs*
by(*induct xs*) *auto*

lemma *set-remdups* [*simp*]: *set* (*remdups xs*) = *set xs*
by (*induct xs*) (*auto simp add*: *insert-absorb*)

lemma *distinct-remdups* [*iff*]: *distinct* (*remdups xs*)
by (*induct xs*) *auto*

lemma *distinct-remdups-id*: *distinct xs* ==> *remdups xs* = *xs*
by (*induct xs*, *auto*)

lemma *remdups-id-iff-distinct* [*simp*]: *remdups xs = xs ⟷ distinct xs*
by (*metis distinct-remdups distinct-remdups-id*)

lemma *finite-distinct-list*: *finite A ⟹ EX xs. set xs = A & distinct xs*
by (*metis distinct-remdups finite-list set-remdups*)

lemma *remdups-eq-nil-iff* [*simp*]: (*remdups x = []*) = (*x = []*)
by (*induct x*, *auto*)

lemma *remdups-eq-nil-right-iff* [*simp*]: ([] = *remdups x*) = (*x = []*)
by (*induct x*, *auto*)

lemma *length-remdups-leq*[*iff*]: *length*(*remdups xs*) <= *length xs*
by (*induct xs*) *auto*

lemma *length-remdups-eq*[*iff*]:
(*length* (*remdups xs*) = *length xs*) = (*remdups xs = xs*)
apply(*induct xs*)
apply *auto*
apply(*subgoal-tac length* (*remdups xs*) <= *length xs*)
apply *arith*
apply(*rule length-remdups-leq*)
done

lemma *distinct-map*:
distinct(*map f xs*) = (*distinct xs* & *inj-on f* (*set xs*))
by (*induct xs*) *auto*

lemma *distinct-filter* [*simp*]: *distinct xs* ==> *distinct* (*filter P xs*)
by (*induct xs*) *auto*

lemma *distinct-upt*[*simp*]: *distinct*[*i*..<*j*]
by (*induct j*) *auto*

lemma *distinct-take*[*simp*]: *distinct xs ⟹ distinct* (*take i xs*)
apply(*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)
apply *simp-all*
apply(*blast dest*:*in-set-takeD*)
done

lemma *distinct-drop*[*simp*]: *distinct xs ⟹ distinct* (*drop i xs*)
apply(*induct xs arbitrary*: *i*)
apply *simp*
apply (*case-tac i*)

apply *simp-all*
done

lemma *distinct-list-update*:
assumes *d*: *distinct xs* **and** *a*: $a \notin set\ xs - \{xs!i\}$
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*:
$distinct\ xs = (\forall i < size\ xs.\ \forall j < size\ xs.\ i \neq j \longrightarrow xs!i \neq xs!j)$
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 ∈ set xs)
    case False with Cons show ?thesis by simp
  next
    case True with Cons.prems
    have card (set xs) = Suc (length xs)
      by (simp add: card-insert-if split: split-if-asm)
    moreover have card (set xs) ≤ length xs by (rule card-length)
    ultimately have False by simp
    thus ?thesis ..
  qed
qed

lemma not-distinct-decomp: ~ distinct ws ==> EX xs ys zs y. ws = xs@[y]@ys@[y]@zs
apply (induct n == length ws arbitrary:ws) apply simp
apply(case-tac ws) apply simp
apply (simp split:split-if-asm)
apply (metis Cons-eq-appendI eq-Nil-appendI split-list)
done

lemma length-remdups-concat:
 length(remdups(concat xss)) = card(⋃ xs ∈ set xss. set xs)
by(simp add: set-concat distinct-card[symmetric])
```

40.1.22 *remove1*

```
lemma remove1-append:
  remove1 x (xs @ ys) =
  (if x ∈ set xs then remove1 x xs @ ys else xs @ remove1 x ys)
by (induct xs) auto

lemma in-set-remove1[simp]:
  a ≠ b ⟹ 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*]:
¬ *P x* ⟹ *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*

40.1.23 *replicate*

lemma *length-replicate* [*simp*]: *length* (*replicate n x*) = *n*
by (*induct n*) *auto*

lemma *map-replicate* [*simp*]: *map f* (*replicate n x*) = *replicate n* (*f x*)
by (*induct n*) *auto*

lemma *replicate-app-Cons-same*:
(*replicate n x*) @ (*x* # *xs*) = *x* # *replicate n x* @ *xs*
by (*induct n*) *auto*

lemma *rev-replicate* [*simp*]: *rev* (*replicate n x*) = *replicate n x*
apply (*induct n*, *simp*)
apply (*simp add*: *replicate-app-Cons-same*)
done

lemma *replicate-add*: *replicate* (*n* + *m*) *x* = *replicate n x* @ *replicate m x*
by (*induct n*) *auto*

Courtesy of Matthias Daum:

lemma *append-replicate-commute*:
replicate n x @ *replicate k x* = *replicate k x* @ *replicate n x*
apply (*simp add*: *replicate-add* [*THEN sym*])
apply (*simp add*: *add-commute*)
done

lemma *hd-replicate* [*simp*]: *n* ≠ *0* ==> *hd* (*replicate n x*) = *x*

by (*induct n*) *auto*

lemma *tl-replicate* [*simp*]: $n \neq 0 \Longrightarrow tl\ (replicate\ n\ x) = replicate\ (n - 1)\ x$
by (*induct n*) *auto*

lemma *last-replicate* [*simp*]: $n \neq 0 \Longrightarrow last\ (replicate\ n\ x) = x$
by (*atomize* (*full*), *induct n*) *auto*

lemma *nth-replicate*[*simp*]: $i < n \Longrightarrow (replicate\ n\ x)!i = x$
apply (*induct n arbitrary*: *i*, *simp*)
apply (*simp add*: *nth-Cons split*: *nat.split*)
done

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate*[*simp*]: *take i* (*replicate k x*) = *replicate* (*min i k*) *x*
apply (*case-tac* $k \leq i$)
apply (*simp add*: *min-def*)
apply (*drule not-leE*)
apply (*simp add*: *min-def*)
apply (*subgoal-tac replicate k x* = *replicate i x* @ *replicate* ($k - i$) *x*)
apply *simp*
apply (*simp add*: *replicate-add* [*symmetric*])
done

lemma *drop-replicate*[*simp*]: *drop i* (*replicate k x*) = *replicate* ($k-i$) *x*
apply (*induct k arbitrary*: *i*)
apply *simp*
apply *clarsimp*
apply (*case-tac i*)
apply *simp*
apply *clarsimp*
done

lemma *set-replicate-Suc*: *set* (*replicate* (*Suc n*) *x*) = $\{x\}$
by (*induct n*) *auto*

lemma *set-replicate* [*simp*]: $n \neq 0 \Longrightarrow set\ (replicate\ n\ x) = \{x\}$
by (*fast dest*!: *not0-implies-Suc intro*!: *set-replicate-Suc*)

lemma *set-replicate-conv-if*: *set* (*replicate n x*) = (*if n* = *0 then* {} *else* $\{x\}$)
by *auto*

lemma *in-set-replicateD*: $x : set\ (replicate\ n\ y) \Longrightarrow x = y$
by (*simp add*: *set-replicate-conv-if split*: *split-if-asm*)

lemma *replicate-append-same*:
replicate i x @ [*x*] = *x* # *replicate i x*
by (*induct i*) *simp-all*

lemma *map-replicate-trivial*:
map (λ*i*. *x*) [*0*..<*i*] = *replicate i x*
by (*induct i*) (*simp-all add*: *replicate-append-same*)

