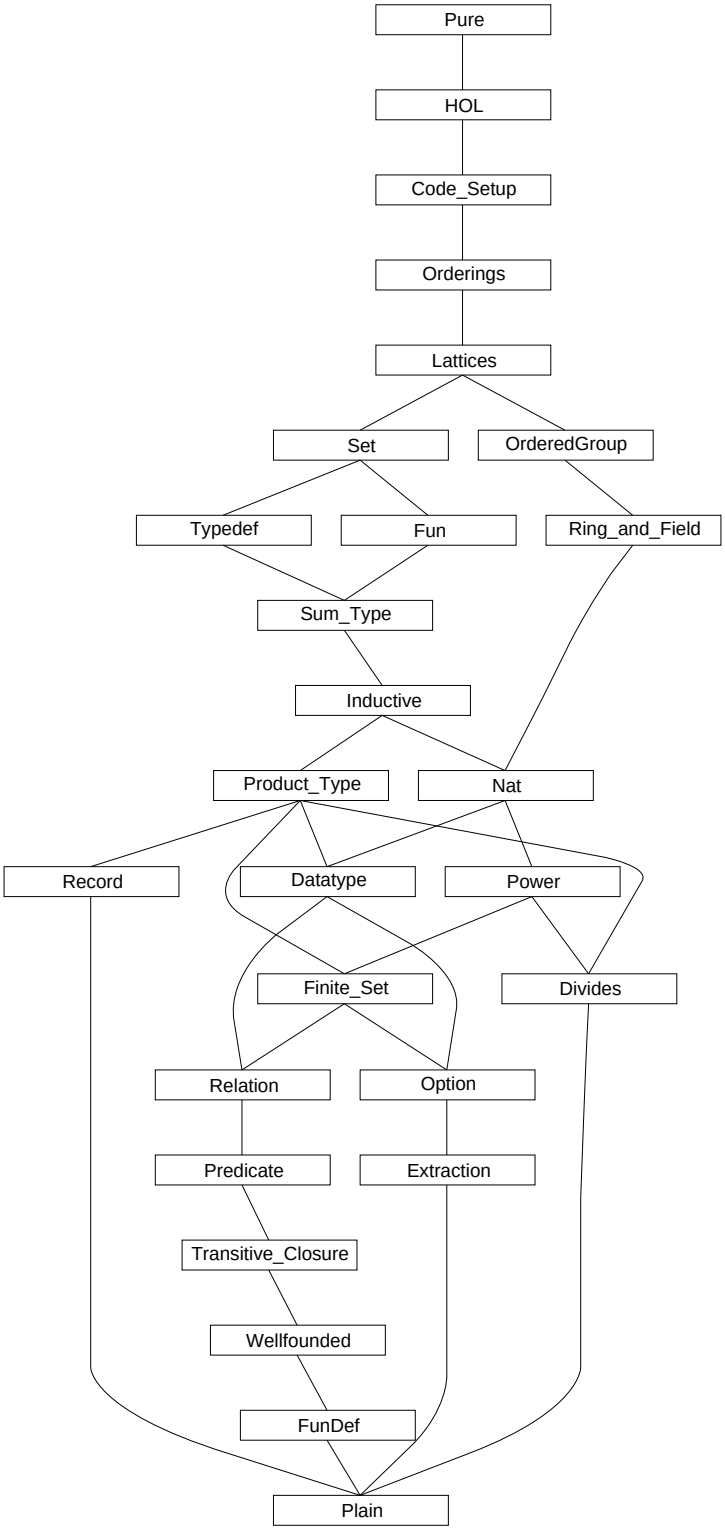
Isabelle/HOL — Higher-Order Logic

April 19, 2009

Contents


Pure
HOL
Code_Setup
Orderings
Lattices
Set
OrderedGroup
Typedef
Fun
Ring_and_Field
Sum_Type
Inductive
Product_Type
Nat
Record
Datatype
Power
Finite_Set
Divides
Relation
Option
Predicate
Extraction
Transitive_Closure
Wellfounded
FunDef
Plain

1 HOL: The basis of Higher-Order Logic

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

setup ⟪ *Intuitionistic.method-setup iprover* ⟫

1.1 Primitive logic

1.1.1 Core syntax

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

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

itself :: (*type*) *type*

global

typedecl *bool*

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

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

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

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

local

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

1.1.2 Additional concrete syntax

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

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

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

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

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

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

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

nonterminals
letbinds letbind
case-syn cases-syn

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

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

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

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

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

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

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

Ex1 (**binder** ∃! *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
refl: $t = (t::'a)$
subst: $s = t \Longrightarrow P\ s \Longrightarrow P\ t$
ext: $(!!x::'a.\ (f\ x ::'b) = g\ x) ==> (\%x.\ f\ x) = (\%x.\ g\ x)$
— Extensionality is built into the meta-logic, and this rule expresses a related property. It is an eta-expanded version of the traditional rule, and similar to the ABS rule of HOL

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

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

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

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

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

finalconsts
op =

op −−>
The

axiomatization
undefined :: *'a*

abbreviation (*input*)
arbitrary ≡ *undefined*

1.1.4 Generic classes and algebraic operations

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

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

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

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

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

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

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

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

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

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

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

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

end

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

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

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

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

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

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

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

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

end

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

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

⟩⟩ — show types that are presumably too general

1.2 Fundamental rules

1.2.1 Equality

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

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

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

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

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

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

For calculational reasoning:

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

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

1.2.2 Congruence rules for application

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

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

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

apply (*rule refl*)
done

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

1.2.3 Equality of booleans – iff

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

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

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

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

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

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

1.2.4 True

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

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

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

1.2.5 Universal quantifier

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

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

apply (*erule fun-cong*)
done

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

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

1.2.6 False

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

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

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

1.2.7 Negation

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

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

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

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

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

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

1.2.8 Implication

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

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

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

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

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

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

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

1.2.9 Existential quantifier

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

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

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

1.2.10 Conjunction

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

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

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

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

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

1.2.11 Disjunction

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

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

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

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

1.2.12 Classical logic

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

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

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

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

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

1.2.13 Unique existence

lemma *ex1I*:

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

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

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

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

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

1.2.14 THE: definite description operator

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

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

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

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

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

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

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

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

1.2.15 Classical intro rules for disjunction and existential quantifiers

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

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

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

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

and *prem2*: $\sim P \Longrightarrow Q$
shows Q
apply (*rule excluded-middle* [*THEN disjE*])
apply (*erule prem2*)
apply (*erule prem1*)
done

lemma *impCE*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $\sim P \Longrightarrow R$ $Q \Longrightarrow R$
shows R
apply (*rule excluded-middle* [*of P*, *THEN disjE*])
apply (*iprover intro*: *minor major* [*THEN mp*])+
done

lemma *impCE'*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $Q \Longrightarrow R$ $\sim P \Longrightarrow R$
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*: $[\![P; Q]\!] \Longrightarrow R$ $[\![\sim P; \sim Q]\!] \Longrightarrow R$
shows R
apply (*rule major* [*THEN iffE*])
apply (*iprover intro*: *minor elim*: *impCE notE*)
done

lemma *exCI*:
assumes *ALL x*. $\sim P(x) \Longrightarrow P(a)$
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*: $P \longrightarrow Q$
and *2*: $Q \Longrightarrow R$
and *3*: $P \longrightarrow Q \Longrightarrow P$
shows R
proof −

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

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

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

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

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

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

use *Tools/hologic.ML*

1.2.17 Atomizing meta-level connectives

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

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

qed

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

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

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

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

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

1.2.18 Atomizing elimination rules

setup *AtomizeElim.setup*

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

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

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

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

1.3 Package setup

1.3.1 Classical Reasoner setup

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

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

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

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

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

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

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

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

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

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

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

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

and *refl* [*intro*!]

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

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

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

ML ⟪ *val HOL-cs* = @{*claset*} ⟫

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

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

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

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

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

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

setup *Blast.setup*

1.3.2 Simplifier

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemma *cases-simp*: $((P --> Q) \,\&\, (\sim P --> Q)) = Q$
— 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*: $((!\ x.\ P\ x) --> Q) = (?\ x.\ P\ x --> Q)$ **by** *blast*
lemma *not-ex*: $(\sim (?\ x.\ P(x))) = (!\ x.\sim P(x))$ **by** *iprover*
lemma *imp-ex*: $((?\ x.\ P\ x) --> Q) = (!\ x.\ P\ x --> Q)$ **by** *iprover*
lemma *all-not-ex*: $(ALL\ x.\ P\ x) = (\sim (EX\ x.\ \sim\ P\ x\ ))$ **by** *blast*

declare *All-def* [*noatp*]

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

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

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

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

The | congruence rule: not included by default!

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

by *blast*

if-then-else rules

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

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

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

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

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

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

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

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

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

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

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

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

let rules for simproc

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
in proc end;
⟩⟩

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

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

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

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

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

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

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

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

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

ML ⟨⟨ *val HOL-ss* = @{*simpset*} ⟩⟩

Simplifies x assuming c and y assuming c

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

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

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

Prevents simplification of t: much faster

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

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

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

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

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

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

1.3.3 Generic cases and induction

Rule projections:

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

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

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

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

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

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

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

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

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

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

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

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

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

Method setup.

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

setup *Induct.setup*

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

1.3.4 Coherent logic

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

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

setup *Coherent.setup*

1.4 Other simple lemmas and lemma duplicates

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

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

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

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

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

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

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

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

1.5 Basic ML bindings

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

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

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

Equality

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

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

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

end

Module setup

use *Tools/recfun-codegen.ML*

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

1.7 Nitpick hooks

This will be relocated once Nitpick is moved to HOL.

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

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

1.8 Legacy tactics and ML bindings

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

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

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

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

⟩⟩

end

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

theory *Code-Setup*
imports *HOL*
begin

2.1 Generic code generator foundation

Datatypes

code-datatype *True False*

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

code-datatype *Trueprop prop*

Code equations

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

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

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

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

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

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

Equality

context *eq*
begin

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

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

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

end

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

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

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

⟩⟩

Cases

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

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

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

code-abort *undefined*

2.2 Generic code generator preprocessor

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

2.3 Generic code generator target languages

type bool

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

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

code-reserved *SML*
  bool true false not

code-reserved *OCaml*
bool not

using built-in Haskell equality

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

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

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

undefined

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

2.4 SML code generator setup

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

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

setup ⟪
let

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

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

2.5 Evaluation and normalization by evaluation

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

ML ⟨⟨
structure Eval-Method =
struct

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

end;
⟩⟩

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

```
    | NONE => dummy
  end
⟩⟩

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

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

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

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

2.6 Quickcheck

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

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

end
```

3 Orderings: Abstract orderings

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

3.1 Quasi orders

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

Reflexivity.

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

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

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

Asymmetry.

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

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

Transitivity.

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

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

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

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

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

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

Transitivity rules for calculational reasoning

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

Dual order

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

end

3.2 Partial orders

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

Reflexivity.

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

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

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

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

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

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

Transitivity rules for calculational reasoning

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

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

Asymmetry.

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

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

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

Least value operator

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

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

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

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

Dual order

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

end

3.3 Linear (total) orders

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Dual order

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

min/max

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

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

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

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

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

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

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

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

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

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

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

end

Explicit dictionaries for code generation

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

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

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

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

3.4 Reasoning tools setup

ML ⟪

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

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

structure Orders: ORDERS =
struct

(** Theory and context data **)

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

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

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


(** Method **)

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

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

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


(** Attribute **)

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

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


(** Diagnostic command **)
```

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


(** Setup **)

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

end;

⟫
```

setup *Orders.setup*

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

context *order*
begin

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

context *linorder*
begin

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

end

setup ⟪
let

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

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

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

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

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

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

3.5 Name duplicates

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

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

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

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

3.6 Bounded quantifiers

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.7 Transitivity reasoning

context *ord*
begin

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

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

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

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

end

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

lemmas [*trans*] =
trans

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

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

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

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

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

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

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

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

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

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

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

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

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

3.8 Monotonicity, least value operator and min/max

context *order*
begin

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

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

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

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

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

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

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

end

context *linorder*
begin

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

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

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

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

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

end

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

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

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

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

3.9 Top and bottom elements

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

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

3.10 Dense orders

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

3.11 Wellorders

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

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

qed

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

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

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

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

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

end

3.12 Order on bool

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

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

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

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

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

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

end

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

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

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

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

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

3.13 Order on functions

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

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

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

instance ..

end

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

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

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

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

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

end

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

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

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

end

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

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

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

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

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

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

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

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

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

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

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

end

4 Lattices: Abstract lattices

theory *Lattices*
imports *Orderings*
begin

4.1 Lattices

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

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

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

Dual lattice

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

end

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

4.1.1 Intro and elim rules

context *lower-semilattice*
begin

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

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

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

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

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

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

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

end

context *upper-semilattice*
begin

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

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

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

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

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

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

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

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

end

4.1.2 Equational laws

context *lower-semilattice*
begin

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

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

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

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

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

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

by (*blast intro*: *antisym*)

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

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

end

context *upper-semilattice*
begin

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

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

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

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

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

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

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

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

end

context *lattice*
begin

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

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

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

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

Towards distributivity

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

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

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

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

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

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

end

4.2 Distributive lattices

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

context *distrib-lattice*
begin

lemma *sup-inf-distrib2*:

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

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

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

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

end

4.3 Uniqueness of inf and sup

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

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

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

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

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

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

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

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

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

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

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

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

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

4.5 Bool as lattice

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

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

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

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

end

4.6 Fun as lattice

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

begin

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

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

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

end

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

redundant bindings

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

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

end

5 Set: Set theory for higher-order logic

theory *Set*
imports *Lattices*
begin

A set in HOL is simply a predicate.

5.1 Basic syntax

global

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

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

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

local

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

translations

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

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

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

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

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

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

5.2 Additional concrete syntax

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.2.1 Bounded quantifiers

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

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

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

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

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

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

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

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

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

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

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

parse-translation ⟪

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

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

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

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

```
print-translation ⟪
let
  fun btr' syn [A, Abs abs] =
    let val (x, t) = atomic-abs-tr' abs
    in Syntax.const syn $ x $ A $ t end
in
[(@{const-syntax Ball}, btr' -Ball), (@{const-syntax Bex}, btr' -Bex),
 (@{const-syntax UNION}, btr' @UNION),(@{const-syntax INTER}, btr' @INTER)]
end
⟫

print-translation ⟪
let
  val ex-tr' = snd (mk-binder-tr' (Ex, DUMMY));

  fun setcompr-tr' [Abs (abs as (-, -, P))] =
    let
      fun check (Const (Ex, -) $ Abs (-, -, P), n) = check (P, n + 1)
        | check (Const (op &, -) $ (Const (op =, -) $ Bound m $ e) $ P, n) =
            n > 0 andalso m = n andalso not (loose-bvar1 (P, n)) andalso
            ((0 upto (n − 1)) subset add-loose-bnos (e, 0, []))
        | check - = false

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

5.3 Rules and definitions

Isomorphisms between predicates and sets.