40.1.24 *rotate1* and *rotate*

lemma *rotate-simps*[*simp*]: *rotate1* [] = [] ∧ *rotate1* (*x*#*xs*) = *xs* @ [*x*]
by(*simp add*:*rotate1-def*)

lemma *rotate0*[*simp*]: *rotate 0* = *id*
by(*simp add*:*rotate-def*)

lemma *rotate-Suc*[*simp*]: *rotate* (*Suc n*) *xs* = *rotate1*(*rotate n xs*)
by(*simp add*:*rotate-def*)

lemma *rotate-add*:
rotate (*m*+*n*) = *rotate m o rotate n*
by(*simp add*:*rotate-def funpow-add*)

lemma *rotate-rotate*: *rotate m* (*rotate n xs*) = *rotate* (*m*+*n*) *xs*
by(*simp add*:*rotate-add*)

lemma *rotate1-rotate-swap*: *rotate1* (*rotate n xs*) = *rotate n* (*rotate1 xs*)
by(*simp add*:*rotate-def funpow-swap1*)

lemma *rotate1-length01*[*simp*]: *length xs* <= *1* ⟹ *rotate1 xs* = *xs*
by(*cases xs*) *simp-all*

lemma *rotate-length01*[*simp*]: *length xs* <= *1* ⟹ *rotate n xs* = *xs*
apply(*induct n*)
apply *simp*
apply (*simp add*:*rotate-def*)
done

lemma *rotate1-hd-tl*: *xs* ≠ [] ⟹ *rotate1 xs* = *tl xs* @ [*hd xs*]
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *rotate-drop-take*:
rotate n xs = *drop* (*n mod length xs*) *xs* @ *take* (*n mod length xs*) *xs*
apply(*induct n*)
apply *simp*
apply(*simp add*:*rotate-def*)
apply(*cases xs* = [])
apply (*simp*)
apply(*case-tac n mod length xs* = *0*)
apply(*simp add*:*mod-Suc*)
apply(*simp add*: *rotate1-hd-tl drop-Suc take-Suc*)
apply(*simp add*:*mod-Suc rotate1-hd-tl drop-Suc*[*symmetric*] *drop-tl*[*symmetric*]

take-hd-drop linorder-not-le)
done

lemma *rotate-conv-mod*: *rotate n xs* = *rotate* (*n mod length xs*) *xs*
by(*simp add*:*rotate-drop-take*)

lemma *rotate-id*[*simp*]: *n mod length xs* = *0* ⟹ *rotate n xs* = *xs*
by(*simp add*:*rotate-drop-take*)

lemma *length-rotate1*[*simp*]: *length*(*rotate1 xs*) = *length xs*
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *length-rotate*[*simp*]: *length*(*rotate n xs*) = *length xs*
by (*induct n arbitrary*: *xs*) (*simp-all add*:*rotate-def*)

lemma *distinct1-rotate*[*simp*]: *distinct*(*rotate1 xs*) = *distinct xs*
by(*simp add*:*rotate1-def split*:*list.split*) *blast*

lemma *distinct-rotate*[*simp*]: *distinct*(*rotate n xs*) = *distinct xs*
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate-map*: *rotate n* (*map f xs*) = *map f* (*rotate n xs*)
by(*simp add*:*rotate-drop-take take-map drop-map*)

lemma *set-rotate1*[*simp*]: *set*(*rotate1 xs*) = *set xs*
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *set-rotate*[*simp*]: *set*(*rotate n xs*) = *set xs*
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate1-is-Nil-conv*[*simp*]: (*rotate1 xs* = []) = (*xs* = [])
by(*simp add*:*rotate1-def split*:*list.split*)

lemma *rotate-is-Nil-conv*[*simp*]: (*rotate n xs* = []) = (*xs* = [])
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate-rev*:
rotate n (*rev xs*) = *rev*(*rotate* (*length xs* − (*n mod length xs*)) *xs*)
apply(*simp add*:*rotate-drop-take rev-drop rev-take*)
apply(*cases length xs* = *0*)
apply *simp*
apply(*cases n mod length xs* = *0*)
apply *simp*
apply(*simp add*:*rotate-drop-take rev-drop rev-take*)
done

lemma *hd-rotate-conv-nth*: *xs* ≠ [] ⟹ *hd*(*rotate n xs*) = *xs*!(*n mod length xs*)
apply(*simp add*:*rotate-drop-take hd-append hd-drop-conv-nth hd-conv-nth*)
apply(*subgoal-tac length xs* ≠ *0*)

prefer *2* **apply** *simp*
using *mod-less-divisor*[*of length xs n*] **by** *arith*

40.1.25 *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* ∧ *i* ∈ *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*) ⊆ *set xs*
by(*auto simp add*:*set-sublist*)

lemma *notin-set-sublistI*[*simp*]: *x* ∉ *set xs* ⟹ *x* ∉ *set*(*sublist xs I*)
by(*auto simp add*:*set-sublist*)

lemma *in-set-sublistD*: *x* ∈ *set*(*sublist xs I*) ⟹ *x* ∈ *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* ⟹ *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 ⟹ *filter* (%*x*. *x* ∈ *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* ⟶ *x* ≠ *a* **by** *auto*
ultimately show *?case* **by**(*simp add*: *sublist-Cons cong*:*filter-cong*)
qed

40.1.26 *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

40.2 Sorting

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not yet available. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

context *linorder*
begin

lemma *sorted-Cons*: *sorted* (*x*#*xs*) = (*sorted xs* & (*ALL y*:*set xs*. *x* <= *y*))
apply(*induct xs arbitrary*: *x*) **apply** *simp*
by *simp* (*blast intro*: *order-trans*)

lemma *sorted-append*:
sorted (*xs*@*ys*) = (*sorted xs* & *sorted ys* & (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *x*≤*y*))
by (*induct xs*) (*auto simp add*:*sorted-Cons*)

lemma *set-insort*: *set*(*insort x xs*) = *insert x* (*set xs*)
by (*induct xs*) *auto*

lemma *set-sort*[*simp*]: *set*(*sort xs*) = *set xs*
by (*induct xs*) (*simp-all add*:*set-insort*)

lemma *distinct-insort*: *distinct* (*insort x xs*) = (*x* ∉ *set xs* ∧ *distinct xs*)
by(*induct xs*)(*auto simp*:*set-insort*)

lemma *distinct-sort*[*simp*]: *distinct* (*sort xs*) = *distinct xs*
by(*induct xs*)(*simp-all add*:*distinct-insort set-sort*)

lemma *sorted-insort*: *sorted* (*insort x xs*) = *sorted xs*
apply (*induct xs*)
apply(*auto simp*:*sorted-Cons set-insort*)
done

theorem *sorted-sort*[*simp*]: *sorted* (*sort xs*)
by (*induct xs*) (*auto simp*:*sorted-insort*)

lemma *insort-is-Cons*: ∀ *x*∈*set xs*. *a* ≤ *x* ⟹ *insort a xs* = *a* # *xs*
by (*cases xs*) *auto*

lemma *sorted-remove1*: *sorted xs* ⟹ *sorted* (*remove1 a xs*)
by (*induct xs*, *auto simp add*: *sorted-Cons*)

lemma *insort-remove1*: ⟦ *a* ∈ *set xs*; *sorted xs* ⟧ ⟹ *insort a* (*remove1 a xs*) = *xs*
by (*induct xs*, *auto simp add*: *sorted-Cons insort-is-Cons*)

lemma *sorted-remdups*[*simp*]:
sorted l ⟹ *sorted* (*remdups l*)
by (*induct l*) (*auto simp*: *sorted-Cons*)

lemma *sorted-distinct-set-unique*:
assumes *sorted xs distinct xs sorted ys distinct ys set xs* = *set ys*
shows *xs* = *ys*
proof −
from *assms* **have** *1*: *length xs* = *length ys* **by** (*auto dest*!: *distinct-card*)
from *assms* **show** *?thesis*
proof(*induct rule*:*list-induct2*[*OF 1*])
case *1* **show** *?case* **by** *simp*
next
case *2* **thus** *?case* **by** (*simp add*:*sorted-Cons*)
(*metis Diff-insert-absorb antisym insertE insert-iff*)
qed
qed

lemma *finite-sorted-distinct-unique*:
shows *finite A* ⟹ *EX*! *xs*. *set xs* = *A* & *sorted xs* & *distinct xs*
apply(*drule finite-distinct-list*)
apply *clarify*
apply(*rule-tac a*=*sort xs* **in** *ex1I*)
apply (*auto simp*: *sorted-distinct-set-unique*)
done

end

lemma *sorted-upt*[*simp*]: *sorted*[*i*..<*j*]
by (*induct j*) (*simp-all add*:*sorted-append*)

40.2.1 *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* ⟹
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

40.2.2 *upto*: the generic interval-list

class *finite-intvl-succ* = *linorder* +
fixes *successor* :: 'a ⇒ 'a
assumes *finite-intvl*: *finite*{*a..b*}
and *successor-incr*: *a* < *successor a*
and *ord-discrete*: ¬(∃ *x*. *a* < *x* & *x* < *successor a*)

context *finite-intvl-succ*
begin

definition
upto :: 'a ⇒ 'a ⇒ 'a *list* ((*1*[-../-])) **where**
upto i j == *sorted-list-of-set* {*i..j*}

lemma *upto*[*simp*]: *set*[*a..b*] = {*a..b*} & *sorted*[*a..b*] & *distinct*[*a..b*]
by(*simp add*:*upto-def finite-intvl*)

lemma *insert-intvl*: *i* ≤ *j* ⟹ *insert i* {*successor i..j*} = {*i..j*}
apply(*insert successor-incr*[*of i*])
apply(*auto simp*: *atLeastAtMost-def atLeast-def atMost-def*)
apply(*metis ord-discrete less-le not-le*)
done

lemma *sorted-list-of-set-rec*: *i* ≤ *j* ⟹
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* ≤ *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:

instantiation *int*:: *finite-intvl-succ*
begin

definition
successor-int-def: *successor* = (%*i*::*int*. *i*+*1*)

instance
by *intro-classes* (*simp-all add*: *successor-int-def*)

end

Now [*i..j*] is defined for integers.

hide (**open**) *const successor*

40.2.3 *lists*: the list-forming operator over sets

inductive-set
lists :: *'a set* => *'a list set*
for *A* :: *'a set*
where
Nil [*intro*!]: []: *lists A*
| *Cons* [*intro*!,*noatp*]: [| *a*: *A*;*l*: *lists A*|] ==> *a*#*l* : *lists A*

inductive-cases *listsE* [*elim*!,*noatp*]: *x*#*l* : *lists A*
inductive-cases *listspE* [*elim*!,*noatp*]: *listsp A* (*x* # *l*)

lemma *listsp-mono* [*mono*]: *A* ≤ *B* ==> *listsp A* ≤ *listsp B*
by (*rule predicate1I*, *erule listsp.induct*, *blast*+)

lemmas *lists-mono* = *listsp-mono* [*to-set pred-subset-eq*]

lemma *listsp-infI*:
assumes *l*: *listsp A l* **shows** *listsp B l* ==> *listsp* (*inf A B*) *l* **using** *l*
by *induct blast*+

lemmas *lists-IntI* = *listsp-infI* [*to-set*]

lemma *listsp-inf-eq* [*simp*]: *listsp* (*inf A B*) = *inf* (*listsp A*) (*listsp B*)
proof (*rule mono-inf* [**where** *f*=*listsp*, *THEN order-antisym*])
show *mono listsp* **by** (*simp add*: *mono-def listsp-mono*)
show *inf* (*listsp A*) (*listsp B*) ≤ *listsp* (*inf A B*) **by** (*blast intro*!: *listsp-infI predicate1I*)
qed

lemmas *listsp-conj-eq* [*simp*] = *listsp-inf-eq* [*simplified inf-fun-eq inf-bool-eq*]

lemmas *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set pred-equals-eq*]

lemma *append-in-listsp-conv* [*iff*]:
(*listsp A* (*xs* @ *ys*)) = (*listsp A xs* ∧ *listsp A ys*)
by (*induct xs*) *auto*

lemmas *append-in-lists-conv* [*iff*] = *append-in-listsp-conv* [*to-set*]

lemma *in-listsp-conv-set*: (*listsp A xs*) = (∀ *x* ∈ *set xs*. *A x*)
— eliminate *listsp* in favour of *set*
by (*induct xs*) *auto*

lemmas *in-lists-conv-set* = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest*!,*noatp*]: *listsp A xs* ==> ∀ *x*∈*set xs*. *A x*
by (*rule in-listsp-conv-set* [*THEN iffD1*])

lemmas *in-listsD* [*dest*!,*noatp*] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro*!,*noatp*]: ∀ *x*∈*set xs*. *A x* ==> *listsp A xs*
by (*rule in-listsp-conv-set* [*THEN iffD2*])

lemmas *in-listsI* [*intro*!,*noatp*] = *in-listspI* [*to-set*]

lemma *lists-UNIV* [*simp*]: *lists UNIV* = *UNIV*
by *auto*

40.2.4 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

40.2.5 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*)

40.3 Relations on Lists

40.3.1 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* ≠ *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

40.3.2 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 $r == \{(x,y).\ \exists\ a\ v.\ y = x\ @\ a\ \#\ v \vee$
$(\exists\ u\ a\ b\ v\ w.\ (a,b) \in r \wedge x = u\ @\ (a\ \#\ v) \wedge y = u\ @\ (b\ \#\ w))\}$

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*:
$((x,y) : lexord\ r) =$
$((length\ x < length\ y \wedge take\ (length\ x)\ y = x) \vee$
$(\exists\, i.\ i < min(length\ x)(length\ y)\ \&\ take\ i\ x = take\ i\ y\ \&\ (x!i,y!i) \in r))$
apply (*unfold lexord-def Let-def*, *clarsimp*)
apply (*rule-tac f* $= (\%\ a\ b.\ a \vee b)$ **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*

40.4 Lexicographic combination of measure functions

These are useful for termination proofs

definition
measures fs = *inv-image* (*lex less-than*) (%*a*. *map* (%*f*. *f a*) *fs*)

lemma *wf-measures*[*recdef-wf*, *simp*]: *wf* (*measures fs*)
unfolding *measures-def*
by *blast*

lemma *in-measures*[*simp*]:
(*x*, *y*) ∈ *measures* [] = *False*
(*x*, *y*) ∈ *measures* (*f* # *fs*)
= (*f x* < *f y* ∨ (*f x* = *f y* ∧ (*x*, *y*) ∈ *measures fs*))
unfolding *measures-def*
by *auto*

lemma *measures-less*: *f x* < *f y* ==> (*x*, *y*) ∈ *measures* (*f*#*fs*)
by *simp*

lemma *measures-lesseq*: *f x* <= *f y* ==> (*x*, *y*) ∈ *measures fs* ==> (*x*, *y*) ∈ *measures* (*f*#*fs*)
by *auto*