```
defs
  mem-def [code]: x : S == S x
  Collect-def [code]: Collect P == P

defs
```

Ball-def: $Ball\ A\ P \quad == ALL\ x.\ x{:}A\ --{>}\ P(x)$
Bex-def: $Bex\ A\ P \quad == EX\ x.\ x{:}A\ \&\ P(x)$
Bex1-def: $Bex1\ A\ P \quad == EX!\ x.\ x{:}A\ \&\ P(x)$

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

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

instance ..

end

instantiation *bool* :: *minus*
begin

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

instance ..

end

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

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

instance ..

end

instantiation *bool* :: *uminus*
begin

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

instance ..

end

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

5.4 Lemmas and proof tool setup

5.4.1 Relating predicates and sets

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

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

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

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

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

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

5.4.2 Bounded quantifiers

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

lemmas *strip* = *impI allI ballI*

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

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

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

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

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

Gives better instantiation for bound:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4.3 Congruence rules

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

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

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

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

5.4.4 Subsets

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

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

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

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

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

ML ⟪

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

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

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

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

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

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

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

5.4.5 Equality

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

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

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

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

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

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

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

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

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

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

5.4.6 The universal set – UNIV

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

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

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

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

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

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

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

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

5.4.7 The empty set

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

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

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

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

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

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

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

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

5.4.8 The Powerset operator – Pow

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

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

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

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

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

5.4.9 Set complement

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

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

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

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

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

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

5.4.10 Binary union – Un

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

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

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

Classical introduction rule: no commitment to A vs B.

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

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

5.4.11 Binary intersection – Int

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

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

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

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

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

5.4.12 Set difference

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

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

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

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

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

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

lemma *Compl-eq-Diff-UNIV*: $-A = (UNIV - A)$
by *blast*

5.4.13 Augmenting a set − insert

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

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

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

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

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

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

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

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

5.4.14 Singletons, using insert

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

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

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

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

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

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

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

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

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

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

lemma *diff-single-insert*: $A - \{x\} \subseteq B \Longrightarrow x \in A \Longrightarrow A \subseteq \mathit{insert}\ x\ B$
by *blast*

lemma *doubleton-eq-iff*: $(\{a,b\} = \{c,d\}) = (a{=}c \ \&\ b{=}d \mid a{=}d \ \&\ b{=}c)$
by (*blast elim*: *equalityE*)

5.4.15 Unions of families

$\mathit{UN}\ x{:}A.\ B\ x$ 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*)

lemma *strong-UN-cong*:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow_{simp} C\ x = D\ x) \Longrightarrow (UN\ x{:}A.\ C\ x) = (UN\ x{:}B.\ D\ x)$
by (*simp add*: *UNION-def simp-implies-def*)

5.4.16 Intersections of families

$INT\ x{:}A.\ B\ x$ is $\bigcap B\ `\ A$.

lemma *INT-iff* [*simp*]: $(b{:}\ (INT\ x{:}A.\ B\ x)) = (ALL\ x{:}A.\ b{:}\ B\ x)$
by (*unfold INTER-def*) *blast*

lemma *INT-I* [*intro*!]: $(!!x.\ x{:}A \Longrightarrow b{:}\ B\ x) \Longrightarrow b : (INT\ x{:}A.\ B\ x)$
by (*unfold INTER-def*) *blast*

lemma *INT-D* [*elim*]: $b : (INT\ x{:}A.\ B\ x) \Longrightarrow a{:}A \Longrightarrow b{:}\ B\ a$
by *auto*

lemma *INT-E* [*elim*]: $b : (INT\ x{:}A.\ B\ x) \Longrightarrow (b{:}\ B\ a \Longrightarrow R) \Longrightarrow (a^{\sim}{:}A \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination – by the Excluded Middle on $a \in A$.
by (*unfold INTER-def*) *blast*

lemma *INT-cong* [*cong*]:
$A = B \Longrightarrow (!!x.\ x{:}B \Longrightarrow C\ x = D\ x) \Longrightarrow (INT\ x{:}A.\ C\ x) = (INT\ x{:}B.\ D\ x)$
by (*simp add*: *INTER-def*)

5.4.17 Union

lemma *Union-iff* [*simp*,*noatp*]: $(A : Union\ C) = (EX\ X{:}C.\ A{:}X)$
by (*unfold Union-def*) *blast*

lemma *UnionI* [*intro*]: $X{:}C \Longrightarrow A{:}X \Longrightarrow A : Union\ C$
— The order of the premises presupposes that C is rigid; A may be flexible.
by *auto*

lemma *UnionE* [*elim*!]: $A : Union\ C \Longrightarrow (!!X.\ A{:}X \Longrightarrow X{:}C \Longrightarrow R) \Longrightarrow R$
by (*unfold Union-def*) *blast*

5.4.18 Inter

lemma *Inter-iff* [*simp*,*noatp*]: $(A : Inter\ C) = (ALL\ X{:}C.\ A{:}X)$
by (*unfold Inter-def*) *blast*

lemma *InterI* [*intro*!]: $(!!X.\ X{:}C \Longrightarrow A{:}X) \Longrightarrow A : Inter\ C$
by (*simp add*: *Inter-def*)

A "destruct" rule – every X in C contains A as an element, but $A \in X$ can hold when $X \in C$ does not! This rule is analogous to *spec*.

lemma *InterD* [*elim*]: $A : Inter\ C \Longrightarrow X{:}C \Longrightarrow A{:}X$
by *auto*

lemma *InterE* [*elim*]: $A : Inter\ C \Longrightarrow (X^{\sim}{:}C \Longrightarrow R) \Longrightarrow (A{:}X \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination rule – does not require proving $X \in C$.
by (*unfold Inter-def*) *blast*

Image of a set under a function. Frequently b does not have the syntactic form of $f\ x$.

declare *image-def* [*noatp*]

lemma *image-eqI* [*simp*, *intro*]: $b = f\ x \Longrightarrow x{:}A \Longrightarrow b : f`A$
by (*unfold image-def*) *blast*

lemma *imageI*: $x : A \Longrightarrow f\ x : f\ `\ A$
by (*rule image-eqI*) (*rule refl*)

lemma *rev-image-eqI*: $x{:}A \Longrightarrow b = f\ x \Longrightarrow b : f`A$
— This version's more effective when we already have the required x.
by (*unfold image-def*) *blast*

lemma *imageE* [*elim*!]:
$b : (\%x.\ f\ x)`A \Longrightarrow (!!x.\ b = f\ x \Longrightarrow x{:}A \Longrightarrow P) \Longrightarrow P$
— The eta-expansion gives variable-name preservation.
by (*unfold image-def*) *blast*

lemma *image-Un*: $f`(A\ Un\ B) = f`A\ Un\ f`B$
by *blast*

lemma *image-eq-UN*: $f`A = (UN\ x{:}A.\ \{f\ x\})$
by *blast*

lemma *image-iff*: $(z : f`A) = (EX\ x{:}A.\ z = f\ x)$

by *blast*

lemma *image-subset-iff*: $(f`A \subseteq B) = (\forall x{\in}A.\ f\ x \in B)$
— This rewrite rule would confuse users if made default.
by *blast*

lemma *subset-image-iff*: $(B \subseteq f`A) = (EX\ AA.\ AA \subseteq A\ \&\ B = f`AA)$
apply *safe*
prefer *2* **apply** *fast*
apply (*rule-tac* $x = \{a.\ a : A\ \&\ f\ a : B\}$ **in** *exI*, *fast*)
done

lemma *image-subsetI*: $(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f`A \subseteq B$
— Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.
by *blast*

Range of a function – just a translation for image!

lemma *range-eqI*: $b = f\ x \Longrightarrow b \in range\ f$
by *simp*

lemma *rangeI*: $f\ x \in range\ f$
by *simp*

lemma *rangeE* [*elim?*]: $b \in range\ (\lambda x.\ f\ x) \Longrightarrow (!!x.\ b = f\ x \Longrightarrow P) \Longrightarrow P$
by *blast*

5.4.19 Set reasoning tools

Rewrite rules for boolean case-splitting: faster than *split-if* [*split*].

lemma *split-if-eq1*: $((if\ Q\ then\ x\ else\ y) = b) = ((Q \dashrightarrow x = b)\ \&\ (\sim Q \dashrightarrow y = b))$
by (*rule split-if*)

lemma *split-if-eq2*: $(a = (if\ Q\ then\ x\ else\ y)) = ((Q \dashrightarrow a = x)\ \&\ (\sim Q \dashrightarrow a = y))$
by (*rule split-if*)

Split ifs on either side of the membership relation. Not for [*simp*] – can cause goals to blow up!

lemma *split-if-mem1*: $((if\ Q\ then\ x\ else\ y) : b) = ((Q \dashrightarrow x : b)\ \&\ (\sim Q \dashrightarrow y : b))$
by (*rule split-if*)

lemma *split-if-mem2*: $(a : (if\ Q\ then\ x\ else\ y)) = ((Q \dashrightarrow a : x)\ \&\ (\sim Q \dashrightarrow a : y))$
by (*rule split-if* [**where** $P{=}\%S.\ a : S$])

lemmas *split-ifs = if-bool-eq-conj split-if-eq1 split-if-eq2 split-if-mem1 split-if-mem2*

ML ⟪
val mksimps-pairs = [(@{const-name Ball}, @{thms bspec})] @ mksimps-pairs;
⟫
declaration ⟪ *fn - =>*
Simplifier.map-ss (fn ss => ss setmksimps (mksimps mksimps-pairs))
⟫

5.4.20 The "proper subset" relation

lemma *psubsetI* [*intro*!,*noatp*]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$
by (*unfold less-le*) *blast*

lemma *psubsetE* [*elim*!,*noatp*]:
$[|A \subset B;\ [|A \subseteq B; \sim (B \subseteq A)|] \Longrightarrow R|] \Longrightarrow R$
by (*unfold less-le*) *blast*

lemma *psubset-insert-iff*:
$(A \subset \mathit{insert}\ x\ B) = (\mathit{if}\ x \in B\ \mathit{then}\ A \subset B\ \mathit{else}\ \mathit{if}\ x \in A\ \mathit{then}\ A - \{x\} \subset B\ \mathit{else}\ A \subseteq B)$
by (*auto simp add*: *less-le subset-insert-iff*)

lemma *psubset-eq*: $(A \subset B) = (A \subseteq B \ \&\ A \neq B)$
by (*simp only*: *less-le*)

lemma *psubset-imp-subset*: $A \subset B \Longrightarrow A \subseteq B$
by (*simp add*: *psubset-eq*)

lemma *psubset-trans*: $[|\ A \subset B;\ B \subset C\ |] \Longrightarrow A \subset C$
apply (*unfold less-le*)
apply (*auto dest*: *subset-antisym*)
done

lemma *psubsetD*: $[|\ A \subset B;\ c \in A\ |] \Longrightarrow c \in B$
apply (*unfold less-le*)
apply (*auto dest*: *subsetD*)
done

lemma *psubset-subset-trans*: $A \subset B \Longrightarrow B \subseteq C \Longrightarrow A \subset C$
by (*auto simp add*: *psubset-eq*)

lemma *subset-psubset-trans*: $A \subseteq B \Longrightarrow B \subset C \Longrightarrow A \subset C$
by (*auto simp add*: *psubset-eq*)

lemma *psubset-imp-ex-mem*: $A \subset B \Longrightarrow \exists\, b.\ b \in (B - A)$
by (*unfold less-le*) *blast*

lemma *atomize-ball*:
(!!x. $x \in A$ ==> $P\ x$) == *Trueprop* ($\forall x{\in}A.\ P\ x$)
by (*simp only*: *Ball-def atomize-all atomize-imp*)

lemmas [*symmetric*, *rulify*] = *atomize-ball*
and [*symmetric*, *defn*] = *atomize-ball*

5.5 Further set-theory lemmas

5.5.1 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq$ *insert a B*
by (*rule subsetI*) (*erule insertI2*)

lemma *subset-insertI2*: $A \subseteq B \Longrightarrow A \subseteq$ *insert b B*
by *blast*

lemma *subset-insert*: $x \notin A$ ==> ($A \subseteq$ *insert x B*) = ($A \subseteq B$)
by *blast*

Big Union – least upper bound of a set.

lemma *Union-upper*: $B \in A$ ==> $B \subseteq$ *Union A*
by (*iprover intro*: *subsetI UnionI*)

lemma *Union-least*: (!!X. $X \in A$ ==> $X \subseteq C$) ==> *Union A* $\subseteq C$
by (*iprover intro*: *subsetI elim*: *UnionE dest*: *subsetD*)

General union.

lemma *UN-upper*: $a \in A$ ==> $B\ a \subseteq (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *UN-least*: (!!x. $x \in A$ ==> $B\ x \subseteq C$) ==> $(\bigcup x{\in}A.\ B\ x) \subseteq C$
by (*iprover intro*: *subsetI elim*: *UN-E dest*: *subsetD*)

Big Intersection – greatest lower bound of a set.

lemma *Inter-lower*: $B \in A$ ==> *Inter A* $\subseteq B$
by *blast*

lemma *Inter-subset*:
[| !!X. $X \in A$ ==> $X \subseteq B$; $A \sim= \{\}$ |] ==> $\bigcap A \subseteq B$
by *blast*

lemma *Inter-greatest*: (!!X. $X \in A$ ==> $C \subseteq X$) ==> $C \subseteq$ *Inter A*
by (*iprover intro*: *InterI subsetI dest*: *subsetD*)

lemma *INT-lower*: $a \in A \Longrightarrow (\bigcap x{\in}A.\ B\ x) \subseteq B\ a$
by *blast*

lemma *INT-greatest*: $(!!x.\ x \in A \Longrightarrow C \subseteq B\ x) \Longrightarrow C \subseteq (\bigcap x{\in}A.\ B\ x)$
by (*iprover intro*: *INT-I subsetI dest*: *subsetD*)

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
by *blast*

lemma *Un-upper2*: $B \subseteq A \cup B$
by *blast*

lemma *Un-least*: $A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow A \cup B \subseteq C$
by *blast*

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
by *blast*

lemma *Int-lower2*: $A \cap B \subseteq B$
by *blast*

lemma *Int-greatest*: $C \subseteq A \Longrightarrow C \subseteq B \Longrightarrow C \subseteq A \cap B$
by *blast*

Set difference.

lemma *Diff-subset*: $A - B \subseteq A$
by *blast*

lemma *Diff-subset-conv*: $(A - B \subseteq C) = (A \subseteq B \cup C)$
by *blast*

5.5.2 Equalities involving union, intersection, inclusion, etc.

{}.

lemma *Collect-const* [*simp*]: $\{s.\ P\} = (\mathit{if}\ P\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{\})$
— supersedes *Collect-False-empty*
by *auto*

lemma *subset-empty* [*simp*]: $(A \subseteq \{\}) = (A = \{\})$
by *blast*

lemma *not-psubset-empty* [*iff*]: $\neg\ (A < \{\})$
by (*unfold less-le*) *blast*

lemma *Collect-empty-eq* [*simp*]: $(\mathit{Collect}\ P = \{\}) = (\forall x.\ \neg\ P\ x)$

by *blast*

lemma *empty-Collect-eq* [*simp*]: $(\{\} = \mathit{Collect}\ P) = (\forall x.\ \neg\ P\ x)$
by *blast*

lemma *Collect-neg-eq*: $\{x.\ \neg\ P\ x\} = -\ \{x.\ P\ x\}$
by *blast*

lemma *Collect-disj-eq*: $\{x.\ P\ x \mid Q\ x\} = \{x.\ P\ x\} \cup \{x.\ Q\ x\}$
by *blast*

lemma *Collect-imp-eq*: $\{x.\ P\ x \longrightarrow Q\ x\} = -\{x.\ P\ x\} \cup \{x.\ Q\ x\}$
by *blast*

lemma *Collect-conj-eq*: $\{x.\ P\ x \ \&\ Q\ x\} = \{x.\ P\ x\} \cap \{x.\ Q\ x\}$
by *blast*

lemma *Collect-all-eq*: $\{x.\ \forall y.\ P\ x\ y\} = (\bigcap y.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-ball-eq*: $\{x.\ \forall y{\in}A.\ P\ x\ y\} = (\bigcap y{\in}A.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-ex-eq* [*noatp*]: $\{x.\ \exists y.\ P\ x\ y\} = (\bigcup y.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *Collect-bex-eq* [*noatp*]: $\{x.\ \exists y{\in}A.\ P\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ P\ x\ y\})$
by *blast*

insert.

lemma *insert-is-Un*: *insert a A* = $\{a\}$ *Un A*
— NOT SUITABLE FOR REWRITING since $\{a\}$ == *insert a* $\{\}$
by *blast*

lemma *insert-not-empty* [*simp*]: *insert a A* $\neq \{\}$
by *blast*

lemmas *empty-not-insert* = *insert-not-empty* [*symmetric*, *standard*]
declare *empty-not-insert* [*simp*]

lemma *insert-absorb*: $a \in A$ ==> *insert a A* = *A*
— [*simp*] causes recursive calls when there are nested inserts
— with *quadratic* running time
by *blast*

lemma *insert-absorb2* [*simp*]: *insert x* (*insert x A*) = *insert x A*
by *blast*

lemma *insert-commute*: *insert x* (*insert y A*) = *insert y* (*insert x A*)

by *blast*

lemma *insert-subset* [*simp*]: (*insert x A* ⊆ *B*) = (*x* ∈ *B* & *A* ⊆ *B*)
by *blast*

lemma *mk-disjoint-insert*: *a* ∈ *A* ==> ∃ *B*. *A* = *insert a B* & *a* ∉ *B*
— use new *B* rather than *A* − {*a*} to avoid infinite unfolding
apply (*rule-tac x* = *A* − {*a*} **in** *exI*, *blast*)
done

lemma *insert-Collect*: *insert a* (*Collect P*) = {*u*. *u* ≠ *a* −−> *P u*}
by *auto*

lemma *UN-insert-distrib*: *u* ∈ *A* ==> (⋃ *x*∈*A*. *insert a* (*B x*)) = *insert a* (⋃ *x*∈*A*. *B x*)
by *blast*

lemma *insert-inter-insert*[*simp*]: *insert a A* ∩ *insert a B* = *insert a* (*A* ∩ *B*)
by *blast*

lemma *insert-disjoint* [*simp*,*noatp*]:
(*insert a A* ∩ *B* = {}) = (*a* ∉ *B* ∧ *A* ∩ *B* = {})
({} = *insert a A* ∩ *B*) = (*a* ∉ *B* ∧ {} = *A* ∩ *B*)
by *auto*

lemma *disjoint-insert* [*simp*,*noatp*]:
(*B* ∩ *insert a A* = {}) = (*a* ∉ *B* ∧ *B* ∩ *A* = {})
({} = *A* ∩ *insert b B*) = (*b* ∉ *A* ∧ {} = *A* ∩ *B*)
by *auto*

image.

lemma *image-empty* [*simp*]: *f*‘{} = {}
by *blast*

lemma *image-insert* [*simp*]: *f* ‘ *insert a B* = *insert* (*f a*) (*f*‘*B*)
by *blast*

lemma *image-constant*: *x* ∈ *A* ==> (λ*x*. *c*) ‘ *A* = {*c*}