40.4.1 Lifting a Relation on List Elements to the Lists

inductive-set
listrel :: ('*a* * '*a*)*set* => ('*a list* * '*a list*)*set*
for *r* :: ('*a* * '*a*)*set*
where
Nil: ([],[]) ∈ *listrel r*
| *Cons*: [| (*x*,*y*) ∈ *r*; (*xs*,*ys*) ∈ *listrel r* |] ==> (*x*#*xs*, *y*#*ys*) ∈ *listrel r*

inductive-cases *listrel-Nil1* [*elim*!]: ([],*xs*) ∈ *listrel r*
inductive-cases *listrel-Nil2* [*elim*!]: (*xs*,[]) ∈ *listrel r*
inductive-cases *listrel-Cons1* [*elim*!]: (*y*#*ys*,*xs*) ∈ *listrel r*
inductive-cases *listrel-Cons2* [*elim*!]: (*xs*,*y*#*ys*) ∈ *listrel r*

lemma *listrel-mono*: *r* ⊆ *s* ⟹ *listrel r* ⊆ *listrel s*
apply *clarify*
apply (*erule listrel.induct*)
apply (*blast intro*: *listrel.intros*)+
done

lemma *listrel-subset*: *r* ⊆ *A* × *A* ⟹ *listrel r* ⊆ *lists A* × *lists A*
apply *clarify*
apply (*erule listrel.induct*, *auto*)
done

lemma *listrel-refl*: *refl A r* ⟹ *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* ⟹ *sym* (*listrel r*)
apply (*auto simp add*: *sym-def*)
apply (*erule listrel.induct*)
apply (*blast intro*: *listrel.intros*)+
done

lemma *listrel-trans*: *trans r* ⟹ *trans* (*listrel r*)
apply (*simp add*: *trans-def*)
apply (*intro allI*)
apply (*rule impI*)
apply (*erule listrel.induct*)
apply (*blast intro*: *listrel.intros*)+
done

theorem *equiv-listrel*: *equiv A r* ⟹ *equiv* (*lists A*) (*listrel r*)
by (*simp add*: *equiv-def listrel-refl 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*)

40.5 Miscellany

40.5.1 Characters and strings

datatype *nibble* =
Nibble0 | *Nibble1* | *Nibble2* | *Nibble3* | *Nibble4* | *Nibble5* | *Nibble6* | *Nibble7*
| *Nibble8* | *Nibble9* | *NibbleA* | *NibbleB* | *NibbleC* | *NibbleD* | *NibbleE* | *NibbleF*

lemma *UNIV-nibble*:
UNIV = {*Nibble0*, *Nibble1*, *Nibble2*, *Nibble3*, *Nibble4*, *Nibble5*, *Nibble6*, *Nibble7*,
Nibble8, *Nibble9*, *NibbleA*, *NibbleB*, *NibbleC*, *NibbleD*, *NibbleE*, *NibbleF*} (**is** - = *?A*)
proof (*rule UNIV-eq-I*)
fix *x* **show** *x* ∈ *?A* **by** (*cases x*) *simp-all*
qed

instance *nibble* :: *finite*
by *default* (*simp add*: *UNIV-nibble*)

datatype *char* = *Char nibble nibble*
— Note: canonical order of character encoding coincides with standard term

ordering

lemma *UNIV-char*:
UNIV = *image* (*split Char*) (*UNIV* × *UNIV*)
proof (*rule UNIV-eq-I*)
fix *x* **show** *x* ∈ *image* (*split Char*) (*UNIV* × *UNIV*) **by** (*cases x*) *auto*
qed

instance *char* :: *finite*
by *default* (*simp add*: *UNIV-char*)

types *string* = *char list*

syntax
-Char :: *xstr* => *char* (*CHR* -)
-String :: *xstr* => *string* (-)

setup *StringSyntax.setup*

40.6 Size function

lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (*list-size f*)
by (*rule is-measure-trivial*)

lemma [*measure-function*]: *is-measure f* ⟹ *is-measure* (*option-size f*)
by (*rule is-measure-trivial*)

lemma *list-size-estimation*[*termination-simp*]:
x ∈ *set xs* ⟹ *y* < *f x* ⟹ *y* < *list-size f xs*
by (*induct xs*) *auto*

lemma *list-size-estimation'*[*termination-simp*]:
x ∈ *set xs* ⟹ *y* ≤ *f x* ⟹ *y* ≤ *list-size f xs*
by (*induct xs*) *auto*

lemma *list-size-map*[*simp*]: *list-size f* (*map g xs*) = *list-size* (*f o g*) *xs*
by (*induct xs*) *auto*

lemma *list-size-pointwise*[*termination-simp*]:
(⋀*x*. *x* ∈ *set xs* ⟹ *f x* < *g x*) ⟹ *list-size f xs* ≤ *list-size g xs*
by (*induct xs*) *force+*

40.7 Code generator

40.7.1 Setup

types-code
list (- *list*)
attach (*term-of*) ⟪
fun term-of-list f T = *HOLogic.mk-list T o map f*;

```
⟩⟩
attach (test) ⟨⟨
fun gen-list' aG aT i j = frequency
  [(i, fn () =>
      let
        val (x, t) = aG j;
        val (xs, ts) = gen-list' aG aT (i-1) j
      in (x :: xs, fn () => HOLogic.cons-const aT $ t () $ ts ()) end),
   (1, fn () => ([], fn () => HOLogic.nil-const aT))] ()
and gen-list aG aT i = gen-list' aG aT i i;
⟩⟩
  char (string)
attach (term-of) ⟨⟨
val term-of-char = HOLogic.mk-char o ord;
⟩⟩
attach (test) ⟨⟨
fun gen-char i =
  let val j = random-range (ord a) (Int.min (ord a + i, ord z))
  in (chr j, fn () => HOLogic.mk-char j) end;
⟩⟩

consts-code Cons ((- ::/ -))

code-type list
  (SML - list)
  (OCaml - list)
  (Haskell ![-])

code-reserved SML
  list

code-reserved OCaml
  list

code-const Nil
  (SML [])
  (OCaml [])
  (Haskell [])

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', Codegen.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;
⟫
```

40.7.2 Generation of efficient code

```
primrec
  member :: 'a ⇒ 'a list ⇒ bool (infixl mem 55)
where
  x mem [] ⟷ False
  | x mem (y#ys) ⟷ (if y = x then True else x mem ys)

primrec
  null:: 'a list ⇒ bool
where
  null [] = True
  | null (x#xs) = False

primrec
  list-inter :: 'a list ⇒ 'a list ⇒ 'a list
where
  list-inter [] bs = []
  | list-inter (a#as) bs =
    (if a ∈ set bs then a # list-inter as bs else list-inter as bs)
```

primrec
list-all :: $('a \Rightarrow bool) \Rightarrow ('a\ list \Rightarrow bool)$
where
list-all P [] = *True*
| *list-all P* $(x\#xs) = (P\ x \wedge$ *list-all P xs*$)$

primrec
list-ex :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$
where
list-ex P [] = *False*
| *list-ex P* $(x\#xs) = (P\ x \vee$ *list-ex P xs*$)$

primrec
filtermap :: $('a \Rightarrow 'b\ option) \Rightarrow 'a\ list \Rightarrow 'b\ list$
where
filtermap f [] = []
| *filtermap f* $(x\#xs) =$
(*case f x of None* $\Rightarrow$ *filtermap f xs*
| *Some y* $\Rightarrow$ *y* # *filtermap f xs*)

primrec
map-filter :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'b\ list$
where
map-filter f P [] = []
| *map-filter f P* $(x\#xs) =$
(*if P x then f x* # *map-filter f P xs else map-filter f P xs*)

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* $(\lambda xs\ x.\ x\ \#\ xs)$ [] *xs*
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case *Cons*
{
fix *x xs ys*
have *foldl* $(\lambda xs\ x.\ x\ \#\ xs)$ *ys xs* @ [*x*]
= *foldl* $(\lambda xs\ x.\ x\ \#\ xs)$ (*ys* @ [*x*]) *xs*
by (*induct xs arbitrary*: *ys*) *auto*
}

note *aux* = *this*
show *?case* **by** (*induct xs*) (*auto simp add*: *Cons aux*)
qed

lemma *mem-iff* [*code post*]:
x mem xs ⟷ *x* ∈ *set xs*
by (*induct xs*) *auto*

lemmas *in-set-code* [*code unfold*] = *mem-iff* [*symmetric*]

lemma *empty-null* [*code inline*]:
xs = [] ⟷ *null xs*
by (*cases xs*) *simp-all*

lemmas *null-empty* [*code post*] =
empty-null [*symmetric*]

lemma *list-inter-conv*:
set (*list-inter xs ys*) = *set xs* ∩ *set ys*
by (*induct xs*) *auto*

lemma *list-all-iff* [*code post*]:
list-all P xs ⟷ (∀ *x* ∈ *set xs*. *P x*)
by (*induct xs*) *auto*

lemmas *list-ball-code* [*code unfold*] = *list-all-iff* [*symmetric*]

lemma *list-all-append* [*simp*]:
list-all P (*xs* @ *ys*) ⟷ (*list-all P xs* ∧ *list-all P ys*)
by (*induct xs*) *auto*

lemma *list-all-rev* [*simp*]:
list-all P (*rev xs*) ⟷ *list-all P xs*
by (*simp add*: *list-all-iff*)

lemma *list-all-length*:
list-all P xs ⟷ (∀ *n* < *length xs*. *P* (*xs* ! *n*))
unfolding *list-all-iff* **by** (*auto intro*: *all-nth-imp-all-set*)

lemma *list-ex-iff* [*code post*]:
list-ex P xs ⟷ (∃ *x* ∈ *set xs*. *P x*)
by (*induct xs*) *simp-all*

lemmas *list-bex-code* [*code unfold*] =
list-ex-iff [*symmetric*]

lemma *list-ex-length*:
list-ex P xs ⟷ (∃ *n* < *length xs*. *P* (*xs* ! *n*))
unfolding *list-ex-iff set-conv-nth* **by** *auto*

lemma *filtermap-conv*:
filtermap f xs = map (λ*x*. *the* (*f x*)) (*filter* (λ*x*. *f x* ≠ *None*) *xs*)
by (*induct xs*) (*simp-all split*: *option.split*)

lemma *map-filter-conv* [*simp*]:
map-filter f P xs = map f (*filter P xs*)
by (*induct xs*) *auto*

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*]:
(∀ *m*<*n*::*nat*. *P m*) ⟷ (∀ *m* ∈ {*0*..<*n*}. *P m*)
by *auto*

lemma *ex-nat-less-eq* [*code unfold*]:
(∃ *m*<*n*::*nat*. *P m*) ⟷ (∃ *m* ∈ {*0*..<*n*}. *P m*)
by *auto*

lemma *all-nat-less* [*code unfold*]:
(∀ *m*≤*n*::*nat*. *P m*) ⟷ (∀ *m* ∈ {*0*..*n*}. *P m*)
by *auto*

lemma *ex-nat-less* [*code unfold*]:
(∃ *m*≤*n*::*nat*. *P m*) ⟷ (∃ *m* ∈ {*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*

end

41 Map: Maps

theory *Map*
imports *List*
begin

types ('*a*,'*b*) "=> = '*a* => '*b option* (**infixr** *0*)
translations (*type*) *a* "=> *b* <= (*type*) *a* => *b option*

syntax (*xsymbols*)
"=> :: [*type*, *type*] => *type* (**infixr** ⇀ *0*)

abbreviation
empty :: '*a* "=> '*b* **where**
empty == %*x*. *None*

definition
map-comp :: ('*b* "=> '*c*) => ('*a* "=> '*b*) => ('*a* "=> '*c*) (**infixl** *o'-m 55*)
where
f o-m g = (λ*k*. *case g k of None* ⇒ *None* | *Some v* ⇒ *f v*)

notation (*xsymbols*)
map-comp (**infixl** ∘$_m$ *55*)

definition
map-add :: ('*a* "=> '*b*) => ('*a* "=> '*b*) => ('*a* "=> '*b*) (**infixl** ++ *100*)
where
m1 ++ *m2* = (λ*x*. *case m2 x of None => m1 x* | *Some y => Some y*)

definition
restrict-map :: ('*a* "=> '*b*) => '*a set* => ('*a* "=> '*b*) (**infixl** |‘ *110*) **where**
m|‘*A* = (λ*x*. *if x* : *A then m x else None*)

notation (*latex* **output**)
restrict-map (-|‘_ [*111*,*110*] *110*)

definition
dom :: ('*a* "=> '*b*) => '*a set* **where**
dom m = {*a*. *m a* "= *None*}

definition
ran :: ('*a* ~=> '*b*) => '*b set* **where**
ran m = {*b*. *EX a*. *m a* = *Some b*}

definition
map-le :: ('*a* ~=> '*b*) => ('*a* ~=> '*b*) => *bool* (**infix** $\subseteq_m$ *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* (- /$\mapsto$/ -)
-maplets :: ['*a*, '*a*] => *maplet* (- /[$\mapsto$]/ -)

translations
-MapUpd m (*-Maplets xy ms*) == *-MapUpd* (*-MapUpd m xy*) *ms*
-MapUpd m (*-maplet x y*) == *m*(*x*:=*Some y*)
-MapUpd m (*-maplets x y*) == *map-upds m x y*
-Map ms == *-MapUpd* (*CONST empty*) *ms*
-Map (*-Maplets ms1 ms2*) <= *-MapUpd* (*-Map ms1*) *ms2*
-Maplets ms1 (*-Maplets ms2 ms3*) <= *-Maplets* (*-Maplets ms1 ms2*) *ms3*

primrec
map-of [] = *empty*
map-of (*p*#*ps*) = (*map-of ps*)(*fst p* |-> *snd p*)

declare *map-of.simps* [*code del*]

lemma *map-of-Cons-code* [*code*]:
map-of [] *k* = *None*
map-of ((*l*, *v*) # *ps*) *k* = (*if l* = *k then Some v else map-of ps k*)
by *simp-all*

defs
map-upds-def [*code func*]: *m*(*xs* [|->] *ys*) == *m* ++ *map-of* (*rev*(*zip xs ys*))

41.1 *empty*

lemma *empty-upd-none* [*simp*]: *empty*(*x* := *None*) = *empty*
by (*rule ext*) *simp*