by *auto*

lemma *image-constant-conv*: (%*x*. *c*) ‘ *A* = (*if A* = {} *then* {} *else* {*c*})
by *auto*

lemma *image-image*: *f* ‘ (*g* ‘ *A*) = (λ*x*. *f* (*g x*)) ‘ *A*
by *blast*

lemma *insert-image* [*simp*]: *x* ∈ *A* ==> *insert* (*f x*) (*f*‘*A*) = *f*‘*A*
by *blast*

lemma *image-is-empty* [*iff*]: $(f`A = \{\}) = (A = \{\})$
by *blast*

lemma *image-Collect* [*noatp*]: $f ` \{x.\ P\ x\} = \{f\ x \mid x.\ P\ x\}$
— NOT suitable as a default simprule: the RHS isn't simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.
by *blast*

lemma *if-image-distrib* [*simp*]:
$(\lambda x.\ if\ P\ x\ then\ f\ x\ else\ g\ x) ` S$
$= (f ` (S \cap \{x.\ P\ x\})) \cup (g ` (S \cap \{x.\ \neg\ P\ x\}))$
by (*auto simp add*: *image-def*)

lemma *image-cong*: $M = N \Longrightarrow (!!x.\ x \in N \Longrightarrow f\ x = g\ x) \Longrightarrow f`M = g`N$
by (*simp add*: *image-def*)

range.

lemma *full-SetCompr-eq* [*noatp*]: $\{u.\ \exists x.\ u = f\ x\} = range\ f$
by *auto*

lemma *range-composition*: $range\ (\lambda x.\ f\ (g\ x)) = f`range\ g$
by (*subst image-image*, *simp*)

Int

lemma *Int-absorb* [*simp*]: $A \cap A = A$
by *blast*

lemma *Int-left-absorb*: $A \cap (A \cap B) = A \cap B$
by *blast*

lemma *Int-commute*: $A \cap B = B \cap A$
by *blast*

lemma *Int-left-commute*: $A \cap (B \cap C) = B \cap (A \cap C)$
by *blast*

lemma *Int-assoc*: $(A \cap B) \cap C = A \cap (B \cap C)$
by *blast*

lemmas *Int-ac = Int-assoc Int-left-absorb Int-commute Int-left-commute*
— Intersection is an AC-operator

lemma *Int-absorb1*: $B \subseteq A \Longrightarrow A \cap B = B$
by *blast*

lemma *Int-absorb2*: $A \subseteq B \Longrightarrow A \cap B = A$

by *blast*

lemma *Int-empty-left* [*simp*]: $\{\} \cap B = \{\}$
by *blast*

lemma *Int-empty-right* [*simp*]: $A \cap \{\} = \{\}$
by *blast*

lemma *disjoint-eq-subset-Compl*: $(A \cap B = \{\}) = (A \subseteq -B)$
by *blast*

lemma *disjoint-iff-not-equal*: $(A \cap B = \{\}) = (\forall x{\in}A.\ \forall y{\in}B.\ x \neq y)$
by *blast*

lemma *Int-UNIV-left* [*simp*]: $UNIV \cap B = B$
by *blast*

lemma *Int-UNIV-right* [*simp*]: $A \cap UNIV = A$
by *blast*

lemma *Int-eq-Inter*: $A \cap B = \bigcap\{A, B\}$
by *blast*

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by *blast*

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
by *blast*

lemma *Int-UNIV* [*simp*,*noatp*]: $(A \cap B = UNIV) = (A = UNIV\ \&\ B = UNIV)$
by *blast*

lemma *Int-subset-iff* [*simp*]: $(C \subseteq A \cap B) = (C \subseteq A\ \&\ C \subseteq B)$
by *blast*

lemma *Int-Collect*: $(x \in A \cap \{x.\ P\ x\}) = (x \in A\ \&\ P\ x)$
by *blast*

Un.

lemma *Un-absorb* [*simp*]: $A \cup A = A$
by *blast*

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
by *blast*

lemma *Un-commute*: $A \cup B = B \cup A$
by *blast*

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$

by *blast*

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
by *blast*

lemmas *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*
— Union is an AC-operator

lemma *Un-absorb1*: $A \subseteq B \Longrightarrow A \cup B = B$
by *blast*

lemma *Un-absorb2*: $B \subseteq A \Longrightarrow A \cup B = A$
by *blast*

lemma *Un-empty-left* [*simp*]: $\{\} \cup B = B$
by *blast*

lemma *Un-empty-right* [*simp*]: $A \cup \{\} = A$
by *blast*

lemma *Un-UNIV-left* [*simp*]: $UNIV \cup B = UNIV$
by *blast*

lemma *Un-UNIV-right* [*simp*]: $A \cup UNIV = UNIV$
by *blast*

lemma *Un-eq-Union*: $A \cup B = \bigcup \{A, B\}$
by *blast*

lemma *Un-insert-left* [*simp*]: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
by *blast*

lemma *Un-insert-right* [*simp*]: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
by *blast*

lemma *Int-insert-left*:
$(insert\ a\ B)\ Int\ C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$
by *auto*

lemma *Int-insert-right*:
$A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
by *auto*

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
by *blast*

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
by *blast*

lemma *Un-Int-crazy*:
$(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
by *blast*

lemma *subset-Un-eq*: $(A \subseteq B) = (A \cup B = B)$
by *blast*

lemma *Un-empty* [*iff*]: $(A \cup B = \{\}) = (A = \{\} \ \& \ B = \{\})$
by *blast*

lemma *Un-subset-iff* [*simp*]: $(A \cup B \subseteq C) = (A \subseteq C \ \& \ B \subseteq C)$
by *blast*

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
by *blast*

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
by *blast*

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap -A = \{\}$
by *blast*

lemma *Compl-disjoint2* [*simp*]: $-A \cap A = \{\}$
by *blast*

lemma *Compl-partition*: $A \cup -A = UNIV$
by *blast*

lemma *Compl-partition2*: $-A \cup A = UNIV$
by *blast*

lemma *double-complement* [*simp*]: $-(-A) = (A::'a\ set)$
by *blast*

lemma *Compl-Un* [*simp*]: $-(A \cup B) = (-A) \cap (-B)$
by *blast*

lemma *Compl-Int* [*simp*]: $-(A \cap B) = (-A) \cup (-B)$
by *blast*

lemma *Compl-UN* [*simp*]: $-(\bigcup x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ -B\ x)$
by *blast*

lemma *Compl-INT* [*simp*]: $-(\bigcap x{\in}A.\ B\ x) = (\bigcup x{\in}A.\ -B\ x)$
by *blast*

lemma *subset-Compl-self-eq*: $(A \subseteq -A) = (A = \{\})$
by *blast*

lemma *Un-Int-assoc-eq*: $((A \cap B) \cup C = A \cap (B \cup C)) = (C \subseteq A)$
— Halmos, Naive Set Theory, page 16.
by *blast*

lemma *Compl-UNIV-eq* [*simp*]: $-UNIV = \{\}$
by *blast*

lemma *Compl-empty-eq* [*simp*]: $-\{\} = UNIV$
by *blast*

lemma *Compl-subset-Compl-iff* [*iff*]: $(-A \subseteq -B) = (B \subseteq A)$
by *blast*

lemma *Compl-eq-Compl-iff* [*iff*]: $(-A = -B) = (A = (B::'a\ set))$
by *blast*

Union.

lemma *Union-empty* [*simp*]: $Union(\{\}) = \{\}$
by *blast*

lemma *Union-UNIV* [*simp*]: $Union\ UNIV = UNIV$
by *blast*

lemma *Union-insert* [*simp*]: $Union\ (insert\ a\ B) = a \cup \bigcup B$
by *blast*

lemma *Union-Un-distrib* [*simp*]: $\bigcup(A\ Un\ B) = \bigcup A \cup \bigcup B$
by *blast*

lemma *Union-Int-subset*: $\bigcup(A \cap B) \subseteq \bigcup A \cap \bigcup B$
by *blast*

lemma *Union-empty-conv* [*simp*,*noatp*]: $(\bigcup A = \{\}) = (\forall x{\in}A.\ x = \{\})$
by *blast*

lemma *empty-Union-conv* [*simp*,*noatp*]: $(\{\} = \bigcup A) = (\forall x{\in}A.\ x = \{\})$
by *blast*

lemma *Union-disjoint*: $(\bigcup C \cap A = \{\}) = (\forall B{\in}C.\ B \cap A = \{\})$
by *blast*

Inter.

lemma *Inter-empty* [*simp*]: $\bigcap\{\} = UNIV$
by *blast*

lemma *Inter-UNIV* [*simp*]: $\bigcap UNIV = \{\}$
by *blast*

lemma *Inter-insert* [*simp*]: $\bigcap(\mathit{insert}\ a\ B) = a \cap \bigcap B$
by *blast*

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap(A \cap B)$
by *blast*

lemma *Inter-Un-distrib*: $\bigcap(A \cup B) = \bigcap A \cap \bigcap B$
by *blast*

lemma *Inter-UNIV-conv* [*simp*,*noatp*]:
$(\bigcap A = \mathit{UNIV}) = (\forall x{\in}A.\ x = \mathit{UNIV})$
$(\mathit{UNIV} = \bigcap A) = (\forall x{\in}A.\ x = \mathit{UNIV})$
by *blast+*

UN and *INT*.

Basic identities:

lemma *UN-empty* [*simp*,*noatp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
by *blast*

lemma *UN-empty2* [*simp*]: $(\bigcup x{\in}A.\ \{\}) = \{\}$
by *blast*

lemma *UN-singleton* [*simp*]: $(\bigcup x{\in}A.\ \{x\}) = A$
by *blast*

lemma *UN-absorb*: $k \in I \Longrightarrow A\ k \cup (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ A\ i)$
by *auto*

lemma *INT-empty* [*simp*]: $(\bigcap x{\in}\{\}.\ B\ x) = \mathit{UNIV}$
by *blast*

lemma *INT-absorb*: $k \in I \Longrightarrow A\ k \cap (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ A\ i)$
by *blast*

lemma *UN-insert* [*simp*]: $(\bigcup x{\in}\mathit{insert}\ a\ A.\ B\ x) = B\ a \cup \mathit{UNION}\ A\ B$
by *blast*

lemma *UN-Un*[*simp*]: $(\bigcup i \in A \cup B.\ M\ i) = (\bigcup i{\in}A.\ M\ i) \cup (\bigcup i{\in}B.\ M\ i)$
by *blast*

lemma *UN-UN-flatten*: $(\bigcup x \in (\bigcup y{\in}A.\ B\ y).\ C\ x) = (\bigcup y{\in}A.\ \bigcup x{\in}B\ y.\ C\ x)$
by *blast*

lemma *UN-subset-iff*: $((\bigcup i{\in}I.\ A\ i) \subseteq B) = (\forall i{\in}I.\ A\ i \subseteq B)$
by *blast*

lemma *INT-subset-iff*: $(B \subseteq (\bigcap i{\in}I.\ A\ i)) = (\forall i{\in}I.\ B \subseteq A\ i)$
by *blast*

lemma *INT-insert* [*simp*]: $(\bigcap x \in insert\ a\ A.\ B\ x) = B\ a \cap INTER\ A\ B$
by *blast*

lemma *INT-Un*: $(\bigcap i \in A \cup B.\ M\ i) = (\bigcap i \in A.\ M\ i) \cap (\bigcap i{\in}B.\ M\ i)$
by *blast*

lemma *INT-insert-distrib*:
$u \in A \Longrightarrow (\bigcap x{\in}A.\ insert\ a\ (B\ x)) = insert\ a\ (\bigcap x{\in}A.\ B\ x)$
by *blast*

lemma *Union-image-eq* [*simp*]: $\bigcup(B`A) = (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *image-Union*: $f\ `\ \bigcup S = (\bigcup x{\in}S.\ f\ `\ x)$
by *blast*

lemma *Inter-image-eq* [*simp*]: $\bigcap(B`A) = (\bigcap x{\in}A.\ B\ x)$
by *blast*

lemma *UN-constant* [*simp*]: $(\bigcup y{\in}A.\ c) = (if\ A = \{\}\ then\ \{\}\ else\ c)$
by *auto*

lemma *INT-constant* [*simp*]: $(\bigcap y{\in}A.\ c) = (if\ A = \{\}\ then\ UNIV\ else\ c)$
by *auto*

lemma *UN-eq*: $(\bigcup x{\in}A.\ B\ x) = \bigcup(\{Y.\ \exists x{\in}A.\ Y = B\ x\})$
by *blast*

lemma *INT-eq*: $(\bigcap x{\in}A.\ B\ x) = \bigcap(\{Y.\ \exists x{\in}A.\ Y = B\ x\})$
— Look: it has an *existential* quantifier
by *blast*

lemma *UNION-empty-conv*[*simp*]:
$(\{\} = (UN\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = \{\})$
$((UN\ x{:}A.\ B\ x) = \{\}) = (\forall x{\in}A.\ B\ x = \{\})$
by *blast+*

lemma *INTER-UNIV-conv*[*simp*]:
$(UNIV = (INT\ x{:}A.\ B\ x)) = (\forall x{\in}A.\ B\ x = UNIV)$
$((INT\ x{:}A.\ B\ x) = UNIV) = (\forall x{\in}A.\ B\ x = UNIV)$
by *blast+*

Distributive laws:

lemma *Int-Union*: $A \cap \bigcup B = (\bigcup C{\in}B.\ A \cap C)$
by *blast*

lemma *Int-Union2*: $\bigcup B \cap A = (\bigcup C{\in}B.\ C \cap A)$
by *blast*

lemma *Un-Union-image*: $(\bigcup x{\in}C.\ A\ x \cup B\ x) = \bigcup(A`C) \cup \bigcup(B`C)$
— Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
— Union of a family of unions
by *blast*

lemma *UN-Un-distrib*: $(\bigcup i{\in}I.\ A\ i \cup B\ i) = (\bigcup i{\in}I.\ A\ i) \cup (\bigcup i{\in}I.\ B\ i)$
— Equivalent version
by *blast*

lemma *Un-Inter*: $A \cup \bigcap B = (\bigcap C{\in}B.\ A \cup C)$
by *blast*

lemma *Int-Inter-image*: $(\bigcap x{\in}C.\ A\ x \cap B\ x) = \bigcap(A`C) \cap \bigcap(B`C)$
by *blast*

lemma *INT-Int-distrib*: $(\bigcap i{\in}I.\ A\ i \cap B\ i) = (\bigcap i{\in}I.\ A\ i) \cap (\bigcap i{\in}I.\ B\ i)$
— Equivalent version
by *blast*

lemma *Int-UN-distrib*: $B \cap (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ B \cap A\ i)$
— Halmos, Naive Set Theory, page 35.
by *blast*

lemma *Un-INT-distrib*: $B \cup (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ B \cup A\ i)$
by *blast*

lemma *Int-UN-distrib2*: $(\bigcup i{\in}I.\ A\ i) \cap (\bigcup j{\in}J.\ B\ j) = (\bigcup i{\in}I.\ \bigcup j{\in}J.\ A\ i \cap B\ j)$
by *blast*

lemma *Un-INT-distrib2*: $(\bigcap i{\in}I.\ A\ i) \cup (\bigcap j{\in}J.\ B\ j) = (\bigcap i{\in}I.\ \bigcap j{\in}J.\ A\ i \cup B\ j)$
by *blast*

Bounded quantifiers.

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

lemma *ball-Un*: $(\forall x \in A \cup B.\ P\ x) = ((\forall x{\in}A.\ P\ x)\ \&\ (\forall x{\in}B.\ P\ x))$
by *blast*

lemma *bex-Un*: $(\exists x \in A \cup B.\ P\ x) = ((\exists x{\in}A.\ P\ x)\ |\ (\exists x{\in}B.\ P\ x))$
by *blast*

lemma *ball-UN*: $(\forall z \in \mathit{UNION}\ A\ B.\ P\ z) = (\forall x{\in}A.\ \forall z \in B\ x.\ P\ z)$
by *blast*

lemma *bex-UN*: $(\exists z \in \mathit{UNION}\ A\ B.\ P\ z) = (\exists x{\in}A.\ \exists z{\in}B\ x.\ P\ z)$

by *blast*

Set difference.

lemma *Diff-eq*: $A - B = A \cap (-B)$
by *blast*

lemma *Diff-eq-empty-iff* [*simp*,*noatp*]: $(A - B = \{\}) = (A \subseteq B)$
by *blast*

lemma *Diff-cancel* [*simp*]: $A - A = \{\}$
by *blast*

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - (B::'a\ set)$
by *blast*

lemma *Diff-triv*: $A \cap B = \{\} \Longrightarrow A - B = A$
by (*blast elim*: *equalityE*)

lemma *empty-Diff* [*simp*]: $\{\} - A = \{\}$
by *blast*

lemma *Diff-empty* [*simp*]: $A - \{\} = A$
by *blast*

lemma *Diff-UNIV* [*simp*]: $A - UNIV = \{\}$
by *blast*

lemma *Diff-insert0* [*simp*,*noatp*]: $x \notin A \Longrightarrow A - insert\ x\ B = A - B$
by *blast*

lemma *Diff-insert*: $A - insert\ a\ B = A - B - \{a\}$
— NOT SUITABLE FOR REWRITING since $\{a\} == insert\ a\ 0$
by *blast*

lemma *Diff-insert2*: $A - insert\ a\ B = A - \{a\} - B$
— NOT SUITABLE FOR REWRITING since $\{a\} == insert\ a\ 0$
by *blast*

lemma *insert-Diff-if*: $insert\ x\ A - B = (if\ x \in B\ then\ A - B\ else\ insert\ x\ (A - B))$
by *auto*

lemma *insert-Diff1* [*simp*]: $x \in B \Longrightarrow insert\ x\ A - B = A - B$
by *blast*

lemma *insert-Diff-single*[*simp*]: $insert\ a\ (A - \{a\}) = insert\ a\ A$
by *blast*

lemma *insert-Diff*: $a \in A \Longrightarrow insert\ a\ (A - \{a\}) = A$

by *blast*

lemma *Diff-insert-absorb*: $x \notin A \Longrightarrow (\mathit{insert}\ x\ A) - \{x\} = A$
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*: $(\forall b.\ P\ b) \longleftrightarrow P\ \mathit{True} \wedge P\ \mathit{False}$

by (*auto intro*: *bool-induct*)

lemma *bool-contrapos*: $P\ x \Longrightarrow \neg\ P\ False \Longrightarrow P\ True$
by (*cases x*) *auto*

lemma *ex-bool-eq*: $(\exists\, b.\ P\ b) \longleftrightarrow P\ True \vee P\ False$
by (*auto intro*: *bool-contrapos*)

lemma *Un-eq-UN*: $A \cup B = (\bigcup b.\ if\ b\ then\ A\ else\ B)$
by (*auto simp add*: *split-if-mem2*)

lemma *UN-bool-eq*: $(\bigcup b{::}bool.\ A\ b) = (A\ True \cup A\ False)$
by (*auto intro*: *bool-contrapos*)

lemma *INT-bool-eq*: $(\bigcap b{::}bool.\ A\ b) = (A\ True \cap A\ False)$
by (*auto intro*: *bool-induct*)

Pow

lemma *Pow-empty* [*simp*]: $Pow\ \{\} = \{\{\}\}$
by (*auto simp add*: *Pow-def*)

lemma *Pow-insert*: $Pow\ (insert\ a\ A) = Pow\ A \cup (insert\ a\ `\ Pow\ A)$
by (*blast intro*: *image-eqI* [**where** *?x* $= u - \{a\}$, *standard*])

lemma *Pow-Compl*: $Pow\ (-\ A) = \{-B \mid B.\ A \in Pow\ B\}$
by (*blast intro*: *exI* [**where** *?x* $= -\ u$, *standard*])

lemma *Pow-UNIV* [*simp*]: $Pow\ UNIV = UNIV$
by *blast*

lemma *Un-Pow-subset*: $Pow\ A \cup Pow\ B \subseteq Pow\ (A \cup B)$
by *blast*

lemma *UN-Pow-subset*: $(\bigcup x{\in}A.\ Pow\ (B\ x)) \subseteq Pow\ (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *subset-Pow-Union*: $A \subseteq Pow\ (\bigcup A)$
by *blast*

lemma *Union-Pow-eq* [*simp*]: $\bigcup(Pow\ A) = A$
by *blast*

lemma *Pow-Int-eq* [*simp*]: $Pow\ (A \cap B) = Pow\ A \cap Pow\ B$
by *blast*

lemma *Pow-INT-eq*: $Pow\ (\bigcap x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ Pow\ (B\ x))$
by *blast*

Miscellany.

lemma *set-eq-subset*: $(A = B) = (A \subseteq B \ \& \ B \subseteq A)$
by *blast*

lemma *subset-iff*: $(A \subseteq B) = (\forall t.\ t \in A \longrightarrow t \in B)$
by *blast*

lemma *subset-iff-psubset-eq*: $(A \subseteq B) = ((A \subset B) \mid (A = B))$
by (*unfold less-le*) *blast*

lemma *all-not-in-conv* [*simp*]: $(\forall x.\ x \notin A) = (A = \{\})$
by *blast*

lemma *ex-in-conv*: $(\exists x.\ x \in A) = (A \neq \{\})$
by *blast*

lemma *distinct-lemma*: $f\ x \neq f\ y \Longrightarrow x \neq y$
by *iprover*

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma *UN-simps* [*simp*]:
$!!a\ B\ C.\ (UN\ x{:}C.\ insert\ a\ (B\ x)) = (if\ C{=}\{\}\ then\ \{\}\ else\ insert\ a\ (UN\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Un\ B) = ((if\ C{=}\{\}\ then\ \{\}\ else\ (UN\ x{:}C.\ A\ x)\ Un\ B))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ Un\ B\ x) = ((if\ C{=}\{\}\ then\ \{\}\ else\ A\ Un\ (UN\ x{:}C.\ B\ x)))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x\ Int\ B) = ((UN\ x{:}C.\ A\ x)\ Int\ B)$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ Int\ B\ x) = (A\ Int\ (UN\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (UN\ x{:}C.\ A\ x - B) = ((UN\ x{:}C.\ A\ x) - B)$
$!!A\ B\ C.\ (UN\ x{:}C.\ A - B\ x) = (A - (INT\ x{:}C.\ B\ x))$
$!!A\ B.\ (UN\ x{:}\ Union\ A.\ B\ x) = (UN\ y{:}A.\ UN\ x{:}y.\ B\ x)$
$!!A\ B\ C.\ (UN\ z{:}\ UNION\ A\ B.\ C\ z) = (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z)$
$!!A\ B\ f.\ (UN\ x{:}f`A.\ B\ x) = (UN\ a{:}A.\ B\ (f\ a))$
by *auto*

lemma *INT-simps* [*simp*]:
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Int\ B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x)\ Int\ B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ Int\ B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A\ Int\ (INT\ x{:}C.\ B\ x))$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x - B) = (if\ C{=}\{\}\ then\ UNIV\ else\ (INT\ x{:}C.\ A\ x) - B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A - B\ x) = (if\ C{=}\{\}\ then\ UNIV\ else\ A - (UN\ x{:}C.\ B\ x))$
$!!a\ B\ C.\ (INT\ x{:}C.\ insert\ a\ (B\ x)) = insert\ a\ (INT\ x{:}C.\ B\ x)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ x\ Un\ B) = ((INT\ x{:}C.\ A\ x)\ Un\ B)$
$!!A\ B\ C.\ (INT\ x{:}C.\ A\ Un\ B\ x) = (A\ Un\ (INT\ x{:}C.\ B\ x))$
$!!A\ B.\ (INT\ x{:}\ Union\ A.\ B\ x) = (INT\ y{:}A.\ INT\ x{:}y.\ B\ x)$
$!!A\ B\ C.\ (INT\ z{:}\ UNION\ A\ B.\ C\ z) = (INT\ x{:}A.\ INT\ z{:}\ B(x).\ C\ z)$

!!$A\ B\ f.\ (INT\ x{:}f`A.\ B\ x) \quad = (INT\ a{:}A.\ B\ (f\ a))$
by *auto*

lemma *ball-simps* [*simp*,*noatp*]:
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x \mid Q) = ((ALL\ x{:}A.\ P\ x) \mid Q)$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P \mid Q\ x) = (P \mid (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P \dashrightarrow Q\ x) = (P \dashrightarrow (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x \dashrightarrow Q) = ((EX\ x{:}A.\ P\ x) \dashrightarrow Q)$
!!$P.\ (ALL\ x{:}\{\}.\ P\ x) = True$
!!$P.\ (ALL\ x{:}UNIV.\ P\ x) = (ALL\ x.\ P\ x)$
!!$a\ B\ P.\ (ALL\ x{:}insert\ a\ B.\ P\ x) = (P\ a\ \&\ (ALL\ x{:}B.\ P\ x))$
!!$A\ P.\ (ALL\ x{:}Union\ A.\ P\ x) = (ALL\ y{:}A.\ ALL\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (ALL\ x{:}\ UNION\ A\ B.\ P\ x) = (ALL\ a{:}A.\ ALL\ x{:}\ B\ a.\ P\ x)$
!!$P\ Q.\ (ALL\ x{:}Collect\ Q.\ P\ x) = (ALL\ x.\ Q\ x \dashrightarrow P\ x)$
!!$A\ P\ f.\ (ALL\ x{:}f`A.\ P\ x) = (ALL\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(ALL\ x{:}A.\ P\ x)) = (EX\ x{:}A.\ \sim P\ x)$
by *auto*

lemma *bex-simps* [*simp*,*noatp*]:
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ x\ \&\ Q) = ((EX\ x{:}A.\ P\ x)\ \&\ Q)$
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x{:}A.\ Q\ x))$
!!$P.\ (EX\ x{:}\{\}.\ P\ x) = False$
!!$P.\ (EX\ x{:}UNIV.\ P\ x) = (EX\ x.\ P\ x)$
!!$a\ B\ P.\ (EX\ x{:}insert\ a\ B.\ P\ x) = (P(a) \mid (EX\ x{:}B.\ P\ x))$
!!$A\ P.\ (EX\ x{:}Union\ A.\ P\ x) = (EX\ y{:}A.\ EX\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (EX\ x{:}\ UNION\ A\ B.\ P\ x) = (EX\ a{:}A.\ EX\ x{:}B\ a.\ P\ x)$
!!$P\ Q.\ (EX\ x{:}Collect\ Q.\ P\ x) = (EX\ x.\ Q\ x\ \&\ P\ x)$
!!$A\ P\ f.\ (EX\ x{:}f`A.\ P\ x) = (EX\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(EX\ x{:}A.\ P\ x)) = (ALL\ x{:}A.\ \sim P\ x)$
by *auto*

lemma *ball-conj-distrib*:
$(ALL\ x{:}A.\ P\ x\ \&\ Q\ x) = ((ALL\ x{:}A.\ P\ x)\ \&\ (ALL\ x{:}A.\ Q\ x))$
by *blast*

lemma *bex-disj-distrib*:
$(EX\ x{:}A.\ P\ x \mid Q\ x) = ((EX\ x{:}A.\ P\ x) \mid (EX\ x{:}A.\ Q\ x))$
by *blast*

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:
!!$a\ B\ C.\ insert\ a\ (UN\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ \{a\}\ else\ (UN\ x{:}C.\ insert\ a\ (B\ x)))$