41.2 *map-upd*

lemma *map-upd-triv*: *t k* = *Some x* ==> *t*(*k*|->*x*) = *t*
by (*rule ext*) *simp*

lemma *map-upd-nonempty* [*simp*]: *t*(*k*|->*x*) ~= *empty*
proof
  assume *t*(*k* ↦ *x*) = *empty*
  then have (*t*(*k* ↦ *x*)) *k* = *None* **by** *simp*
  then show *False* **by** *simp*
qed

lemma *map-upd-eqD1*:
  assumes *m*(*a*↦*x*) = *n*(*a*↦*y*)
  shows *x* = *y*
proof −
  from *prems* **have** (*m*(*a*↦*x*)) *a* = (*n*(*a*↦*y*)) *a* **by** *simp*
  then show *?thesis* **by** *simp*
qed

lemma *map-upd-Some-unfold*:
  ((*m*(*a*|->*b*)) *x* = *Some y*) = (*x* = *a* ∧ *b* = *y* ∨ *x* ≠ *a* ∧ *m x* = *Some y*)
by *auto*

lemma *image-map-upd* [*simp*]: *x* ∉ *A* ⟹ *m*(*x* ↦ *y*) ‘ *A* = *m* ‘ *A*
by *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

41.3 *map-of*

lemma *map-of-eq-None-iff*:
  (*map-of xys x* = *None*) = (*x* ∉ *fst* ‘ (*set xys*))
by (*induct xys*) *simp-all*

lemma *map-of-is-SomeD*: *map-of xys x* = *Some y* ⟹ (*x*,*y*) ∈ *set xys*
apply (*induct xys*)
 apply *simp*
apply (*clarsimp split*: *if-splits*)

done

lemma *map-of-eq-Some-iff* [*simp*]:
distinct(*map fst xys*) ⟹ (*map-of xys x* = *Some y*) = ((*x*,*y*) ∈ *set xys*)
apply (*induct xys*)
apply *simp*
apply *simp*
apply (*auto simp*: *map-of-eq-None-iff* [*symmetric*])
done

lemma *Some-eq-map-of-iff* [*simp*]:
distinct(*map fst xys*) ⟹ (*Some y* = *map-of xys x*) = ((*x*,*y*) ∈ *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*) ∈ *set xys* ⟧
⟹ *map-of xys x* = *Some y*
apply (*induct xys*)
apply *simp*
apply *simp*
apply *force*
done

lemma *map-of-zip-is-None* [*simp*]:
length xs = *length ys* ⟹ (*map-of* (*zip xs ys*) *x* = *None*) = (*x* ∉ *set xs*)
by (*induct rule*: *list-induct2*) *simp-all*

lemma *map-of-zip-is-Some*:
assumes *length xs* = *length ys*
shows *x* ∈ *set xs* ⟷ (∃ *y*. *map-of* (*zip xs ys*) *x* = *Some y*)
using *assms* **by** (*induct rule*: *list-induct2*) *simp-all*

lemma *map-of-zip-upd*:
fixes *x* :: *'a* **and** *xs* :: *'a list* **and** *ys zs* :: *'b list*
assumes *length ys* = *length xs*
and *length zs* = *length xs*
and *x* ∉ *set xs*
and *map-of* (*zip xs ys*)(*x* ↦ *y*) = *map-of* (*zip xs zs*)(*x* ↦ *z*)
shows *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*)
proof
fix *x'* :: *'a*
show *map-of* (*zip xs ys*) *x'* = *map-of* (*zip xs zs*) *x'*
proof (*cases x* = *x'*)
case *True*
from *assms True map-of-zip-is-None* [*of xs ys x'*]
have *map-of* (*zip xs ys*) *x'* = *None* **by** *simp*
moreover from *assms True map-of-zip-is-None* [*of xs zs x'*]
have *map-of* (*zip xs zs*) *x'* = *None* **by** *simp*
ultimately show *?thesis* **by** *simp*
next
case *False* **from** *assms*
have (*map-of* (*zip xs ys*)(*x* ↦ *y*)) *x'* = (*map-of* (*zip xs zs*)(*x* ↦ *z*)) *x'* **by**

auto
  with *False* **show** *?thesis* **by** *simp*
 qed
qed

lemma *map-of-zip-inject*:
 assumes *length ys = length xs*
  and *length zs = length xs*
  and *dist*: *distinct xs*
  and *map-of*: *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*)
 shows *ys = zs*
using *assms*(*1*) *assms*(*2*)[*symmetric*] **using** *dist map-of* **proof** (*induct ys xs zs rule*: *list-induct3*)
 case *Nil* **show** *?case* **by** *simp*
next
 case (*Cons y ys x xs z zs*)
 from ‹*map-of* (*zip* (*x*#*xs*) (*y*#*ys*)) = *map-of* (*zip* (*x*#*xs*) (*z*#*zs*))›
  have *map-of*: *map-of* (*zip xs ys*)(*x* ↦ *y*) = *map-of* (*zip xs zs*)(*x* ↦ *z*) **by** *simp*
 from *Cons* **have** *length ys = length xs* **and** *length zs = length xs*
  and *x* ∉ *set xs* **by** *simp-all*
 then have *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*) **using** *map-of* **by** (*rule map-of-zip-upd*)
 with *Cons.hyps* ‹*distinct* (*x* # *xs*)› **have** *ys = zs* **by** *simp*
 moreover from *map-of* **have** *y = z* **by** (*rule map-upd-eqD1*)
 ultimately show *?case* **by** *simp*
qed

lemma *finite-range-map-of*: *finite* (*range* (*map-of xys*))
apply (*induct xys*)
apply (*simp-all add*: *image-constant*)
apply (*rule finite-subset*)
prefer *2* **apply** *assumption*
apply *auto*
done

lemma *map-of-SomeD*: *map-of xs k = Some y* ⟹ (*k*, *y*) ∈ *set xs*
by (*induct xs*) (*simp*, *atomize* (*full*), *auto*)

lemma *map-of-mapk-SomeI*:
 inj f ==> *map-of t k = Some x* ==>
  map-of (*map* (*split* (%*k*. *Pair* (*f k*))) *t*) (*f k*) = *Some x*
by (*induct t*) (*auto simp add*: *inj-eq*)

lemma *weak-map-of-SomeI*: (*k*, *x*) : *set l* ==> ∃*x*. *map-of l k = Some x*
by (*induct l*) *auto*

lemma *map-of-filter-in*:
 map-of xs k = Some z ⟹ *P k z* ⟹ *map-of* (*filter* (*split P*) *xs*) *k = Some z*
by (*induct xs*) *auto*

lemma *map-of-map*: *map-of* (*map* (%(*a*,*b*). (*a*,*f b*)) *xs*) *x* = *option-map f* (*map-of xs x*)
by (*induct xs*) *auto*

41.4 *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*

41.5 *map-comp* related

lemma *map-comp-empty* [*simp*]:
m ∘$_m$ *empty* = *empty*
empty ∘$_m$ *m* = *empty*
by (*auto simp add*: *map-comp-def intro*: *ext split*: *option.splits*)

lemma *map-comp-simps* [*simp*]:
m2 k = *None* ⟹ (*m1* ∘$_m$ *m2*) *k* = *None*
m2 k = *Some k'* ⟹ (*m1* ∘$_m$ *m2*) *k* = *m1 k'*
by (*auto simp add*: *map-comp-def*)

lemma *map-comp-Some-iff*:
((*m1* ∘$_m$ *m2*) *k* = *Some v*) = (∃ *k'*. *m2 k* = *Some k'* ∧ *m1 k'* = *Some v*)
by (*auto simp add*: *map-comp-def split*: *option.splits*)

lemma *map-comp-None-iff*:
((*m1* ∘$_m$ *m2*) *k* = *None*) = (*m2 k* = *None* ∨ (∃ *k'*. *m2 k* = *Some k'* ∧ *m1 k'* = *None*))
by (*auto simp add*: *map-comp-def split*: *option.splits*)