!!$A\ B\ C.\ (UN\ x{:}C.\ A\ x)\ Un\ B \quad = (if\ C{=}\{\}\ then\ B\ else\ (UN\ x{:}C.\ A\ x\ Un\ B))$
!!$A\ B\ C.\ A\ Un\ (UN\ x{:}C.\ B\ x) \quad = (if\ C{=}\{\}\ then\ A\ else\ (UN\ x{:}C.\ A\ Un\ B\ x))$
!!$A\ B\ C.\ ((UN\ x{:}C.\ A\ x)\ Int\ B) = (UN\ x{:}C.\ A\ x\ Int\ B)$
!!$A\ B\ C.\ (A\ Int\ (UN\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A\ Int\ B\ x)$
!!$A\ B\ C.\ ((UN\ x{:}C.\ A\ x) - B) = (UN\ x{:}C.\ A\ x - B)$
!!$A\ B\ C.\ (A - (INT\ x{:}C.\ B\ x)) = (UN\ x{:}C.\ A - B\ x)$

!!$A\ B.\ (UN\ y{:}A.\ UN\ x{:}y.\ B\ x) = (UN\ x{:}\ Union\ A.\ B\ x)$
!!$A\ B\ C.\ (UN\ \ x{:}A.\ UN\ z{:}\ B(x).\ C\ z) = (UN\ z{:}\ UNION\ A\ B.\ C\ z)$
!!$A\ B\ f.\ (UN\ a{:}A.\ B\ (f\ a)) = (UN\ x{:}f`A.\ B\ x)$
by *auto*

lemma *INT-extend-simps*:
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x)\ Int\ B = (if\ C{=}\{\}\ then\ B\ else\ (INT\ x{:}C.\ A\ x\ Int\ B))$
!!$A\ B\ C.\ A\ Int\ (INT\ x{:}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (INT\ x{:}C.\ A\ Int\ B\ x))$
!!$A\ B\ C.\ (INT\ x{:}C.\ A\ x) - B\ \ = (if\ C{=}\{\}\ then\ UNIV{-}B\ else\ (INT\ x{:}C.\ A\ x - B))$
!!$A\ B\ C.\ A - (UN\ x{:}C.\ B\ x)\ \ = (if\ C{=}\{\}\ then\ A\ else\ (INT\ x{:}C.\ A - B\ x))$
!!$a\ B\ C.\ insert\ a\ (INT\ x{:}C.\ B\ x) = (INT\ x{:}C.\ insert\ a\ (B\ x))$
!!$A\ B\ C.\ ((INT\ x{:}C.\ A\ x)\ Un\ B)\ = (INT\ x{:}C.\ A\ x\ Un\ B)$
!!$A\ B\ C.\ A\ Un\ (INT\ x{:}C.\ B\ x)\ = (INT\ x{:}C.\ A\ Un\ B\ x)$
!!$A\ B.\ (INT\ y{:}A.\ INT\ x{:}y.\ B\ x) = (INT\ x{:}\ Union\ A.\ B\ x)$
!!$A\ B\ C.\ (INT\ x{:}A.\ INT\ z{:}\ B(x).\ C\ z) = (INT\ z{:}\ UNION\ A\ B.\ C\ z)$
!!$A\ B\ f.\ (INT\ a{:}A.\ B\ (f\ a))\ \ = (INT\ x{:}f`A.\ B\ x)$
by *auto*

5.5.3 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \Longrightarrow f`A \subseteq f`B$
by *blast*

lemma *Pow-mono*: $A \subseteq B \Longrightarrow Pow\ A \subseteq Pow\ B$
by *blast*

lemma *Union-mono*: $A \subseteq B \Longrightarrow \bigcup A \subseteq \bigcup B$
by *blast*

lemma *Inter-anti-mono*: $B \subseteq A \Longrightarrow \bigcap A \subseteq \bigcap B$
by *blast*

lemma *UN-mono*:
$A \subseteq B \Longrightarrow (!!x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow$
$(\bigcup x{\in}A.\ f\ x) \subseteq (\bigcup x{\in}B.\ g\ x)$
by (*blast dest*: *subsetD*)

lemma *INT-anti-mono*:
$B \subseteq A \Longrightarrow (!!x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow$
$(\bigcap x{\in}A.\ f\ x) \subseteq (\bigcap x{\in}A.\ g\ x)$
— The last inclusion is POSITIVE!
by (*blast dest*: *subsetD*)

lemma *insert-mono*: $C \subseteq D \Longrightarrow insert\ a\ C \subseteq insert\ a\ D$
by *blast*

lemma *Un-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cup B \subseteq C \cup D$
by *blast*

lemma *Int-mono*: $A \subseteq C ==> B \subseteq D ==> A \cap B \subseteq C \cap D$
by *blast*

lemma *Diff-mono*: $A \subseteq C ==> D \subseteq B ==> A - B \subseteq C - D$
by *blast*

lemma *Compl-anti-mono*: $A \subseteq B ==> -B \subseteq -A$
by *blast*

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B ==> x \in A --> x \in B$
apply (*rule impI*)
apply (*erule subsetD*, *assumption*)
done

lemma *conj-mono*: $P1 --> Q1 ==> P2 --> Q2 ==> (P1 \& P2) --> (Q1 \& Q2)$
by *iprover*

lemma *disj-mono*: $P1 --> Q1 ==> P2 --> Q2 ==> (P1 \mid P2) --> (Q1 \mid Q2)$
by *iprover*

lemma *imp-mono*: $Q1 --> P1 ==> P2 --> Q2 ==> (P1 --> P2) --> (Q1 --> Q2)$
by *iprover*

lemma *imp-refl*: $P --> P$ **..**

lemma *ex-mono*: $(!!x.\ P\ x --> Q\ x) ==> (EX\ x.\ P\ x) --> (EX\ x.\ Q\ x)$
by *iprover*

lemma *all-mono*: $(!!x.\ P\ x --> Q\ x) ==> (ALL\ x.\ P\ x) --> (ALL\ x.\ Q\ x)$
by *iprover*

lemma *Collect-mono*: $(!!x.\ P\ x --> Q\ x) ==> Collect\ P \subseteq Collect\ Q$
by *blast*

lemma *Int-Collect-mono*:
$A \subseteq B ==> (!!x.\ x \in A ==> P\ x --> Q\ x) ==> A \cap Collect\ P \subseteq B \cap Collect\ Q$
by *blast*

lemmas *basic-monos* =
subset-refl imp-refl disj-mono conj-mono
ex-mono Collect-mono in-mono

lemma *eq-to-mono*: $a = b ==> c = d ==> b --> d ==> a --> c$

by *iprover*

lemma *eq-to-mono2*: $a = b \Longrightarrow c = d \Longrightarrow \sim b \longrightarrow \sim d \Longrightarrow \sim a \longrightarrow \sim c$
by *iprover*

5.6 Inverse image of a function

constdefs
vimage :: $('a \Rightarrow 'b) \Rightarrow 'b\ set \Rightarrow 'a\ set$ (**infixr** $-`$ *90*)
[*code del*]: $f\ -`\ B == \{x.\ f\ x : B\}$

5.6.1 Basic rules

lemma *vimage-eq* [*simp*]: $(a : f\ -`\ B) = (f\ a : B)$
by (*unfold vimage-def*) *blast*

lemma *vimage-singleton-eq*: $(a : f\ -`\ \{b\}) = (f\ a = b)$
by *simp*

lemma *vimageI* [*intro*]: $f\ a = b \Longrightarrow b{:}B \Longrightarrow a : f\ -`\ B$
by (*unfold vimage-def*) *blast*

lemma *vimageI2*: $f\ a : A \Longrightarrow a : f\ -`\ A$
by (*unfold vimage-def*) *fast*

lemma *vimageE* [*elim!*]: $a{:}\ f\ -`\ B \Longrightarrow (!!x.\ f\ a = x \Longrightarrow x{:}B \Longrightarrow P) \Longrightarrow P$
by (*unfold vimage-def*) *blast*

lemma *vimageD*: $a : f\ -`\ A \Longrightarrow f\ a : A$
by (*unfold vimage-def*) *fast*

5.6.2 Equations

lemma *vimage-empty* [*simp*]: $f\ -`\ \{\} = \{\}$
by *blast*

lemma *vimage-Compl*: $f\ -`\ (-A) = -(f\ -`\ A)$
by *blast*

lemma *vimage-Un* [*simp*]: $f\ -`\ (A\ Un\ B) = (f\ -`\ A)\ Un\ (f\ -`\ B)$
by *blast*

lemma *vimage-Int* [*simp*]: $f\ -`\ (A\ Int\ B) = (f\ -`\ A)\ Int\ (f\ -`\ B)$
by *fast*

lemma *vimage-Union*: $f\ -`\ (Union\ A) = (UN\ X{:}A.\ f\ -`\ X)$
by *blast*

lemma *vimage-UN*: $f-`(UN\ x{:}A.\ B\ x) = (UN\ x{:}A.\ f\ -`\ B\ x)$

by *blast*

lemma *vimage-INT*: $f-`(INT\ x{:}A.\ B\ x) = (INT\ x{:}A.\ f\ -`\ B\ x)$
by *blast*

lemma *vimage-Collect-eq* [*simp*]: $f\ -`\ Collect\ P = \{y.\ P\ (f\ y)\}$
by *blast*

lemma *vimage-Collect*: $(!!x.\ P\ (f\ x) = Q\ x) ==> f\ -`\ (Collect\ P) = Collect\ Q$
by *blast*

lemma *vimage-insert*: $f-`(insert\ a\ B) = (f-`\{a\})\ Un\ (f-`B)$
— NOT suitable for rewriting because of the recurrence of $\{a\}$.
by *blast*

lemma *vimage-Diff*: $f\ -`\ (A\ -\ B) = (f\ -`\ A)\ -\ (f\ -`\ B)$
by *blast*

lemma *vimage-UNIV* [*simp*]: $f\ -`\ UNIV = UNIV$
by *blast*

lemma *vimage-eq-UN*: $f-`B = (UN\ y{:}\ B.\ f-`\{y\})$
— NOT suitable for rewriting
by *blast*

lemma *vimage-mono*: $A \subseteq B ==> f\ -`\ A \subseteq f\ -`\ B$
— monotonicity
by *blast*

lemma *vimage-image-eq* [*noatp*]: $f\ -`\ (f\ `\ A) = \{y.\ EX\ x{:}A.\ f\ x = f\ y\}$
by (*blast intro*: *sym*)

lemma *image-vimage-subset*: $f\ `\ (f\ -`\ A) <= A$
by *blast*

lemma *image-vimage-eq* [*simp*]: $f\ `\ (f\ -`\ A) = A\ Int\ range\ f$
by *blast*

lemma *image-Int-subset*: $f`(A\ Int\ B) <= f`A\ Int\ f`B$
by *blast*

lemma *image-diff-subset*: $f`A\ -\ f`B <= f`(A\ -\ B)$
by *blast*

lemma *image-UN*: $(f\ `\ (UNION\ A\ B)) = (UN\ x{:}A.(f\ `\ (B\ x)))$
by *blast*

5.7 Getting the Contents of a Singleton Set

definition *contents* :: *'a set* ⇒ *'a* **where**
  [*code del*]: *contents X* = (*THE x*. *X* = {*x*})

lemma *contents-eq* [*simp*]: *contents* {*x*} = *x*
  by (*simp add*: *contents-def*)

5.8 Transitivity rules for calculational reasoning

lemma *set-rev-mp*: *x*:*A* ==> *A* ⊆ *B* ==> *x*:*B*
  by (*rule subsetD*)

lemma *set-mp*: *A* ⊆ *B* ==> *x*:*A* ==> *x*:*B*
  by (*rule subsetD*)

lemmas *basic-trans-rules* [*trans*] =
  order-trans-rules set-rev-mp set-mp

5.9 Least value operator

lemma *Least-mono*:
  mono (*f*::*'a*::*order* => *'b*::*order*) ==> *EX x*:*S*. *ALL y*:*S*. *x* <= *y*
    ==> (*LEAST y*. *y* : *f* ‘ *S*) = *f* (*LEAST x*. *x* : *S*)
    — Courtesy of Stephan Merz
  apply *clarify*
  apply (*erule-tac P* = %*x*. *x* : *S* **in** *LeastI2-order*, *fast*)
  apply (*rule LeastI2-order*)
  apply (*auto elim*: *monoD intro*!: *order-antisym*)
  done

5.10 Rudimentary code generation

lemma *empty-code* [*code*]: {} *x* ⟷ *False*
  unfolding *empty-def Collect-def* **..**

lemma *UNIV-code* [*code*]: *UNIV x* ⟷ *True*
  unfolding *UNIV-def Collect-def* **..**

lemma *insert-code* [*code*]: *insert y A x* ⟷ *y* = *x* ∨ *A x*
  unfolding *insert-def Collect-def mem-def Un-def* **by** *auto*

lemma *inter-code* [*code*]: (*A* ∩ *B*) *x* ⟷ *A x* ∧ *B x*
  unfolding *Int-def Collect-def mem-def* **..**

lemma *union-code* [*code*]: (*A* ∪ *B*) *x* ⟷ *A x* ∨ *B x*
  unfolding *Un-def Collect-def mem-def* **..**

lemma *vimage-code* [*code*]: (*f* −‘ *A*) *x* = *A* (*f x*)
  unfolding *vimage-def Collect-def mem-def* **..**

5.11 Complete lattices

notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*)

class *complete-lattice* = *lattice* + *bot* + *top* +
fixes *Inf* :: *'a set* $\Rightarrow$ *'a* ($\bigsqcap$- [*900*] *900*)
and *Sup* :: *'a set* $\Rightarrow$ *'a* ($\bigsqcup$- [*900*] *900*)
assumes *Inf-lower*: $x \in A \Longrightarrow \bigsqcap A \sqsubseteq x$
and *Inf-greatest*: $(\bigwedge x.\ x \in A \Longrightarrow z \sqsubseteq x) \Longrightarrow z \sqsubseteq \bigsqcap A$
assumes *Sup-upper*: $x \in A \Longrightarrow x \sqsubseteq \bigsqcup A$
and *Sup-least*: $(\bigwedge x.\ x \in A \Longrightarrow x \sqsubseteq z) \Longrightarrow \bigsqcup A \sqsubseteq z$
begin

lemma *Inf-Sup*: $\bigsqcap A = \bigsqcup \{b.\ \forall a \in A.\ b \leq a\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Sup-Inf*: $\bigsqcup A = \bigsqcap \{b.\ \forall a \in A.\ a \leq b\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Inf-Univ*: $\bigsqcap UNIV = \bigsqcup \{\}$
unfolding *Sup-Inf* **by** *auto*

lemma *Sup-Univ*: $\bigsqcup UNIV = \bigsqcap \{\}$
unfolding *Inf-Sup* **by** *auto*

lemma *Inf-insert*: $\bigsqcap insert\ a\ A = a \sqcap \bigsqcap A$
by (*auto intro*: *antisym Inf-greatest Inf-lower*)

lemma *Sup-insert*: $\bigsqcup insert\ a\ A = a \sqcup \bigsqcup A$
by (*auto intro*: *antisym Sup-least Sup-upper*)

lemma *Inf-singleton* [*simp*]:
$\bigsqcap \{a\} = a$
by (*auto intro*: *antisym Inf-lower Inf-greatest*)

lemma *Sup-singleton* [*simp*]:
$\bigsqcup \{a\} = a$
by (*auto intro*: *antisym Sup-upper Sup-least*)

lemma *Inf-insert-simp*:
$\bigsqcap insert\ a\ A = (if\ A = \{\}\ then\ a\ else\ a \sqcap \bigsqcap A)$
by (*cases* $A = \{\}$) (*simp-all*, *simp add*: *Inf-insert*)

lemma *Sup-insert-simp*:
$\bigsqcup insert\ a\ A = (if\ A = \{\}\ then\ a\ else\ a \sqcup \bigsqcup A)$
by (*cases* $A = \{\}$) (*simp-all*, *simp add*: *Sup-insert*)

lemma *Inf-binary*:
$\bigsqcap\{a, b\} = a \sqcap b$
by (*simp add*: *Inf-insert-simp*)

lemma *Sup-binary*:
$\bigsqcup\{a, b\} = a \sqcup b$
by (*simp add*: *Sup-insert-simp*)

lemma *bot-def*:
$bot = \bigsqcup\{\}$
by (*auto intro*: *antisym Sup-least*)

lemma *top-def*:
$top = \bigsqcap\{\}$
by (*auto intro*: *antisym Inf-greatest*)

lemma *sup-bot* [*simp*]:
$x \sqcup bot = x$
using *bot-least* [*of x*] **by** (*simp add*: *le-iff-sup sup-commute*)

lemma *inf-top* [*simp*]:
$x \sqcap top = x$
using *top-greatest* [*of x*] **by** (*simp add*: *le-iff-inf inf-commute*)

definition *SUPR* :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a* **where**
SUPR A f == $\bigsqcup$ (*f* ‘ *A*)

definition *INFI* :: *'b set* ⇒ (*'b* ⇒ *'a*) ⇒ *'a* **where**
INFI A f == $\bigsqcap$ (*f* ‘ *A*)

end

syntax
-SUP1 :: *pttrns* => *'b* => *'b* (*3SUP -./ -*) [*0*, *10*] *10*)
-SUP :: *pttrn* => *'a set* => *'b* => *'b* (*3SUP -:-./ -*) [*0*, *10*] *10*)
-INF1 :: *pttrns* => *'b* => *'b* (*3INF -./ -*) [*0*, *10*] *10*)
-INF :: *pttrn* => *'a set* => *'b* => *'b* (*3INF -:-./ -*) [*0*, *10*] *10*)

translations
SUP x y. B == *SUP x. SUP y. B*
SUP x. B == *CONST SUPR CONST UNIV* (%*x. B*)
SUP x. B == *SUP x:CONST UNIV. B*
SUP x:A. B == *CONST SUPR A* (%*x. B*)
INF x y. B == *INF x. INF y. B*
INF x. B == *CONST INFI CONST UNIV* (%*x. B*)
INF x. B == *INF x:CONST UNIV. B*
INF x:A. B == *CONST INFI A* (%*x. B*)

print-translation ⟪
let
  fun btr' syn (*A* :: *Abs abs* :: *ts*) =
    let val (*x*,*t*) = *atomic-abs-tr' abs*
    in list-comb (*Syntax.const syn* $ *x* $ *A* $ *t*, *ts*) *end*
  val const-syntax-name = *Sign.const-syntax-name* @{*theory*} *o fst o dest-Const*
in
[(*const-syntax-name* @{*term SUPR*}, *btr' -SUP*),(*const-syntax-name* @{*term INFI*}, *btr' -INF*)]
end
⟫

context *complete-lattice*
begin

lemma *le-SUPI*: *i* : *A* ⟹ *M i* ≤ (*SUP i*:*A*. *M i*)
  by (*auto simp add*: *SUPR-def intro*: *Sup-upper*)

lemma *SUP-leI*: (⋀*i*. *i* : *A* ⟹ *M i* ≤ *u*) ⟹ (*SUP i*:*A*. *M i*) ≤ *u*
  by (*auto simp add*: *SUPR-def intro*: *Sup-least*)

lemma *INF-leI*: *i* : *A* ⟹ (*INF i*:*A*. *M i*) ≤ *M i*
  by (*auto simp add*: *INFI-def intro*: *Inf-lower*)

lemma *le-INFI*: (⋀*i*. *i* : *A* ⟹ *u* ≤ *M i*) ⟹ *u* ≤ (*INF i*:*A*. *M i*)
  by (*auto simp add*: *INFI-def intro*: *Inf-greatest*)

lemma *SUP-const*[*simp*]: *A* ≠ {} ⟹ (*SUP i*:*A*. *M*) = *M*
  by (*auto intro*: *antisym SUP-leI le-SUPI*)

lemma *INF-const*[*simp*]: *A* ≠ {} ⟹ (*INF i*:*A*. *M*) = *M*
  by (*auto intro*: *antisym INF-leI le-INFI*)

end

5.12 Bool as complete lattice

instantiation *bool* :: *complete-lattice*
begin

definition
  Inf-bool-def: ⨅ *A* ⟷ (∀ *x*∈*A*. *x*)

definition
  Sup-bool-def: ⨆ *A* ⟷ (∃ *x*∈*A*. *x*)

instance
  by *intro-classes* (*auto simp add*: *Inf-bool-def Sup-bool-def le-bool-def*)

end

lemma *Inf-empty-bool* [*simp*]:
$\bigsqcap \{\}$
unfolding *Inf-bool-def* **by** *auto*

lemma *not-Sup-empty-bool* [*simp*]:
$\neg \bigsqcup \{\}$
unfolding *Sup-bool-def* **by** *auto*

5.13 Fun as complete lattice

instantiation *fun* :: (*type*, *complete-lattice*) *complete-lattice*
begin

definition
Inf-fun-def [*code del*]: $\bigsqcap A = (\lambda x.\ \bigsqcap \{y.\ \exists f{\in}A.\ y = f\ x\})$

definition
Sup-fun-def [*code del*]: $\bigsqcup A = (\lambda x.\ \bigsqcup \{y.\ \exists f{\in}A.\ y = f\ x\})$

instance
by *intro-classes*
(*auto simp add*: *Inf-fun-def Sup-fun-def le-fun-def*
intro: *Inf-lower Sup-upper Inf-greatest Sup-least*)

end

lemma *Inf-empty-fun*:
$\bigsqcap \{\} = (\lambda\text{-}.\ \bigsqcap \{\})$
by *rule* (*auto simp add*: *Inf-fun-def*)

lemma *Sup-empty-fun*:
$\bigsqcup \{\} = (\lambda\text{-}.\ \bigsqcup \{\})$
by *rule* (*auto simp add*: *Sup-fun-def*)

5.14 Set as lattice

lemma *inf-set-eq*: $A \sqcap B = A \cap B$
apply (*rule subset-antisym*)
apply (*rule Int-greatest*)
apply (*rule inf-le1*)
apply (*rule inf-le2*)
apply (*rule inf-greatest*)
apply (*rule Int-lower1*)
apply (*rule Int-lower2*)
done

lemma *sup-set-eq*: $A \sqcup B = A \cup B$

apply (*rule subset-antisym*)
apply (*rule sup-least*)
apply (*rule Un-upper1*)
apply (*rule Un-upper2*)
apply (*rule Un-least*)
apply (*rule sup-ge1*)
apply (*rule sup-ge2*)
done

lemma *mono-Int*: *mono f* $\Longrightarrow$ *f* (*A* $\cap$ *B*) $\subseteq$ *f A* $\cap$ *f B*
apply (*fold inf-set-eq sup-set-eq*)
apply (*erule mono-inf*)
done

lemma *mono-Un*: *mono f* $\Longrightarrow$ *f A* $\cup$ *f B* $\subseteq$ *f* (*A* $\cup$ *B*)
apply (*fold inf-set-eq sup-set-eq*)
apply (*erule mono-sup*)
done

lemma *Inf-set-eq*: $\bigsqcap S = \bigcap S$
apply (*rule subset-antisym*)
apply (*rule Inter-greatest*)
apply (*erule Inf-lower*)
apply (*rule Inf-greatest*)
apply (*erule Inter-lower*)
done

lemma *Sup-set-eq*: $\bigsqcup S = \bigcup S$
apply (*rule subset-antisym*)
apply (*rule Sup-least*)
apply (*erule Union-upper*)
apply (*rule Union-least*)
apply (*erule Sup-upper*)
done

lemma *top-set-eq*: *top* = *UNIV*
by (*iprover intro*!: *subset-antisym subset-UNIV top-greatest*)

lemma *bot-set-eq*: *bot* = {}
by (*iprover intro*!: *subset-antisym empty-subsetI bot-least*)

no-notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
Inf ($\bigsqcap$ - [*900*] *900*) **and**
Sup ($\bigsqcup$ - [*900*] *900*)

5.15 Misc theorem and ML bindings

lemmas *equalityI = subset-antisym*
lemmas *mem-simps =*
insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff
mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [*simp*] above.

ML ⟪
val Ball-def = @{thm Ball-def}
val Bex-def = @{thm Bex-def}
val CollectD = @{thm CollectD}
val CollectE = @{thm CollectE}
val CollectI = @{thm CollectI}
val Collect-conj-eq = @{thm Collect-conj-eq}
val Collect-mem-eq = @{thm Collect-mem-eq}
val IntD1 = @{thm IntD1}
val IntD2 = @{thm IntD2}
val IntE = @{thm IntE}
val IntI = @{thm IntI}
val Int-Collect = @{thm Int-Collect}
val UNIV-I = @{thm UNIV-I}
val UNIV-witness = @{thm UNIV-witness}
val UnE = @{thm UnE}
val UnI1 = @{thm UnI1}
val UnI2 = @{thm UnI2}
val ballE = @{thm ballE}
val ballI = @{thm ballI}
val bexCI = @{thm bexCI}
val bexE = @{thm bexE}
val bexI = @{thm bexI}
val bex-triv = @{thm bex-triv}
val bspec = @{thm bspec}
val contra-subsetD = @{thm contra-subsetD}
val distinct-lemma = @{thm distinct-lemma}
val eq-to-mono = @{thm eq-to-mono}
val eq-to-mono2 = @{thm eq-to-mono2}
val equalityCE = @{thm equalityCE}
val equalityD1 = @{thm equalityD1}
val equalityD2 = @{thm equalityD2}
val equalityE = @{thm equalityE}
val equalityI = @{thm equalityI}
val imageE = @{thm imageE}
val imageI = @{thm imageI}
val image-Un = @{thm image-Un}
val image-insert = @{thm image-insert}
val insert-commute = @{thm insert-commute}
val insert-iff = @{thm insert-iff}
val mem-Collect-eq = @{thm mem-Collect-eq}
val rangeE = @{thm rangeE}

val rangeI = @{thm rangeI}
val range-eqI = @{thm range-eqI}
val subsetCE = @{thm subsetCE}
val subsetD = @{thm subsetD}
val subsetI = @{thm subsetI}
val subset-refl = @{thm subset-refl}
val subset-trans = @{thm subset-trans}
val vimageD = @{thm vimageD}
val vimageE = @{thm vimageE}
val vimageI = @{thm vimageI}
val vimageI2 = @{thm vimageI2}
val vimage-Collect = @{thm vimage-Collect}
val vimage-Int = @{thm vimage-Int}
val vimage-Un = @{thm vimage-Un}
⟩⟩

end

6 Typedef: HOL type definitions

theory *Typedef*
imports *Set*
uses
(*Tools/typedef-package.ML*)
(*Tools/typecopy-package.ML*)
(*Tools/typedef-codegen.ML*)
begin

ML ⟨⟨
structure HOL = struct val thy = theory HOL end;
⟩⟩ — belongs to theory HOL

locale *type-definition* =
fixes *Rep* **and** *Abs* **and** *A*
assumes *Rep*: *Rep x* ∈ *A*
and *Rep-inverse*: *Abs* (*Rep x*) = *x*
and *Abs-inverse*: *y* ∈ *A* ==> *Rep* (*Abs y*) = *y*
— This will be axiomatized for each typedef!
begin

lemma *Rep-inject*:
(*Rep x* = *Rep y*) = (*x* = *y*)
proof
assume *Rep x* = *Rep y*
then have *Abs* (*Rep x*) = *Abs* (*Rep y*) **by** (*simp only*:)
moreover have *Abs* (*Rep x*) = *x* **by** (*rule Rep-inverse*)
moreover have *Abs* (*Rep y*) = *y* **by** (*rule Rep-inverse*)
ultimately show *x* = *y* **by** *simp*

next
assume $x = y$
thus $Rep\ x = Rep\ y$ **by** (*simp only*:)
qed

lemma *Abs-inject*:
assumes x: $x \in A$ **and** y: $y \in A$
shows $(Abs\ x = Abs\ y) = (x = y)$
proof
assume $Abs\ x = Abs\ y$
then have $Rep\ (Abs\ x) = Rep\ (Abs\ y)$ **by** (*simp only*:)
moreover from x **have** $Rep\ (Abs\ x) = x$ **by** (*rule Abs-inverse*)
moreover from y **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
ultimately show $x = y$ **by** *simp*
next
assume $x = y$
thus $Abs\ x = Abs\ y$ **by** (*simp only*:)
qed

lemma *Rep-cases* [*cases set*]:
assumes y: $y \in A$
and *hyp*: $!!x.\ y = Rep\ x \Longrightarrow P$
shows P
proof (*rule hyp*)
from y **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
thus $y = Rep\ (Abs\ y)$ **..**
qed

lemma *Abs-cases* [*cases type*]:
assumes r: $!!y.\ x = Abs\ y \Longrightarrow y \in A \Longrightarrow P$
shows P
proof (*rule r*)
have $Abs\ (Rep\ x) = x$ **by** (*rule Rep-inverse*)
thus $x = Abs\ (Rep\ x)$ **..**
show $Rep\ x \in A$ **by** (*rule Rep*)
qed

lemma *Rep-induct* [*induct set*]:
assumes y: $y \in A$
and *hyp*: $!!x.\ P\ (Rep\ x)$
shows $P\ y$
proof −
have $P\ (Rep\ (Abs\ y))$ **by** (*rule hyp*)
moreover from y **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
ultimately show $P\ y$ **by** *simp*
qed

lemma *Abs-induct* [*induct type*]:
assumes r: $!!y.\ y \in A \Longrightarrow P\ (Abs\ y)$

```
  shows P x
proof −
  have Rep x ∈ A by (rule Rep)
  then have P (Abs (Rep x)) by (rule r)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  ultimately show P x by simp
qed

lemma Rep-range: range Rep = A
proof
  show range Rep <= A using Rep by (auto simp add: image-def)
  show A <= range Rep
  proof
    fix x assume x : A
    hence x = Rep (Abs x) by (rule Abs-inverse [symmetric])
    thus x : range Rep by (rule range-eqI)
  qed
qed

lemma Abs-image: Abs ‘ A = UNIV
proof
  show Abs ‘ A <= UNIV by (rule subset-UNIV)
next
  show UNIV <= Abs ‘ A
  proof
    fix x
    have x = Abs (Rep x) by (rule Rep-inverse [symmetric])
    moreover have Rep x : A by (rule Rep)
    ultimately show x : Abs ‘ A by (rule image-eqI)
  qed
qed

end

use Tools/typedef-package.ML setup TypedefPackage.setup
use Tools/typecopy-package.ML setup TypecopyPackage.setup
use Tools/typedef-codegen.ML setup TypedefCodegen.setup

end
```

7 Fun: Notions about functions

```
theory Fun
imports Set
begin
```

As a simplification rule, it replaces all function equalities by first-order equalities.

lemma *expand-fun-eq*: $f = g \longleftrightarrow (\forall x.\ f\ x = g\ x)$
apply (*rule iffI*)
apply (*simp* (*no-asm-simp*))
apply (*rule ext*)
apply (*simp* (*no-asm-simp*))
done

lemma *apply-inverse*:
$f\ x = u \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow g\ (f\ x) = x) \Longrightarrow P\ x \Longrightarrow x = g\ u$
by *auto*

7.1 The Identity Function *id*

definition
$id :: 'a \Rightarrow 'a$
where
$id = (\lambda x.\ x)$

lemma *id-apply* [*simp*]: $id\ x = x$
by (*simp add*: *id-def*)

lemma *image-ident* [*simp*]: $(\%x.\ x)\ `\ Y = Y$
by *blast*

lemma *image-id* [*simp*]: $id\ `\ Y = Y$
by (*simp add*: *id-def*)

lemma *vimage-ident* [*simp*]: $(\%x.\ x) -` Y = Y$
by *blast*

lemma *vimage-id* [*simp*]: $id -` A = A$
by (*simp add*: *id-def*)

7.2 The Composition Operator $f \circ g$

definition
$comp :: ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** *o* *55*)
where
$f\ o\ g = (\lambda x.\ f\ (g\ x))$

notation (*xsymbols*)
comp (**infixl** $\circ$ *55*)

notation (*HTML* **output**)
comp (**infixl** $\circ$ *55*)

compatibility

lemmas *o-def* = *comp-def*

lemma *o-apply* [*simp*]: $(f\ o\ g)\ x = f\ (g\ x)$
by (*simp add*: *comp-def*)

lemma *o-assoc*: $f\ o\ (g\ o\ h) = f\ o\ g\ o\ h$
by (*simp add*: *comp-def*)

lemma *id-o* [*simp*]: $id\ o\ g = g$
by (*simp add*: *comp-def*)

lemma *o-id* [*simp*]: $f\ o\ id = f$
by (*simp add*: *comp-def*)

lemma *image-compose*: $(f\ o\ g)\ `\ r = f`(g`r)$
by (*simp add*: *comp-def*, *blast*)

lemma *UN-o*: $UNION\ A\ (g\ o\ f) = UNION\ (f`A)\ g$
by (*unfold comp-def*, *blast*)

7.3 The Forward Composition Operator *fcomp*

definition
$fcomp :: ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'c$ (**infixl** $o>\ 60$)
where
$f\ o>\ g = (\lambda x.\ g\ (f\ x))$

lemma *fcomp-apply*: $(f\ o>\ g)\ x = g\ (f\ x)$
by (*simp add*: *fcomp-def*)

lemma *fcomp-assoc*: $(f\ o>\ g)\ o>\ h = f\ o>\ (g\ o>\ h)$
by (*simp add*: *fcomp-def*)

lemma *id-fcomp* [*simp*]: $id\ o>\ g = g$
by (*simp add*: *fcomp-def*)

lemma *fcomp-id* [*simp*]: $f\ o>\ id = f$
by (*simp add*: *fcomp-def*)

no-notation *fcomp* (**infixl** $o>\ 60$)

7.4 Injectivity and Surjectivity

constdefs
$inj\text{-}on :: ['a => 'b, 'a\ set] => bool$ — injective
$inj\text{-}on\ f\ A == !\ x{:}A.\ !\ y{:}A.\ f(x){=}f(y)\ --> x{=}y$

A common special case: functions injective over the entire domain type.

abbreviation
$inj\ f == inj\text{-}on\ f\ UNIV$

definition
bij-betw :: ('*a* => '*b*) => '*a set* => '*b set* => *bool* **where** — bijective
[*code del*]: *bij-betw f A B* ⟷ *inj-on f A* & *f* ' *A* = *B*

constdefs
surj :: ('*a* => '*b*) => *bool*
surj f == ! *y*. ? *x*. *y*=*f*(*x*)

bij :: ('*a* => '*b*) => *bool*
bij f == *inj f* & *surj f*

lemma *injI*:
assumes ⋀*x y*. *f x* = *f y* ⟹ *x* = *y*
shows *inj f*
using *assms* **unfolding** *inj-on-def* **by** *auto*

For Proofs in *Tools/datatype-rep-proofs*

lemma *datatype-injI*:
(!! *x*. *ALL y*. *f*(*x*) = *f*(*y*) −−> *x*=*y*) ==> *inj*(*f*)
by (*simp add*: *inj-on-def*)

theorem *range-ex1-eq*: *inj f* ⟹ *b* : *range f* = (*EX*! *x*. *b* = *f x*)
by (*unfold inj-on-def*, *blast*)

lemma *injD*: [| *inj*(*f*); *f*(*x*) = *f*(*y*) |] ==> *x*=*y*
by (*simp add*: *inj-on-def*)

lemma *inj-eq*: *inj*(*f*) ==> (*f*(*x*) = *f*(*y*)) = (*x*=*y*)
by (*force simp add*: *inj-on-def*)

lemma *inj-on-id*[*simp*]: *inj-on id A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-id2*[*simp*]: *inj-on* (%*x*. *x*) *A*
by (*simp add*: *inj-on-def*)

lemma *surj-id*[*simp*]: *surj id*
by (*simp add*: *surj-def*)

lemma *bij-id*[*simp*]: *bij id*
by (*simp add*: *bij-def inj-on-id surj-id*)

lemma *inj-onI*:
(!! *x y*. [| *x*:*A*; *y*:*A*; *f*(*x*) = *f*(*y*) |] ==> *x*=*y*) ==> *inj-on f A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-inverseI*: (!!*x*. *x*:*A* ==> *g*(*f*(*x*)) = *x*) ==> *inj-on f A*
by (*auto dest*: *arg-cong* [*of concl*: *g*] *simp add*: *inj-on-def*)

lemma *inj-onD*: [| *inj-on f A*; $f(x)=f(y)$; x:*A*; y:*A* |] ==> $x=y$
by (*unfold inj-on-def*, *blast*)

lemma *inj-on-iff*: [| *inj-on f A*; x:*A*; y:*A* |] ==> $(f(x)=f(y)) = (x=y)$
by (*blast dest*!: *inj-onD*)

lemma *comp-inj-on*:
[| *inj-on f A*; *inj-on g* (*f*'*A*) |] ==> *inj-on* (*g o f*) *A*
by (*simp add*: *comp-def inj-on-def*)

lemma *inj-on-imageI*: *inj-on* (*g o f*) *A* ⟹ *inj-on g* (*f* ' *A*)
apply(*simp add*:*inj-on-def image-def*)
apply *blast*
done

lemma *inj-on-image-iff*: ⟦ *ALL* x:*A*. *ALL* y:*A*. $(g(f\,x) = g(f\,y)) = (g\,x = g\,y)$;
inj-on f A ⟧ ⟹ *inj-on g* (*f* ' *A*) = *inj-on g A*
apply(*unfold inj-on-def*)
apply *blast*
done

lemma *inj-on-contraD*: [| *inj-on f A*; ~x=y; x:*A*; y:*A* |] ==> ~ $f(x)$=$f(y)$
by (*unfold inj-on-def*, *blast*)

lemma *inj-singleton*: *inj* (%s. {s})
by (*simp add*: *inj-on-def*)

lemma *inj-on-empty*[*iff*]: *inj-on f* {}
by(*simp add*: *inj-on-def*)

lemma *subset-inj-on*: [| *inj-on f B*; *A* <= *B* |] ==> *inj-on f A*
by (*unfold inj-on-def*, *blast*)

lemma *inj-on-Un*:
inj-on f (*A Un B*) =
(*inj-on f A* & *inj-on f B* & *f*'(*A*−*B*) *Int f*'(*B*−*A*) = {})
apply(*unfold inj-on-def*)
apply (*blast intro*:*sym*)
done

lemma *inj-on-insert*[*iff*]:
inj-on f (*insert a A*) = (*inj-on f A* & *f a* ~: *f*'(*A*−{*a*}))
apply(*unfold inj-on-def*)
apply (*blast intro*:*sym*)
done

lemma *inj-on-diff*: *inj-on f A* ==> *inj-on f* (*A*−*B*)
apply(*unfold inj-on-def*)

apply (*blast*)
done

lemma *surjI*: (!! *x*. *g*(*f x*) = *x*) ==> *surj g*
apply (*simp add*: *surj-def*)
apply (*blast intro*: *sym*)
done

lemma *surj-range*: *surj f* ==> *range f* = *UNIV*
by (*auto simp add*: *surj-def*)

lemma *surjD*: *surj f* ==> *EX x*. *y* = *f x*
by (*simp add*: *surj-def*)

lemma *surjE*: *surj f* ==> (!!*x*. *y* = *f x* ==> *C*) ==> *C*
by (*simp add*: *surj-def*, *blast*)

lemma *comp-surj*: [| *surj f*; *surj g* |] ==> *surj* (*g o f*)
apply (*simp add*: *comp-def surj-def*, *clarify*)
apply (*drule-tac x* = *y* **in** *spec*, *clarify*)
apply (*drule-tac x* = *x* **in** *spec*, *blast*)
done

lemma *bijI*: [| *inj f*; *surj f* |] ==> *bij f*
by (*simp add*: *bij-def*)

lemma *bij-is-inj*: *bij f* ==> *inj f*
by (*simp add*: *bij-def*)

lemma *bij-is-surj*: *bij f* ==> *surj f*
by (*simp add*: *bij-def*)

lemma *bij-betw-imp-inj-on*: *bij-betw f A B* ⟹ *inj-on f A*
by (*simp add*: *bij-betw-def*)

lemma *bij-betw-inv*: **assumes** *bij-betw f A B* **shows** *EX g*. *bij-betw g B A*
proof −
have *i*: *inj-on f A* **and** *s*: *f* ‘ *A* = *B*
using *assms* **by**(*auto simp*:*bij-betw-def*)
let *?P* = %*b a*. *a*:*A* ∧ *f a* = *b* **let** *?g* = %*b*. *The* (*?P b*)
{ fix *a b* **assume** *P*: *?P b a*
hence *ex1*: ∃ *a*. *?P b a* **using** *s* **unfolding** *image-def* **by** *blast*
hence *uex1*: ∃!*a*. *?P b a* **by**(*blast dest*:*inj-onD*[*OF i*])
hence *?g b* = *a* **using** *the1-equality*[*OF uex1*, *OF P*] *P* **by** *simp*
} note *g* = *this*
have *inj-on ?g B*
proof(*rule inj-onI*)
fix *x y* **assume** *x*:*B y*:*B ?g x* = *?g y*
from *s* ‹*x*:*B*› **obtain** *a1* **where** *a1*: *?P x a1* **unfolding** *image-def* **by** *blast*