41.6 ++

lemma *map-add-empty*[*simp*]: *m* ++ *empty* = *m*
by(*simp add*: *map-add-def*)

lemma *empty-map-add*[*simp*]: *empty* ++ *m* = *m*
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-assoc*[*simp*]: *m1* ++ (*m2* ++ *m3*) = (*m1* ++ *m2*) ++ *m3*
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-Some-iff*:
((*m* ++ *n*) *k* = *Some x*) = (*n k* = *Some x* | *n k* = *None* & *m k* = *Some x*)
by (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-SomeD* [*dest*!]:
(*m* ++ *n*) *k* = *Some x* ⟹ *n k* = *Some x* ∨ *n k* = *None* ∧ *m k* = *Some x*
by (*rule map-add-Some-iff* [*THEN iffD1*])

lemma *map-add-find-right* [*simp*]: !!*xx*. *n k* = *Some xx* ==> (*m* ++ *n*) *k* = *Some xx*
by (*subst map-add-Some-iff*) *fast*

lemma *map-add-None* [*iff*]: ((*m* ++ *n*) *k* = *None*) = (*n k* = *None* & *m k* = *None*)
by (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-upd*[*simp*]: *f* ++ *g*(*x*|−>*y*) = (*f* ++ *g*)(*x*|−>*y*)
by (*rule ext*) (*simp add*: *map-add-def*)

lemma *map-add-upds*[*simp*]: *m1* ++ (*m2*(*xs*[↦]*ys*)) = (*m1*++*m2*)(*xs*[↦]*ys*)
by (*simp add*: *map-upds-def*)

lemma *map-of-append*[*simp*]: *map-of* (*xs* @ *ys*) = *map-of ys* ++ *map-of xs*
unfolding *map-add-def*
apply (*induct xs*)
apply *simp*
apply (*rule ext*)
apply (*simp split add*: *option.split*)
done

lemma *finite-range-map-of-map-add*:
finite (*range f*) ==> *finite* (*range* (*f* ++ *map-of l*))
apply (*induct l*)
apply (*auto simp del*: *fun-upd-apply*)
apply (*erule finite-range-updI*)
done

lemma *inj-on-map-add-dom* [*iff*]:
inj-on (*m* ++ *m'*) (*dom m'*) = *inj-on m'* (*dom m'*)
by (*fastsimp simp*: *map-add-def dom-def inj-on-def split*: *option.splits*)

41.7 *restrict-map*

lemma *restrict-map-to-empty* [*simp*]: *m*|‘{} = *empty*
by (*simp add*: *restrict-map-def*)

lemma *restrict-map-empty* [*simp*]: *empty*|‘*D* = *empty*
by (*simp add*: *restrict-map-def*)

lemma *restrict-in* [*simp*]: *x* ∈ *A* ⟹ (*m*|‘*A*) *x* = *m x*
by (*simp add*: *restrict-map-def*)

lemma *restrict-out* [*simp*]: *x* ∉ *A* ⟹ (*m*|‘*A*) *x* = *None*

by (*simp add*: *restrict-map-def*)

lemma *ran-restrictD*: $y \in ran\ (m|`A) \Longrightarrow \exists x{\in}A.\ m\ x = Some\ y$
by (*auto simp*: *restrict-map-def ran-def split*: *split-if-asm*)

lemma *dom-restrict* [*simp*]: $dom\ (m|`A) = dom\ m \cap A$
by (*auto simp*: *restrict-map-def dom-def split*: *split-if-asm*)

lemma *restrict-upd-same* [*simp*]: $m(x{\mapsto}y)|`(-\{x\}) = m|`(-\{x\})$
by (*rule ext*) (*auto simp*: *restrict-map-def*)

lemma *restrict-restrict* [*simp*]: $m|`A|`B = m|`(A{\cap}B)$
by (*rule ext*) (*auto simp*: *restrict-map-def*)

lemma *restrict-fun-upd* [*simp*]:
$m(x := y)|`D = (if\ x \in D\ then\ (m|`(D{-}\{x\}))(x := y)\ else\ m|`D)$
by (*simp add*: *restrict-map-def expand-fun-eq*)

lemma *fun-upd-None-restrict* [*simp*]:
$(m|`D)(x := None) = (if\ x{:}D\ then\ m|`(D - \{x\})\ else\ m|`D)$
by (*simp add*: *restrict-map-def expand-fun-eq*)

lemma *fun-upd-restrict*: $(m|`D)(x := y) = (m|`(D{-}\{x\}))(x := y)$
by (*simp add*: *restrict-map-def expand-fun-eq*)

lemma *fun-upd-restrict-conv* [*simp*]:
$x \in D \Longrightarrow (m|`D)(x := y) = (m|`(D{-}\{x\}))(x := y)$
by (*simp add*: *restrict-map-def expand-fun-eq*)

41.8 *map-upds*

lemma *map-upds-Nil1* [*simp*]: $m([]\ [|{-}{>}]\ bs) = m$
by (*simp add*: *map-upds-def*)

lemma *map-upds-Nil2* [*simp*]: $m(as\ [|{-}{>}]\ []) = m$
by (*simp add*:*map-upds-def*)

lemma *map-upds-Cons* [*simp*]: $m(a\#as\ [|{-}{>}]\ b\#bs) = (m(a|{-}{>}b))(as[|{-}{>}]bs)$
by (*simp add*:*map-upds-def*)

lemma *map-upds-append1* [*simp*]: $\bigwedge ys\ m.\ size\ xs < size\ ys \Longrightarrow$
$m(xs@[x]\ [\mapsto]\ ys) = m(xs\ [\mapsto]\ ys)(x \mapsto ys!size\ xs)$
apply(*induct xs*)
apply (*clarsimp simp add*: *neq-Nil-conv*)
apply (*case-tac ys*)
apply *simp*
apply *simp*
done

lemma *map-upds-list-update2-drop* [*simp*]:
  ⟦*size xs* ≤ *i*; *i* < *size ys*⟧
    ⟹ *m*(*xs*[↦]*ys*[*i*:=*y*]) = *m*(*xs*[↦]*ys*)
apply (*induct xs arbitrary*: *m ys i*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp*
apply (*simp split*: *nat.split*)
done

lemma *map-upd-upds-conv-if*:
  (*f*(*x*|−>*y*))(*xs* [|−>] *ys*) =
   (*if x* : *set*(*take* (*length ys*) *xs*) *then f*(*xs* [|−>] *ys*)
                                else (*f*(*xs* [|−>] *ys*))(*x*|−>*y*))
apply (*induct xs arbitrary*: *x y ys f*)
 apply *simp*
apply (*case-tac ys*)
 apply (*auto split*: *split-if simp*: *fun-upd-twist*)
done

lemma *map-upds-twist* [*simp*]:
  a ~: *set as* ==> *m*(*a*|−>*b*)(*as*[|−>]*bs*) = *m*(*as*[|−>]*bs*)(*a*|−>*b*)
using *set-take-subset* **by** (*fastsimp simp add*: *map-upd-upds-conv-if*)

lemma *map-upds-apply-nontin* [*simp*]:
  x ~: *set xs* ==> (*f*(*xs*[|−>]*ys*)) *x* = *f x*
apply (*induct xs arbitrary*: *ys*)
 apply *simp*
apply (*case-tac ys*)
 apply (*auto simp*: *map-upd-upds-conv-if*)
done

lemma *fun-upds-append-drop* [*simp*]:
  size xs = *size ys* ⟹ *m*(*xs*@*zs*[↦]*ys*) = *m*(*xs*[↦]*ys*)
apply (*induct xs arbitrary*: *m ys*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp-all*
done

lemma *fun-upds-append2-drop* [*simp*]:
  size xs = *size ys* ⟹ *m*(*xs*[↦]*ys*@*zs*) = *m*(*xs*[↦]*ys*)
apply (*induct xs arbitrary*: *m ys*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp-all*
done

lemma *restrict-map-upds*[*simp*]:
  ⟦ *length xs* = *length ys*; *set xs* ⊆ *D* ⟧
    ⟹ *m*(*xs* [↦] *ys*)|`*D* = (*m*|`(*D* − *set xs*))(*xs* [↦] *ys*)
apply (*induct xs arbitrary*: *m ys*)
 apply *simp*
apply (*case-tac ys*)
 apply *simp*
apply (*simp add*: *Diff-insert* [*symmetric*] *insert-absorb*)
apply (*simp add*: *map-upd-upds-conv-if*)
done