```
    from s ‹y:B› obtain a2 where a2: ?P y a2 unfolding image-def by blast
    from g[OF a1] a1 g[OF a2] a2 ‹?g x = ?g y› show x=y by simp
  qed
  moreover have ?g ‘ B = A
  proof(auto simp:image-def)
    fix b assume b:B
    with s obtain a where P: ?P b a unfolding image-def by blast
    thus ?g b ∈ A using g[OF P] by auto
  next
    fix a assume a:A
    then obtain b where P: ?P b a using s unfolding image-def by blast
    then have b:B using s unfolding image-def by blast
    with g[OF P] show ∃ b∈B. a = ?g b by blast
  qed
  ultimately show ?thesis by(auto simp:bij-betw-def)
qed

lemma surj-image-vimage-eq: surj f ==> f ‘ (f −‘ A) = A
by (simp add: surj-range)

lemma inj-vimage-image-eq: inj f ==> f −‘ (f ‘ A) = A
by (simp add: inj-on-def, blast)

lemma vimage-subsetD: surj f ==> f −‘ B <= A ==> B <= f ‘ A
apply (unfold surj-def)
apply (blast intro: sym)
done

lemma vimage-subsetI: inj f ==> B <= f ‘ A ==> f −‘ B <= A
by (unfold inj-on-def, blast)

lemma vimage-subset-eq: bij f ==> (f −‘ B <= A) = (B <= f ‘ A)
apply (unfold bij-def)
apply (blast del: subsetI intro: vimage-subsetI vimage-subsetD)
done

lemma inj-on-image-Int:
   [| inj-on f C;  A<=C;  B<=C |] ==> f‘(A Int B) = f‘A Int f‘B
apply (simp add: inj-on-def, blast)
done

lemma inj-on-image-set-diff:
   [| inj-on f C;  A<=C;  B<=C |] ==> f‘(A−B) = f‘A − f‘B
apply (simp add: inj-on-def, blast)
done

lemma image-Int: inj f ==> f‘(A Int B) = f‘A Int f‘B
by (simp add: inj-on-def, blast)
```

lemma *image-set-diff*: *inj f ==> f‘(A−B) = f‘A − f‘B*
by (*simp add*: *inj-on-def*, *blast*)

lemma *inj-image-mem-iff*: *inj f ==> (f a : f‘A) = (a : A)*
by (*blast dest*: *injD*)

lemma *inj-image-subset-iff*: *inj f ==> (f‘A <= f‘B) = (A<=B)*
by (*simp add*: *inj-on-def*, *blast*)

lemma *inj-image-eq-iff*: *inj f ==> (f‘A = f‘B) = (A = B)*
by (*blast dest*: *injD*)

lemma *image-INT*:
[| *inj-on f C*; *ALL x:A. B x <= C*; *j:A* |]
==> *f ‘ (INTER A B) = (INT x:A. f ‘ B x)*
apply (*simp add*: *inj-on-def*, *blast*)
done

lemma *bij-image-INT*: *bij f ==> f ‘ (INTER A B) = (INT x:A. f ‘ B x)*
apply (*simp add*: *bij-def*)
apply (*simp add*: *inj-on-def surj-def*, *blast*)
done

lemma *surj-Compl-image-subset*: *surj f ==> −(f‘A) <= f‘(−A)*
by (*auto simp add*: *surj-def*)

lemma *inj-image-Compl-subset*: *inj f ==> f‘(−A) <= −(f‘A)*
by (*auto simp add*: *inj-on-def*)

lemma *bij-image-Compl-eq*: *bij f ==> f‘(−A) = −(f‘A)*
apply (*simp add*: *bij-def*)
apply (*rule equalityI*)
apply (*simp-all* (*no-asm-simp*) *add*: *inj-image-Compl-subset surj-Compl-image-subset*)
done

7.5 Function Updating

constdefs
fun-upd :: (*'a => 'b*) => *'a => 'b => ('a => 'b*)
fun-upd f a b == % x. if x=a then b else f x

nonterminals
updbinds updbind
syntax
-*updbind* :: [*'a*, *'a*] => *updbind* ((*2*- :=/ -))
:: *updbind* => *updbinds* (-)
-*updbinds*:: [*updbind*, *updbinds*] => *updbinds* (-,/ -)

-Update :: ['*a*, *updbinds*] => '*a* (-/'((-)') [*1000*,*0*] *900*)

translations
-Update f (*-updbinds b bs*) == *-Update* (*-Update f b*) *bs*
f(*x*:=*y*) == *fun-upd f x y*

lemma *fun-upd-idem-iff*: (*f*(*x*:=*y*) = *f*) = (*f x* = *y*)
apply (*simp add*: *fun-upd-def*, *safe*)
apply (*erule subst*)
apply (*rule-tac* [*2*] *ext*, *auto*)
done

lemmas *fun-upd-idem* = *fun-upd-idem-iff* [*THEN iffD2*, *standard*]

lemmas *fun-upd-triv* = *refl* [*THEN fun-upd-idem*]
declare *fun-upd-triv* [*iff*]

lemma *fun-upd-apply* [*simp*]: (*f*(*x*:=*y*))*z* = (*if z*=*x then y else f z*)
by (*simp add*: *fun-upd-def*)

lemma *fun-upd-same*: (*f*(*x*:=*y*)) *x* = *y*
by *simp*

lemma *fun-upd-other*: *z*~=*x* ==> (*f*(*x*:=*y*)) *z* = *f z*
by *simp*

lemma *fun-upd-upd* [*simp*]: *f*(*x*:=*y*,*x*:=*z*) = *f*(*x*:=*z*)
by (*simp add*: *expand-fun-eq*)

lemma *fun-upd-twist*: *a* ~= *c* ==> (*m*(*a*:=*b*))(*c*:=*d*) = (*m*(*c*:=*d*))(*a*:=*b*)
by (*rule ext*, *auto*)

lemma *inj-on-fun-updI*: ⟦ *inj-on f A*; *y* ∉ *f*‘*A* ⟧ ⟹ *inj-on* (*f*(*x*:=*y*)) *A*
by(*fastsimp simp*:*inj-on-def image-def*)

lemma *fun-upd-image*:
f(*x*:=*y*) ‘ *A* = (*if x* ∈ *A then insert y* (*f* ‘ (*A*−{*x*})) *else f* ‘ *A*)
by *auto*

7.6 *override-on*

definition
override-on :: ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*a set* ⇒ '*a* ⇒ '*b*
where

override-on f g A = (λa. if a ∈ A then g a else f a)

lemma *override-on-emptyset*[*simp*]: *override-on f g {} = f*
by(*simp add*:*override-on-def*)

lemma *override-on-apply-notin*[*simp*]: *a ~: A ==> (override-on f g A) a = f a*
by(*simp add*:*override-on-def*)

lemma *override-on-apply-in*[*simp*]: *a : A ==> (override-on f g A) a = g a*
by(*simp add*:*override-on-def*)

7.7 *swap*

definition
swap :: 'a ⇒ 'a ⇒ ('a ⇒ 'b) ⇒ ('a ⇒ 'b)
where
swap a b f = f (a := f b, b:= f a)

lemma *swap-self*: *swap a a f = f*
by (*simp add*: *swap-def*)

lemma *swap-commute*: *swap a b f = swap b a f*
by (*rule ext*, *simp add*: *fun-upd-def swap-def*)

lemma *swap-nilpotent* [*simp*]: *swap a b (swap a b f) = f*
by (*rule ext*, *simp add*: *fun-upd-def swap-def*)

lemma *inj-on-imp-inj-on-swap*:
[|inj-on f A; a ∈ A; b ∈ A|] ==> inj-on (swap a b f) A
by (*simp add*: *inj-on-def swap-def*, *blast*)

lemma *inj-on-swap-iff* [*simp*]:
assumes *A*: *a ∈ A b ∈ A* **shows** *inj-on (swap a b f) A = inj-on f A*
proof
assume *inj-on (swap a b f) A*
with *A* **have** *inj-on (swap a b (swap a b f)) A*
by (*iprover intro*: *inj-on-imp-inj-on-swap*)
thus *inj-on f A* **by** *simp*
next
assume *inj-on f A*
with *A* **show** *inj-on (swap a b f) A* **by**(*iprover intro*: *inj-on-imp-inj-on-swap*)
qed

lemma *surj-imp-surj-swap*: *surj f ==> surj (swap a b f)*
apply (*simp add*: *surj-def swap-def*, *clarify*)
apply (*case-tac y = f b*, *blast*)
apply (*case-tac y = f a*, *auto*)
done

lemma *surj-swap-iff* [*simp*]: *surj* (*swap a b f*) = *surj f*
proof
assume *surj* (*swap a b f*)
hence *surj* (*swap a b* (*swap a b f*)) **by** (*rule surj-imp-surj-swap*)
thus *surj f* **by** *simp*
next
assume *surj f*
thus *surj* (*swap a b f*) **by** (*rule surj-imp-surj-swap*)
qed

lemma *bij-swap-iff*: *bij* (*swap a b f*) = *bij f*
by (*simp add*: *bij-def*)

hide (**open**) *const swap*

7.8 Proof tool setup

simplifies terms of the form f(...,x:=y,...,x:=z,...) to f(...,x:=z,...)

simproc-setup *fun-upd2* (*f*(*v* := *w*, *x* := *y*)) = ⟪ *fn* - =>
let
fun gen-fun-upd NONE T - - = *NONE*
| *gen-fun-upd* (*SOME f*) *T x y* = *SOME* (*Const* (@{*const-name fun-upd*}, *T*)
$ *f* $ *x* $ *y*)
fun dest-fun-T1 (*Type* (-, *T* :: *Ts*)) = *T*
fun find-double (*t as Const* (@{*const-name fun-upd*},*T*) $ *f* $ *x* $ *y*) =
let
fun find (*Const* (@{*const-name fun-upd*},*T*) $ *g* $ *v* $ *w*) =
if v aconv x then SOME g else gen-fun-upd (*find g*) *T v w*
| *find t* = *NONE*
in (*dest-fun-T1 T*, *gen-fun-upd* (*find f*) *T x y*) *end*

fun proc ss ct =
let
val ctxt = *Simplifier.the-context ss*
val t = *Thm.term-of ct*
in
case find-double t of
(*T*, *NONE*) => *NONE*
| (*T*, *SOME rhs*) =>
SOME (*Goal.prove ctxt* [] [] (*Logic.mk-equals* (*t*, *rhs*))
(*fn* - =>
rtac eq-reflection 1 THEN
rtac ext 1 THEN
simp-tac (*Simplifier.inherit-context ss* @{*simpset*}) *1*))
end
in proc end
⟫

7.9 Code generator setup

types-code
fun ((- −>/ -))
attach (*term-of*) ⟪
fun term-of-fun-type - aT - bT - = Free (<*function*>, *aT* −−> *bT*);
⟫
attach (*test*) ⟪
fun gen-fun-type aF aT bG bT i =
let
val tab = ref [];
fun mk-upd (*x*, (-, *y*)) *t = Const* (*Fun.fun-upd*,
(*aT* −−> *bT*) −−> *aT* −−> *bT* −−> *aT* −−> *bT*) $ *t* $ *aF x* $ *y* ()
in
(*fn x =>*
case ALList.lookup op = (!*tab*) *x of*
NONE =>
let val p as (*y*, -) = *bG i*
in (*tab* := (*x*, *p*) :: !*tab*; *y*) *end*
| *SOME* (*y*, -) => *y*,
fn () => *Basics.fold mk-upd* (!*tab*) (*Const* (*HOL.undefined*, *aT* −−> *bT*)))
end;
⟫

code-const *op* ∘
(*SML* **infixl** *5 o*)
(*Haskell* **infixr** *9* .)

code-const *id*
(*Haskell id*)

end

8 Sum-Type: The Disjoint Sum of Two Types

theory *Sum-Type*
imports *Typedef Fun*
begin

The representations of the two injections

constdefs
Inl-Rep :: [*'a*, *'a*, *'b*, *bool*] => *bool*
Inl-Rep == (%*a*. %*x y p*. *x*=*a* & *p*)

Inr-Rep :: [*'b*, *'a*, *'b*, *bool*] => *bool*
Inr-Rep == (%*b*. %*x y p*. *y*=*b* & ~*p*)

global

typedef (*Sum*)
(*'a*, *'b*) + (**infixr** + *10*)
= {*f*. (*? a*. *f* = *Inl-Rep*(*a*::*'a*)) | (*? b*. *f* = *Inr-Rep*(*b*::*'b*))}
by *auto*

local

abstract constants and syntax

constdefs
Inl :: *'a* => *'a* + *'b*
Inl == (%*a*. *Abs-Sum*(*Inl-Rep*(*a*)))

Inr :: *'b* => *'a* + *'b*
Inr == (%*b*. *Abs-Sum*(*Inr-Rep*(*b*)))

Plus :: [*'a set*, *'b set*] => (*'a* + *'b*) *set* (**infixr** <+> *65*)
A <+> *B* == (*Inl*‘*A*) *Un* (*Inr*‘*B*)
— disjoint sum for sets; the operator + is overloaded with wrong type!

Part :: [*'a set*, *'b* => *'a*] => *'a set*
Part A h == *A Int* {*x*. *? z*. *x* = *h*(*z*)}
— for selecting out the components of a mutually recursive definition

lemma *Inl-RepI*: *Inl-Rep*(*a*) : *Sum*
by (*auto simp add*: *Sum-def*)

lemma *Inr-RepI*: *Inr-Rep*(*b*) : *Sum*
by (*auto simp add*: *Sum-def*)

lemma *inj-on-Abs-Sum*: *inj-on Abs-Sum Sum*
apply (*rule inj-on-inverseI*)
apply (*erule Abs-Sum-inverse*)
done

8.1 Freeness Properties for *Inl* and *Inr*

Distinctness

lemma *Inl-Rep-not-Inr-Rep*: *Inl-Rep*(*a*) ~= *Inr-Rep*(*b*)
by (*auto simp add*: *Inl-Rep-def Inr-Rep-def expand-fun-eq*)

lemma *Inl-not-Inr* [*iff*]: *Inl*(*a*) ~= *Inr*(*b*)
apply (*simp add*: *Inl-def Inr-def*)

apply (*rule inj-on-Abs-Sum* [*THEN inj-on-contraD*])
apply (*rule Inl-Rep-not-Inr-Rep*)
apply (*rule Inl-RepI*)
apply (*rule Inr-RepI*)
done

lemmas *Inr-not-Inl* = *Inl-not-Inr* [*THEN not-sym*, *standard*]
declare *Inr-not-Inl* [*iff*]

lemmas *Inl-neq-Inr* = *Inl-not-Inr* [*THEN notE*, *standard*]
lemmas *Inr-neq-Inl* = *sym* [*THEN Inl-neq-Inr*, *standard*]

Injectiveness

lemma *Inl-Rep-inject*: *Inl-Rep*(*a*) = *Inl-Rep*(*c*) ==> *a*=*c*
by (*auto simp add*: *Inl-Rep-def expand-fun-eq*)

lemma *Inr-Rep-inject*: *Inr-Rep*(*b*) = *Inr-Rep*(*d*) ==> *b*=*d*
by (*auto simp add*: *Inr-Rep-def expand-fun-eq*)

lemma *inj-Inl* [*simp*]: *inj-on Inl A*
apply (*simp add*: *Inl-def*)
apply (*rule inj-onI*)
apply (*erule inj-on-Abs-Sum* [*THEN inj-onD*, *THEN Inl-Rep-inject*])
apply (*rule Inl-RepI*)
apply (*rule Inl-RepI*)
done

lemmas *Inl-inject* = *inj-Inl* [*THEN injD*, *standard*]

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
apply (*simp add*: *Inr-def*)
apply (*rule inj-onI*)
apply (*erule inj-on-Abs-Sum* [*THEN inj-onD*, *THEN Inr-Rep-inject*])
apply (*rule Inr-RepI*)
apply (*rule Inr-RepI*)
done

lemmas *Inr-inject* = *inj-Inr* [*THEN injD*, *standard*]

lemma *Inl-eq* [*iff*]: (*Inl*(*x*)=*Inl*(*y*)) = (*x*=*y*)
by (*blast dest*!: *Inl-inject*)

lemma *Inr-eq* [*iff*]: (*Inr*(*x*)=*Inr*(*y*)) = (*x*=*y*)
by (*blast dest*!: *Inr-inject*)

8.2 The Disjoint Sum of Sets

lemma *InlI* [*intro*!]: *a* : *A* ==> *Inl*(*a*) : *A* <+> *B*
by (*simp add*: *Plus-def*)

lemma *InrI* [*intro*!]: $b : B \Longrightarrow Inr(b) : A <+> B$
by (*simp add*: *Plus-def*)

lemma *PlusE* [*elim*!]:
$[\!| u: A <+> B;$
$\quad !!x.\ [\!| x{:}A;\ u{=}Inl(x) |\!] \Longrightarrow P;$
$\quad !!y.\ [\!| y{:}B;\ u{=}Inr(y) |\!] \Longrightarrow P$
$|\!] \Longrightarrow P$
by (*auto simp add*: *Plus-def*)

Exhaustion rule for sums, a degenerate form of induction

lemma *sumE*:
$[\!| !!x{::}'a.\ s = Inl(x) \Longrightarrow P;\ !!y{::}'b.\ s = Inr(y) \Longrightarrow P$
$|\!] \Longrightarrow P$
apply (*rule Abs-Sum-cases* [*of s*])
apply (*auto simp add*: *Sum-def Inl-def Inr-def*)
done

lemma *UNIV-Plus-UNIV* [*simp*]: $UNIV <+> UNIV = UNIV$
apply (*rule set-ext*)
apply(*rename-tac s*)
apply(*rule-tac s=s* **in** *sumE*)
apply *auto*
done

8.3 The *Part* Primitive

lemma *Part-eqI* [*intro*]: $[\!| a : A;\ a{=}h(b) |\!] \Longrightarrow a : Part\ A\ h$
by (*auto simp add*: *Part-def*)

lemmas *PartI* = *Part-eqI* [*OF - refl*, *standard*]

lemma *PartE* [*elim*!]: $[\!| a : Part\ A\ h;\ !!z.\ [\!| a : A;\ a{=}h(z) |\!] \Longrightarrow P |\!] \Longrightarrow P$
by (*auto simp add*: *Part-def*)

lemma *Part-subset*: $Part\ A\ h <= A$
by (*auto simp add*: *Part-def*)

lemma *Part-mono*: $A{<=}B \Longrightarrow Part\ A\ h <= Part\ B\ h$
by *blast*

lemmas *basic-monos* = *basic-monos Part-mono*

lemma *PartD1*: $a : Part\ A\ h \Longrightarrow a : A$

by (*simp add*: *Part-def*)

lemma *Part-id*: *Part A* (%*x*. *x*) = *A*
by *blast*

lemma *Part-Int*: *Part* (*A Int B*) *h* = (*Part A h*) *Int* (*Part B h*)
by *blast*

lemma *Part-Collect*: *Part* (*A Int* {*x*. *P x*}) *h* = (*Part A h*) *Int* {*x*. *P x*}
by *blast*

ML
⟪
val Inl-RepI = *thm Inl-RepI*;
val Inr-RepI = *thm Inr-RepI*;
val inj-on-Abs-Sum = *thm inj-on-Abs-Sum*;
val Inl-Rep-not-Inr-Rep = *thm Inl-Rep-not-Inr-Rep*;
val Inl-not-Inr = *thm Inl-not-Inr*;
val Inr-not-Inl = *thm Inr-not-Inl*;
val Inl-neq-Inr = *thm Inl-neq-Inr*;
val Inr-neq-Inl = *thm Inr-neq-Inl*;
val Inl-Rep-inject = *thm Inl-Rep-inject*;
val Inr-Rep-inject = *thm Inr-Rep-inject*;
val inj-Inl = *thm inj-Inl*;
val Inl-inject = *thm Inl-inject*;
val inj-Inr = *thm inj-Inr*;
val Inr-inject = *thm Inr-inject*;
val Inl-eq = *thm Inl-eq*;
val Inr-eq = *thm Inr-eq*;
val InlI = *thm InlI*;
val InrI = *thm InrI*;
val PlusE = *thm PlusE*;
val sumE = *thm sumE*;
val Part-eqI = *thm Part-eqI*;
val PartI = *thm PartI*;
val PartE = *thm PartE*;
val Part-subset = *thm Part-subset*;
val Part-mono = *thm Part-mono*;
val PartD1 = *thm PartD1*;
val Part-id = *thm Part-id*;
val Part-Int = *thm Part-Int*;
val Part-Collect = *thm Part-Collect*;

val basic-monos = *thms basic-monos*;
⟫

end

9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*
imports *Lattices Sum-Type*
uses
(*Tools/inductive-package.ML*)
Tools/dseq.ML
(*Tools/inductive-codegen.ML*)
(*Tools/datatype-aux.ML*)
(*Tools/datatype-prop.ML*)
(*Tools/datatype-rep-proofs.ML*)
(*Tools/datatype-abs-proofs.ML*)
(*Tools/datatype-case.ML*)
(*Tools/datatype-package.ML*)
(*Tools/old-primrec-package.ML*)
(*Tools/primrec-package.ML*)
(*Tools/datatype-codegen.ML*)
begin

9.1 Least and greatest fixed points

context *complete-lattice*
begin

definition
lfp :: $('a \Rightarrow 'a) \Rightarrow 'a$ **where**
lfp f = *Inf* $\{u.\ f\ u \leq u\}$ — least fixed point

definition
gfp :: $('a \Rightarrow 'a) \Rightarrow 'a$ **where**
gfp f = *Sup* $\{u.\ u \leq f\ u\}$ — greatest fixed point

9.2 Proof of Knaster-Tarski Theorem using *lfp*

lfp f is the least upper bound of the set $\{u.\ f\ u \leq u\}$

lemma *lfp-lowerbound*: *f A* $\leq$ *A* ==> *lfp f* $\leq$ *A*
by (*auto simp add*: *lfp-def intro*: *Inf-lower*)

lemma *lfp-greatest*: (!!*u*. *f u* $\leq$ *u* ==> *A* $\leq$ *u*) ==> *A* $\leq$ *lfp f*
by (*auto simp add*: *lfp-def intro*: *Inf-greatest*)

end

lemma *lfp-lemma2*: *mono f* ==> *f* (*lfp f*) $\leq$ *lfp f*
by (*iprover intro*: *lfp-greatest order-trans monoD lfp-lowerbound*)

lemma *lfp-lemma3*: *mono f* ==> *lfp f* $\leq$ *f* (*lfp f*)
by (*iprover intro*: *lfp-lemma2 monoD lfp-lowerbound*)

lemma *lfp-unfold*: *mono f ==> lfp f = f (lfp f)*
by (*iprover intro*: *order-antisym lfp-lemma2 lfp-lemma3*)

lemma *lfp-const*: *lfp* ($\lambda x.\ t$) = *t*
by (*rule lfp-unfold*) (*simp add*:*mono-def*)

9.3 General induction rules for least fixed points

theorem *lfp-induct*:
assumes *mono*: *mono f* **and** *ind*: *f* (*inf* (*lfp f*) *P*) <= *P*
shows *lfp f* <= *P*
proof −
have *inf* (*lfp f*) *P* <= *lfp f* **by** (*rule inf-le1*)
with *mono* **have** *f* (*inf* (*lfp f*) *P*) <= *f* (*lfp f*) **..**
also from *mono* **have** *f* (*lfp f*) = *lfp f* **by** (*rule lfp-unfold* [*symmetric*])
finally have *f* (*inf* (*lfp f*) *P*) <= *lfp f* **.**
from *this* **and** *ind* **have** *f* (*inf* (*lfp f*) *P*) <= *inf* (*lfp f*) *P* **by** (*rule le-infI*)
hence *lfp f* <= *inf* (*lfp f*) *P* **by** (*rule lfp-lowerbound*)
also have *inf* (*lfp f*) *P* <= *P* **by** (*rule inf-le2*)
finally show *?thesis* **.**
qed

lemma *lfp-induct-set*:
assumes *lfp*: *a*: *lfp*(*f*)
and *mono*: *mono*(*f*)
and *indhyp*: !!*x*. [| *x*: *f*(*lfp*(*f*) *Int* {*x*. *P*(*x*)}) |] ==> *P*(*x*)
shows *P*(*a*)
by (*rule lfp-induct* [*THEN subsetD*, *THEN CollectD*, *OF mono - lfp*])
(*auto simp*: *inf-set-eq intro*: *indhyp*)

lemma *lfp-ordinal-induct*:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'a*
assumes *mono*: *mono f*
and *P-f*: ⋀*S*. *P S* ⟹ *P* (*f S*)
and *P-Union*: ⋀*M*. ∀ *S*∈*M*. *P S* ⟹ *P* (*Sup M*)
shows *P* (*lfp f*)
proof −
let *?M* = {*S*. *S* ≤ *lfp f* ∧ *P S*}
have *P* (*Sup ?M*) **using** *P-Union* **by** *simp*
also have *Sup ?M* = *lfp f*
proof (*rule antisym*)
show *Sup ?M* ≤ *lfp f* **by** (*blast intro*: *Sup-least*)
hence *f* (*Sup ?M*) ≤ *f* (*lfp f*) **by** (*rule mono* [*THEN monoD*])
hence *f* (*Sup ?M*) ≤ *lfp f* **using** *mono* [*THEN lfp-unfold*] **by** *simp*
hence *f* (*Sup ?M*) ∈ *?M* **using** *P-f P-Union* **by** *simp*
hence *f* (*Sup ?M*) ≤ *Sup ?M* **by** (*rule Sup-upper*)
thus *lfp f* ≤ *Sup ?M* **by** (*rule lfp-lowerbound*)
qed

finally show *?thesis* **.**
qed

lemma *lfp-ordinal-induct-set*:
assumes *mono*: *mono f*
and *P-f*: !!*S*. *P S* ==> *P*(*f S*)
and *P-Union*: !!*M*. !*S*:*M*. *P S* ==> *P*(*Union M*)
shows *P*(*lfp f*)
using *assms* **unfolding** *Sup-set-eq* [*symmetric*]
by (*rule lfp-ordinal-induct* [**where** *P*=*P*])

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding

lemma *def-lfp-unfold*: [| *h*==*lfp*(*f*); *mono*(*f*) |] ==> *h* = *f*(*h*)
by (*auto intro*!: *lfp-unfold*)

lemma *def-lfp-induct*:
[| *A* == *lfp*(*f*); *mono*(*f*);
f (*inf A P*) ≤ *P*
|] ==> *A* ≤ *P*
by (*blast intro*: *lfp-induct*)

lemma *def-lfp-induct-set*:
[| *A* == *lfp*(*f*); *mono*(*f*); *a*:*A*;
!!*x*. [| *x*: *f*(*A Int* {*x*. *P*(*x*)}) |] ==> *P*(*x*)
|] ==> *P*(*a*)
by (*blast intro*: *lfp-induct-set*)

lemma *lfp-mono*: (!!*Z*. *f Z* ≤ *g Z*) ==> *lfp f* ≤ *lfp g*
by (*rule lfp-lowerbound* [*THEN lfp-greatest*], *blast intro*: *order-trans*)

9.4 Proof of Knaster-Tarski Theorem using *gfp*

gfp f is the greatest lower bound of the set {*u*. *u* ≤ *f u*}

lemma *gfp-upperbound*: *X* ≤ *f X* ==> *X* ≤ *gfp f*
by (*auto simp add*: *gfp-def intro*: *Sup-upper*)

lemma *gfp-least*: (!!*u*. *u* ≤ *f u* ==> *u* ≤ *X*) ==> *gfp f* ≤ *X*
by (*auto simp add*: *gfp-def intro*: *Sup-least*)

lemma *gfp-lemma2*: *mono f* ==> *gfp f* ≤ *f* (*gfp f*)
by (*iprover intro*: *gfp-least order-trans monoD gfp-upperbound*)

lemma *gfp-lemma3*: *mono f* ==> *f* (*gfp f*) ≤ *gfp f*
by (*iprover intro*: *gfp-lemma2 monoD gfp-upperbound*)

lemma *gfp-unfold*: *mono f* ==> *gfp f* = *f* (*gfp f*)
by (*iprover intro*: *order-antisym gfp-lemma2 gfp-lemma3*)

9.5 Coinduction rules for greatest fixed points

weak version

lemma *weak-coinduct*: [| *a*: *X*; *X* ⊆ *f*(*X*) |] ==> *a* : *gfp*(*f*)
by (*rule gfp-upperbound* [*THEN subsetD*], *auto*)

lemma *weak-coinduct-image*: !!*X*. [| *a* : *X*; *g*‘*X* ⊆ *f* (*g*‘*X*) |] ==> *g a* : *gfp f*
apply (*erule gfp-upperbound* [*THEN subsetD*])
apply (*erule imageI*)
done

lemma *coinduct-lemma*:
[| *X* ≤ *f* (*sup X* (*gfp f*)); *mono f* |] ==> *sup X* (*gfp f*) ≤ *f* (*sup X* (*gfp f*))
apply (*frule gfp-lemma2*)
apply (*drule mono-sup*)
apply (*rule le-supI*)
apply *assumption*
apply (*rule order-trans*)
apply (*rule order-trans*)
apply *assumption*
apply (*rule sup-ge2*)
apply *assumption*
done

strong version, thanks to Coen and Frost

lemma *coinduct-set*: [| *mono*(*f*); *a*: *X*; *X* ⊆ *f*(*X Un gfp*(*f*)) |] ==> *a* : *gfp*(*f*)
by (*blast intro*: *weak-coinduct* [*OF - coinduct-lemma*, *simplified sup-set-eq*])

lemma *coinduct*: [| *mono*(*f*); *X* ≤ *f* (*sup X* (*gfp f*)) |] ==> *X* ≤ *gfp*(*f*)
apply (*rule order-trans*)
apply (*rule sup-ge1*)
apply (*erule gfp-upperbound* [*OF coinduct-lemma*])
apply *assumption*
done

lemma *gfp-fun-UnI2*: [| *mono*(*f*); *a*: *gfp*(*f*) |] ==> *a*: *f*(*X Un gfp*(*f*))
by (*blast dest*: *gfp-lemma2 mono-Un*)

9.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition *X* ⊆ *f X* to one expressed using both *lfp* and *gfp*

lemma *coinduct3-mono-lemma*: *mono*(*f*) ==> *mono*(%*x*. *f*(*x*) *Un X Un B*)
by (*iprover intro*: *subset-refl monoI Un-mono monoD*)

lemma *coinduct3-lemma*:
[| *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*))); *mono*(*f*) |]
==> *lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*)) ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*)))
apply (*rule subset-trans*)

apply (*erule coinduct3-mono-lemma* [*THEN lfp-lemma3*])
apply (*rule Un-least* [*THEN Un-least*])
apply (*rule subset-refl*, *assumption*)
apply (*rule gfp-unfold* [*THEN equalityD1*, *THEN subset-trans*], *assumption*)
apply (*rule monoD* [**where** *f*=*f*], *assumption*)
apply (*subst coinduct3-mono-lemma* [*THEN lfp-unfold*], *auto*)
done

lemma *coinduct3*:
[| *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un gfp*(*f*))) |] ==> *a* : *gfp*(*f*)
apply (*rule coinduct3-lemma* [*THEN* [*2*] *weak-coinduct*])
apply (*rule coinduct3-mono-lemma* [*THEN lfp-unfold*, *THEN ssubst*], *auto*)
done

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding

lemma *def-gfp-unfold*: [| *A*==*gfp*(*f*); *mono*(*f*) |] ==> *A* = *f*(*A*)
by (*auto intro*!: *gfp-unfold*)

lemma *def-coinduct*:
[| *A*==*gfp*(*f*); *mono*(*f*); *X* ≤ *f*(*sup X A*) |] ==> *X* ≤ *A*
by (*iprover intro*!: *coinduct*)

lemma *def-coinduct-set*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*X Un A*) |] ==> *a*: *A*
by (*auto intro*!: *coinduct-set*)

lemma *def-Collect-coinduct*:
[| *A* == *gfp*(%*w*. *Collect*(*P*(*w*))); *mono*(%*w*. *Collect*(*P*(*w*)));
a: *X*; !!*z*. *z*: *X* ==> *P* (*X Un A*) *z* |] ==>
a : *A*
apply (*erule def-coinduct-set*, *auto*)
done

lemma *def-coinduct3*:
[| *A*==*gfp*(*f*); *mono*(*f*); *a*:*X*; *X* ⊆ *f*(*lfp*(%*x*. *f*(*x*) *Un X Un A*)) |] ==> *a*: *A*
by (*auto intro*!: *coinduct3*)

Monotonicity of *gfp*!

lemma *gfp-mono*: (!!*Z*. *f Z* ≤ *g Z*) ==> *gfp f* ≤ *gfp g*
by (*rule gfp-upperbound* [*THEN gfp-least*], *blast intro*: *order-trans*)

9.7 Inductive predicates and sets

Inversion of injective functions.

constdefs
myinv :: ('*a* => '*b*) => ('*b* => '*a*)
myinv (*f* :: '*a* => '*b*) == λ*y*. *THE x*. *f x* = *y*

lemma *myinv-f-f*: *inj f* ==> *myinv f* (*f x*) = *x*
proof −
assume *inj f*
hence (*THE x'*. *f x'* = *f x*) = (*THE x'*. *x'* = *x*)
by (*simp only*: *inj-eq*)
also have ... = *x* **by** (*rule the-eq-trivial*)
finally show *?thesis* **by** (*unfold myinv-def*)
qed

lemma *f-myinv-f*: *inj f* ==> *y* ∈ *range f* ==> *f* (*myinv f y*) = *y*
proof (*unfold myinv-def*)
assume *inj*: *inj f*
assume *y* ∈ *range f*
then obtain *x* **where** *y* = *f x* **..**
hence *x*: *f x* = *y* **..**
thus *f* (*THE x*. *f x* = *y*) = *y*
proof (*rule theI*)
fix *x'* **assume** *f x'* = *y*
with *x* **have** *f x'* = *f x* **by** *simp*
with *inj* **show** *x'* = *x* **by** (*rule injD*)
qed
qed

hide *const myinv*

Package setup.

theorems *basic-monos* =
subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
Collect-mono in-mono vimage-mono
imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
not-all not-ex
Ball-def Bex-def
induct-rulify-fallback

ML ⟪
val def-lfp-unfold = @{*thm def-lfp-unfold*}
val def-gfp-unfold = @{*thm def-gfp-unfold*}
val def-lfp-induct = @{*thm def-lfp-induct*}
val def-coinduct = @{*thm def-coinduct*}
val inf-bool-eq = @{*thm inf-bool-eq*} *RS* @{*thm eq-reflection*}
val inf-fun-eq = @{*thm inf-fun-eq*} *RS* @{*thm eq-reflection*}
val sup-bool-eq = @{*thm sup-bool-eq*} *RS* @{*thm eq-reflection*}
val sup-fun-eq = @{*thm sup-fun-eq*} *RS* @{*thm eq-reflection*}
val le-boolI = @{*thm le-boolI*}
val le-boolI' = @{*thm le-boolI'*}
val le-funI = @{*thm le-funI*}
val le-boolE = @{*thm le-boolE*}
val le-funE = @{*thm le-funE*}

val le-boolD = @{thm le-boolD}
val le-funD = @{thm le-funD}
val le-bool-def = @{thm le-bool-def} RS @{thm eq-reflection}
val le-fun-def = @{thm le-fun-def} RS @{thm eq-reflection}
⟩⟩

use *Tools/inductive-package.ML*
setup *InductivePackage.setup*

theorems [*mono*] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-conv-disj not-not de-Morgan-disj de-Morgan-conj
not-all not-ex
Ball-def Bex-def
induct-rulify-fallback

9.8 Inductive datatypes and primitive recursion

Package setup.

use *Tools/datatype-aux.ML*
use *Tools/datatype-prop.ML*
use *Tools/datatype-rep-proofs.ML*
use *Tools/datatype-abs-proofs.ML*
use *Tools/datatype-case.ML*
use *Tools/datatype-package.ML*
setup *DatatypePackage.setup*
use *Tools/old-primrec-package.ML*
use *Tools/primrec-package.ML*

use *Tools/datatype-codegen.ML*
setup *DatatypeCodegen.setup*

use *Tools/inductive-codegen.ML*
setup *InductiveCodegen.setup*

Lambda-abstractions with pattern matching:

syntax
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((%-) *10*)
syntax (*xsymbols*)
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((λ-) *10*)

parse-translation (**advanced**) ⟪
let
fun fun-tr ctxt [*cs*] =
let
val x = Free (Name.variant (Term.add-free-names cs []*) x, dummyT)*;
val ft = DatatypeCase.case-tr true DatatypePackage.datatype-of-constr
ctxt [*x*, *cs*]
in lambda x ft end

in [(*-lam-pats-syntax*, *fun-tr*)] *end*
⟫

end

10 OrderedGroup: Ordered Groups

theory *OrderedGroup*
imports *Lattices*
uses ~~/*src*/*Provers*/*Arith*/*abel-cancel.ML*
begin

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

ML⟪
structure AlgebraSimps =
  NamedThmsFun(*val name* = *algebra-simps*
    val description = *algebra simplification rules*);
⟫

setup *AlgebraSimps.setup*

The rewrites accumulated in *algebra-simps* deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by *field-simps*.

10.1 Semigroups and Monoids

class *semigroup-add* = *plus* +
  assumes *add-assoc*[*algebra-simps*]: $(a + b) + c = a + (b + c)$

class *ab-semigroup-add* = *semigroup-add* +

assumes *add-commute*[*algebra-simps*]: $a + b = b + a$
begin

lemma *add-left-commute*[*algebra-simps*]: $a + (b + c) = b + (a + c)$
by (*rule mk-left-commute* [*of plus*, *OF add-assoc add-commute*])

theorems *add-ac* = *add-assoc add-commute add-left-commute*

end

theorems *add-ac* = *add-assoc add-commute add-left-commute*

class *semigroup-mult* = *times* +
assumes *mult-assoc*[*algebra-simps*]: $(a * b) * c = a * (b * c)$

class *ab-semigroup-mult* = *semigroup-mult* +
assumes *mult-commute*[*algebra-simps*]: $a * b = b * a$
begin

lemma *mult-left-commute*[*algebra-simps*]: $a * (b * c) = b * (a * c)$
by (*rule mk-left-commute* [*of times*, *OF mult-assoc mult-commute*])

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

end

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

class *ab-semigroup-idem-mult* = *ab-semigroup-mult* +
assumes *mult-idem*[*simp*]: $x * x = x$
begin

lemma *mult-left-idem*[*simp*]: $x * (x * y) = x * y$
unfolding *mult-assoc* [*symmetric*, *of x*] *mult-idem* **..**

end

class *monoid-add* = *zero* + *semigroup-add* +
assumes *add-0-left* [*simp*]: $0 + a = a$
and *add-0-right* [*simp*]: $a + 0 = a$

lemma *zero-reorient*: $0 = x \longleftrightarrow x = 0$
by (*rule eq-commute*)

class *comm-monoid-add* = *zero* + *ab-semigroup-add* +
assumes *add-0*: $0 + a = a$
begin

subclass *monoid-add*

proof qed (*insert add-0*, *simp-all add*: *add-commute*)

end

class *monoid-mult* = *one* + *semigroup-mult* +
assumes *mult-1-left* [*simp*]: *1* ∗ *a* = *a*
assumes *mult-1-right* [*simp*]: *a* ∗ *1* = *a*

lemma *one-reorient*: *1* = *x* ⟷ *x* = *1*
by (*rule eq-commute*)

class *comm-monoid-mult* = *one* + *ab-semigroup-mult* +
assumes *mult-1*: *1* ∗ *a* = *a*
begin

subclass *monoid-mult*
proof qed (*insert mult-1*, *simp-all add*: *mult-commute*)

end

class *cancel-semigroup-add* = *semigroup-add* +
assumes *add-left-imp-eq*: *a* + *b* = *a* + *c* ⟹ *b* = *c*
assumes *add-right-imp-eq*: *b* + *a* = *c* + *a* ⟹ *b* = *c*
begin

lemma *add-left-cancel* [*simp*]:
a + *b* = *a* + *c* ⟷ *b* = *c*
by (*blast dest*: *add-left-imp-eq*)

lemma *add-right-cancel* [*simp*]:
b + *a* = *c* + *a* ⟷ *b* = *c*
by (*blast dest*: *add-right-imp-eq*)

end

class *cancel-ab-semigroup-add* = *ab-semigroup-add* +
assumes *add-imp-eq*: *a* + *b* = *a* + *c* ⟹ *b* = *c*
begin

subclass *cancel-semigroup-add*
proof
fix *a b c* :: *'a*
assume *a* + *b* = *a* + *c*
then show *b* = *c* **by** (*rule add-imp-eq*)
next
fix *a b c* :: *'a*
assume *b* + *a* = *c* + *a*
then have *a* + *b* = *a* + *c* **by** (*simp only*: *add-commute*)
then show *b* = *c* **by** (*rule add-imp-eq*)

qed

end

class *cancel-comm-monoid-add* = *cancel-ab-semigroup-add* + *comm-monoid-add*

10.2 Groups

class *group-add* = *minus* + *uminus* + *monoid-add* +
  assumes *left-minus* [*simp*]: − *a* + *a* = *0*
  assumes *diff-minus*: *a* − *b* = *a* + (− *b*)
begin

lemma *minus-add-cancel*: − *a* + (*a* + *b*) = *b*
by (*simp add*: *add-assoc*[*symmetric*])

lemma *minus-zero* [*simp*]: − *0* = *0*
proof −
  have − *0* = − *0* + (*0* + *0*) **by** (*simp only*: *add-0-right*)
  also have ... = *0* **by** (*rule minus-add-cancel*)
  finally show *?thesis* **.**
qed

lemma *minus-minus* [*simp*]: − (− *a*) = *a*
proof −
  have − (− *a*) = − (− *a*) + (− *a* + *a*) **by** *simp*
  also have ... = *a* **by** (*rule minus-add-cancel*)
  finally show *?thesis* **.**
qed

lemma *right-minus* [*simp*]: *a* + − *a* = *0*
proof −
  have *a* + − *a* = − (− *a*) + − *a* **by** *simp*
  also have ... = *0* **by** (*rule left-minus*)
  finally show *?thesis* **.**
qed

lemma *right-minus-eq*: *a* − *b* = *0* ⟷ *a* = *b*
proof
  assume *a* − *b* = *0*
  have *a* = (*a* − *b*) + *b* **by** (*simp add*:*diff-minus add-assoc*)
  also have ... = *b* **using** ‹*a* − *b* = *0*› **by** *simp*
  finally show *a* = *b* **.**
next
  assume *a* = *b* **thus** *a* − *b* = *0* **by** (*simp add*: *diff-minus*)
qed

lemma *equals-zero-I*:
  assumes *a* + *b* = *0* **shows** − *a* = *b*

proof −
have − *a* = − *a* + (*a* + *b*) **using** *assms* **by** *simp*
also have ... = *b* **by** (*simp add*: *add-assoc*[*symmetric*])
finally show *?thesis* **.**
qed

lemma *diff-self* [*simp*]: *a* − *a* = *0*
by (*simp add*: *diff-minus*)

lemma *diff-0* [*simp*]: *0* − *a* = − *a*
by (*simp add*: *diff-minus*)

lemma *diff-0-right* [*simp*]: *a* − *0* = *a*
by (*simp add*: *diff-minus*)

lemma *diff-minus-eq-add* [*simp*]: *a* − − *b* = *a* + *b*
by (*simp add*: *diff-minus*)

lemma *neg-equal-iff-equal* [*simp*]:
− *a* = − *b* ⟷ *a* = *b*
proof
assume − *a* = − *b*
hence − (− *a*) = − (− *b*) **by** *simp*
thus *a* = *b* **by** *simp*
next
assume *a* = *b*
thus − *a* = − *b* **by** *simp*
qed

lemma *neg-equal-0-iff-equal* [*simp*]:
− *a* = *0* ⟷ *a* = *0*
by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

lemma *neg-0-equal-iff-equal* [*simp*]:
0 = − *a* ⟷ *0* = *a*
by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*:
a = − *b* ⟷ *b* = − *a*
proof −
have − (− *a*) = − *b* ⟷ − *a* = *b* **by** (*rule neg-equal-iff-equal*)
thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

lemma *minus-equation-iff*:
− *a* = *b* ⟷ − *b* = *a*
proof −
have − *a* = − (− *b*) ⟷ *a* = −*b* **by** (*rule neg-equal-iff-equal*)

thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

lemma *diff-add-cancel*: $a - b + b = a$
by (*simp add*: *diff-minus add-assoc*)

lemma *add-diff-cancel*: $a + b - b = a$
by (*simp add*: *diff-minus add-assoc*)

declare *diff-minus*[*symmetric*, *algebra-simps*]

lemma *eq-neg-iff-add-eq-0*: $a = - b \longleftrightarrow a + b = 0$
proof
assume $a = - b$ **then show** $a + b = 0$ **by** *simp*
next
assume $a + b = 0$
moreover have $a + (b + - b) = (a + b) + - b$
by (*simp only*: *add-assoc*)
ultimately show $a = - b$ **by** *simp*
qed

end

class *ab-group-add* = *minus* + *uminus* + *comm-monoid-add* +
assumes *ab-left-minus*: $- a + a = 0$
assumes *ab-diff-minus*: $a - b = a + (- b)$
begin

subclass *group-add*
proof qed (*simp-all add*: *ab-left-minus ab-diff-minus*)

subclass *cancel-comm-monoid-add*
proof
fix $a\ b\ c :: 'a$
assume $a + b = a + c$
then have $- a + a + b = - a + a + c$
unfolding *add-assoc* **by** *simp*
then show $b = c$ **by** *simp*
qed

lemma *uminus-add-conv-diff*[*algebra-simps*]:
$- a + b = b - a$
by (*simp add*:*diff-minus add-commute*)

lemma *minus-add-distrib* [*simp*]:
$- (a + b) = - a + - b$
by (*rule equals-zero-I*) (*simp add*: *add-ac*)

lemma *minus-diff-eq* [*simp*]:

$-(a - b) = b - a$
by (*simp add*: *diff-minus add-commute*)

lemma *add-diff-eq*[*algebra-simps*]: $a + (b - c) = (a + b) - c$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-add-eq*[*algebra-simps*]: $(a - b) + c = (a + c) - b$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-eq-eq*[*algebra-simps*]: $a - b = c \longleftrightarrow a = c + b$
by (*auto simp add*: *diff-minus add-assoc*)

lemma *eq-diff-eq*[*algebra-simps*]: $a = c - b \longleftrightarrow a + b = c$
by (*auto simp add*: *diff-minus add-assoc*)

lemma *diff-diff-eq*[*algebra-simps*]: $(a - b) - c = a - (b + c)$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-diff-eq2*[*algebra-simps*]: $a - (b - c) = (a + c) - b$
by (*simp add*: *diff-minus add-ac*)

lemma *eq-iff-diff-eq-0*: $a = b \longleftrightarrow a - b = 0$
by (*simp add*: *algebra-simps*)

lemma *diff-eq-0-iff-eq* [*simp*, *noatp*]: $a - b = 0 \longleftrightarrow a = b$
by (*simp add*: *algebra-simps*)

end

10.3 (Partially) Ordered Groups

class *pordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +
  assumes *add-left-mono*: $a \leq b \Longrightarrow c + a \leq c + b$
begin

lemma *add-right-mono*:
  $a \leq b \Longrightarrow a + c \leq b + c$
by (*simp add*: *add-commute* [*of* - *c*] *add-left-mono*)

non-strict, in both arguments

lemma *add-mono*:
  $a \leq b \Longrightarrow c \leq d \Longrightarrow a + c \leq b + d$
  apply (*erule add-right-mono* [*THEN order-trans*])
  apply (*simp add*: *add-commute add-left-mono*)
  done

end

class *pordered-cancel-ab-semigroup-add* =

pordered-ab-semigroup-add + *cancel-ab-semigroup-add*
begin

lemma *add-strict-left-mono*:
$a < b \Longrightarrow c + a < c + b$
by (*auto simp add*: *less-le add-left-mono*)

lemma *add-strict-right-mono*:
$a < b \Longrightarrow a + c < b + c$
by (*simp add*: *add-commute* [*of* - *c*] *add-strict-left-mono*)

Strict monotonicity in both arguments

lemma *add-strict-mono*:
$a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$
apply (*erule add-strict-right-mono* [*THEN less-trans*])
apply (*erule add-strict-left-mono*)
done

lemma *add-less-le-mono*:
$a < b \Longrightarrow c \leq d \Longrightarrow a + c < b + d$
apply (*erule add-strict-right-mono* [*THEN less-le-trans*])
apply (*erule add-left-mono*)
done

lemma *add-le-less-mono*:
$a \leq b \Longrightarrow c < d \Longrightarrow a + c < b + d$
apply (*erule add-right-mono* [*THEN le-less-trans*])
apply (*erule add-strict-left-mono*)
done

end

class *pordered-ab-semigroup-add-imp-le* =
pordered-cancel-ab-semigroup-add +
assumes *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
begin

lemma *add-less-imp-less-left*:
assumes *less*: $c + a < c + b$ **shows** $a < b$
proof −
from *less* **have** *le*: $c + a <= c + b$ **by** (*simp add*: *order-le-less*)
have $a <= b$
apply (*insert le*)
apply (*drule add-le-imp-le-left*)
by (*insert le*, *drule add-le-imp-le-left*, *assumption*)
moreover have $a \neq b$
proof (*rule ccontr*)
assume $\sim(a \neq b)$
then have $a = b$ **by** *simp*