41.9 *dom*

lemma *domI*: *m a* = *Some b* ==> *a* : *dom m*
by(*simp add*:*dom-def*)

lemma *domD*: *a* : *dom m* ==> ∃ *b*. *m a* = *Some b*
by (*cases m a*) (*auto simp add*: *dom-def*)

lemma *domIff* [*iff*, *simp del*]: (*a* : *dom m*) = (*m a* ~= *None*)
by(*simp add*:*dom-def*)

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

lemma *dom-fun-upd* [*simp*]:
  dom(*f*(*x* := *y*)) = (*if y*=*None then dom f* − {*x*} *else insert x* (*dom f*))
by(*auto simp add*:*dom-def*)

lemma *dom-map-of*: *dom*(*map-of xys*) = {*x*. ∃ *y*. (*x*,*y*) : *set xys*}
by (*induct xys*) (*auto simp del*: *fun-upd-apply*)

lemma *dom-map-of-conv-image-fst*:
  dom(*map-of xys*) = *fst* ` (*set xys*)
by(*force simp*: *dom-map-of*)

lemma *dom-map-of-zip* [*simp*]: [| *length xs* = *length ys*; *distinct xs* |] ==>
  dom(*map-of*(*zip xs ys*)) = *set xs*
by (*induct rule*: *list-induct2*) *simp-all*

lemma *finite-dom-map-of*: *finite* (*dom* (*map-of l*))
by (*induct l*) (*auto simp add*: *dom-def insert-Collect* [*symmetric*])

lemma *dom-map-upds* [*simp*]:
  dom(*m*(*xs*[|->]*ys*)) = *set*(*take* (*length ys*) *xs*) *Un dom m*
apply (*induct xs arbitrary*: *m ys*)
 apply *simp*
apply (*case-tac ys*)

apply *auto*
done

lemma *dom-map-add* [*simp*]: *dom*(*m*++*n*) = *dom n Un dom m*
by(*auto simp*:*dom-def*)

lemma *dom-override-on* [*simp*]:
dom(*override-on f g A*) =
(*dom f* − {*a*. *a* : *A* − *dom g*}) *Un* {*a*. *a* : *A Int dom g*}
by(*auto simp*: *dom-def override-on-def*)

lemma *map-add-comm*: *dom m1* ∩ *dom m2* = {} ⟹ *m1*++*m2* = *m2*++*m1*
by (*rule ext*) (*force simp*: *map-add-def dom-def split*: *option.split*)

lemma *finite-map-freshness*:
finite (*dom* (*f* :: 'a ⇀ 'b)) ⟹ ¬ *finite* (*UNIV* :: 'a *set*) ⟹
∃ *x*. *f x* = *None*
by(*bestsimp dest*:*ex-new-if-finite*)

41.10 *ran*

lemma *ranI*: *m a* = *Some b* ==> *b* : *ran m*
by(*auto simp*: *ran-def*)

lemma *ran-empty* [*simp*]: *ran empty* = {}
by(*auto simp*: *ran-def*)

lemma *ran-map-upd* [*simp*]: *m a* = *None* ==> *ran*(*m*(*a*|->*b*)) = *insert b* (*ran m*)
unfolding *ran-def*
apply *auto*
apply (*subgoal-tac aa* ~= *a*)
apply *auto*
done

41.11 *map-le*

lemma *map-le-empty* [*simp*]: *empty* ⊆$_m$ *g*
by (*simp add*: *map-le-def*)

lemma *upd-None-map-le* [*simp*]: *f*(*x* := *None*) ⊆$_m$ *f*
by (*force simp add*: *map-le-def*)

lemma *map-le-upd*[*simp*]: *f* ⊆$_m$ *g* ==> *f*(*a* := *b*) ⊆$_m$ *g*(*a* := *b*)
by (*fastsimp simp add*: *map-le-def*)

lemma *map-le-imp-upd-le* [*simp*]: *m1* ⊆$_m$ *m2* ⟹ *m1*(*x* := *None*) ⊆$_m$ *m2*(*x* ↦ *y*)

by (*force simp add*: *map-le-def*)

lemma *map-le-upds* [*simp*]:
  $f \subseteq_m g$ ==> $f(as\ [|->]\ bs) \subseteq_m g(as\ [|->]\ bs)$
apply (*induct as arbitrary*: *f g bs*)
 apply *simp*
apply (*case-tac bs*)
 apply *auto*
done

lemma *map-le-implies-dom-le*: $(f \subseteq_m g) \Longrightarrow (dom\ f \subseteq dom\ g)$
by (*fastsimp simp add*: *map-le-def dom-def*)

lemma *map-le-refl* [*simp*]: $f \subseteq_m f$
by (*simp add*: *map-le-def*)

lemma *map-le-trans*[*trans*]: $[\![\ m1 \subseteq_m m2;\ m2 \subseteq_m m3\ ]\!] \Longrightarrow m1 \subseteq_m m3$
by (*auto simp add*: *map-le-def dom-def*)

lemma *map-le-antisym*: $[\![\ f \subseteq_m g;\ g \subseteq_m f\ ]\!] \Longrightarrow f = g$
unfolding *map-le-def*
apply (*rule ext*)
apply (*case-tac* $x \in dom\ f$, *simp*)
apply (*case-tac* $x \in dom\ g$, *simp*, *fastsimp*)
done

lemma *map-le-map-add* [*simp*]: $f \subseteq_m (g\ ++\ f)$
by (*fastsimp simp add*: *map-le-def*)

lemma *map-le-iff-map-add-commute*: $(f \subseteq_m f\ ++\ g) = (f++g = g++f)$
by(*fastsimp simp*: *map-add-def map-le-def expand-fun-eq split*: *option.splits*)

lemma *map-add-le-mapE*: $f++g \subseteq_m h \Longrightarrow g \subseteq_m h$
by (*fastsimp simp add*: *map-le-def map-add-def dom-def*)

lemma *map-add-le-mapI*: $[\![\ f \subseteq_m h;\ g \subseteq_m h;\ f \subseteq_m f++g\ ]\!] \Longrightarrow f++g \subseteq_m h$
by (*clarsimp simp add*: *map-le-def map-add-def dom-def split*: *option.splits*)

end

42 Main: Main HOL

theory *Main*
imports *Map*
begin

ML ⟪ *val HOL-proofs* = ! *Proofterm.proofs* ⟫

ML ⟪ *path-add ~~/src/HOL/Library* ⟫

end

References