```
    then have c + a = c + b by simp
    with less show Falseby simp
  qed
  ultimately show a < b by (simp add: order-le-less)
qed

lemma add-less-imp-less-right:
  a + c < b + c ⟹ a < b
apply (rule add-less-imp-less-left [of c])
apply (simp add: add-commute)
done

lemma add-less-cancel-left [simp]:
  c + a < c + b ⟷ a < b
by (blast intro: add-less-imp-less-left add-strict-left-mono)

lemma add-less-cancel-right [simp]:
  a + c < b + c ⟷ a < b
by (blast intro: add-less-imp-less-right add-strict-right-mono)

lemma add-le-cancel-left [simp]:
  c + a ≤ c + b ⟷ a ≤ b
by (auto, drule add-le-imp-le-left, simp-all add: add-left-mono)

lemma add-le-cancel-right [simp]:
  a + c ≤ b + c ⟷ a ≤ b
by (simp add: add-commute [of a c] add-commute [of b c])

lemma add-le-imp-le-right:
  a + c ≤ b + c ⟹ a ≤ b
by simp

lemma max-add-distrib-left:
  max x y + z = max (x + z) (y + z)
  unfolding max-def by auto

lemma min-add-distrib-left:
  min x y + z = min (x + z) (y + z)
  unfolding min-def by auto

end
```

10.4 Support for reasoning about signs

```
class pordered-comm-monoid-add =
  pordered-cancel-ab-semigroup-add + comm-monoid-add
begin

lemma add-pos-nonneg:
```

assumes $0 < a$ **and** $0 \leq b$ **shows** $0 < a + b$
proof −
have $0 + 0 < a + b$
using *assms* **by** (*rule add-less-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-pos-pos*:
assumes $0 < a$ **and** $0 < b$ **shows** $0 < a + b$
by (*rule add-pos-nonneg*) (*insert assms*, *auto*)

lemma *add-nonneg-pos*:
assumes $0 \leq a$ **and** $0 < b$ **shows** $0 < a + b$
proof −
have $0 + 0 < a + b$
using *assms* **by** (*rule add-le-less-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonneg-nonneg*:
assumes $0 \leq a$ **and** $0 \leq b$ **shows** $0 \leq a + b$
proof −
have $0 + 0 \leq a + b$
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-neg-nonpos*:
assumes $a < 0$ **and** $b \leq 0$ **shows** $a + b < 0$
proof −
have $a + b < 0 + 0$
using *assms* **by** (*rule add-less-le-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-neg-neg*:
assumes $a < 0$ **and** $b < 0$ **shows** $a + b < 0$
by (*rule add-neg-nonpos*) (*insert assms*, *auto*)

lemma *add-nonpos-neg*:
assumes $a \leq 0$ **and** $b < 0$ **shows** $a + b < 0$
proof −
have $a + b < 0 + 0$
using *assms* **by** (*rule add-le-less-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonpos-nonpos*:
assumes $a \leq 0$ **and** $b \leq 0$ **shows** $a + b \leq 0$

proof −
have $a + b \leq 0 + 0$
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemmas *add-sign-intros* =
add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

lemma *add-nonneg-eq-0-iff*:
assumes x: $0 \leq x$ **and** y: $0 \leq y$
shows $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
proof (*intro iffI conjI*)
have $x = x + 0$ **by** *simp*
also have $x + 0 \leq x + y$ **using** y **by** (*rule add-left-mono*)
also assume $x + y = 0$
also have $0 \leq x$ **using** x **.**
finally show $x = 0$ **.**
next
have $y = 0 + y$ **by** *simp*
also have $0 + y \leq x + y$ **using** x **by** (*rule add-right-mono*)
also assume $x + y = 0$
also have $0 \leq y$ **using** y **.**
finally show $y = 0$ **.**
next
assume $x = 0 \wedge y = 0$
then show $x + y = 0$ **by** *simp*
qed

end

class *pordered-ab-group-add* =
ab-group-add + *pordered-ab-semigroup-add*
begin

subclass *pordered-cancel-ab-semigroup-add* **..**

subclass *pordered-ab-semigroup-add-imp-le*
proof
fix a b c :: $'a$
assume $c + a \leq c + b$
hence $(-c) + (c + a) \leq (-c) + (c + b)$ **by** (*rule add-left-mono*)
hence $((-c) + c) + a \leq ((-c) + c) + b$ **by** (*simp only*: *add-assoc*)
thus $a \leq b$ **by** *simp*
qed

subclass *pordered-comm-monoid-add* **..**

lemma *max-diff-distrib-left*:
shows $max\ x\ y - z = max\ (x - z)\ (y - z)$
by (*simp add*: *diff-minus*, *rule max-add-distrib-left*)

lemma *min-diff-distrib-left*:
shows $min\ x\ y - z = min\ (x - z)\ (y - z)$
by (*simp add*: *diff-minus*, *rule min-add-distrib-left*)

lemma *le-imp-neg-le*:
assumes $a \leq b$ **shows** $-b \leq -a$
proof −
have $-a+a \leq -a+b$ **using** ‹$a \leq b$› **by** (*rule add-left-mono*)
hence $0 \leq -a+b$ **by** *simp*
hence $0 + (-b) \leq (-a + b) + (-b)$ **by** (*rule add-right-mono*)
thus *?thesis* **by** (*simp add*: *add-assoc*)
qed

lemma *neg-le-iff-le* [*simp*]: $- b \leq - a \longleftrightarrow a \leq b$
proof
assume $- b \leq - a$
hence $- (- a) \leq - (- b)$ **by** (*rule le-imp-neg-le*)
thus $a \leq b$ **by** *simp*
next
assume $a \leq b$
thus $-b \leq -a$ **by** (*rule le-imp-neg-le*)
qed

lemma *neg-le-0-iff-le* [*simp*]: $- a \leq 0 \longleftrightarrow 0 \leq a$
by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-0-le-iff-le* [*simp*]: $0 \leq - a \longleftrightarrow a \leq 0$
by (*subst neg-le-iff-le* [*symmetric*], *simp*)

lemma *neg-less-iff-less* [*simp*]: $- b < - a \longleftrightarrow a < b$
by (*force simp add*: *less-le*)

lemma *neg-less-0-iff-less* [*simp*]: $- a < 0 \longleftrightarrow 0 < a$
by (*subst neg-less-iff-less* [*symmetric*], *simp*)

lemma *neg-0-less-iff-less* [*simp*]: $0 < - a \longleftrightarrow a < 0$
by (*subst neg-less-iff-less* [*symmetric*], *simp*)

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < - b \longleftrightarrow b < - a$
proof −
have $(- (-a) < - b) = (b < - a)$ **by** (*rule neg-less-iff-less*)
thus *?thesis* **by** *simp*
qed

lemma *minus-less-iff*: $- a < b \longleftrightarrow - b < a$
proof −
  have $(- a < - (-b)) = (- b < a)$ **by** (*rule neg-less-iff-less*)
  thus *?thesis* **by** *simp*
qed

lemma *le-minus-iff*: $a \leq - b \longleftrightarrow b \leq - a$
proof −
 have *mm*: !! a (b::'a). $(-(-a)) < -b \Longrightarrow -(-b) < -a$ **by** (*simp only*: *minus-less-iff*)
  have $(- (- a) <= -b) = (b <= - a)$
    apply (*auto simp only*: *le-less*)
    apply (*drule mm*)
    apply (*simp-all*)
    apply (*drule mm*[*simplified*], *assumption*)
    done
  then show *?thesis* **by** *simp*
qed

lemma *minus-le-iff*: $- a \leq b \longleftrightarrow - b \leq a$
by (*auto simp add*: *le-less minus-less-iff*)

lemma *less-iff-diff-less-0*: $a < b \longleftrightarrow a - b < 0$
proof −
  have $(a < b) = (a + (- b) < b + (-b))$
    by (*simp only*: *add-less-cancel-right*)
  also have ... = $(a - b < 0)$ **by** (*simp add*: *diff-minus*)
  finally show *?thesis* **.**
qed

lemma *diff-less-eq*[*algebra-simps*]: $a - b < c \longleftrightarrow a < c + b$
apply (*subst less-iff-diff-less-0* [*of a*])
apply (*rule less-iff-diff-less-0* [*of* - *c*, *THEN ssubst*])
apply (*simp add*: *diff-minus add-ac*)
done

lemma *less-diff-eq*[*algebra-simps*]: $a < c - b \longleftrightarrow a + b < c$
apply (*subst less-iff-diff-less-0* [*of plus a b*])
apply (*subst less-iff-diff-less-0* [*of a*])
apply (*simp add*: *diff-minus add-ac*)
done

lemma *diff-le-eq*[*algebra-simps*]: $a - b \leq c \longleftrightarrow a \leq c + b$
by (*auto simp add*: *le-less diff-less-eq diff-add-cancel add-diff-cancel*)

lemma *le-diff-eq*[*algebra-simps*]: $a \leq c - b \longleftrightarrow a + b \leq c$
by (*auto simp add*: *le-less less-diff-eq diff-add-cancel add-diff-cancel*)

lemma *le-iff-diff-le-0*: $a \leq b \longleftrightarrow a - b \leq 0$
by (*simp add*: *algebra-simps*)

Legacy - use *algebra-simps*

lemmas *group-simps*[*noatp*] = *algebra-simps*

end

Legacy - use *algebra-simps*

lemmas *group-simps*[*noatp*] = *algebra-simps*

class *ordered-ab-semigroup-add* =
  linorder + *pordered-ab-semigroup-add*

class *ordered-cancel-ab-semigroup-add* =
  linorder + *pordered-cancel-ab-semigroup-add*
begin

subclass *ordered-ab-semigroup-add* **..**

subclass *pordered-ab-semigroup-add-imp-le*
proof
  fix *a b c* :: *'a*
  assume *le*: *c* + *a* <= *c* + *b*
  show *a* <= *b*
  proof (*rule ccontr*)
    assume *w*: ~ *a* ≤ *b*
    hence *b* <= *a* **by** (*simp add*: *linorder-not-le*)
    hence *le2*: *c* + *b* <= *c* + *a* **by** (*rule add-left-mono*)
    have *a* = *b*
      apply (*insert le*)
      apply (*insert le2*)
      apply (*drule antisym*, *simp-all*)
      done
    with *w* **show** *False*
      by (*simp add*: *linorder-not-le* [*symmetric*])
  qed
qed

end

class *ordered-ab-group-add* =
  linorder + *pordered-ab-group-add*
begin

subclass *ordered-cancel-ab-semigroup-add* **..**

lemma *neg-less-eq-nonneg*:
  − *a* ≤ *a* ⟷ *0* ≤ *a*
proof
  assume *A*: − *a* ≤ *a* **show** *0* ≤ *a*
  proof (*rule classical*)

```
    assume ¬ 0 ≤ a
    then have a < 0 by auto
    with A have − a < 0 by (rule le-less-trans)
    then show ?thesis by auto
  qed
next
  assume A: 0 ≤ a show − a ≤ a
  proof (rule order-trans)
    show − a ≤ 0 using A by (simp add: minus-le-iff)
  next
    show 0 ≤ a using A .
  qed
qed

lemma less-eq-neg-nonpos:
  a ≤ − a ⟷ a ≤ 0
proof
  assume A: a ≤ − a show a ≤ 0
  proof (rule classical)
    assume ¬ a ≤ 0
    then have 0 < a by auto
    then have 0 < − a using A by (rule less-le-trans)
    then show ?thesis by auto
  qed
next
  assume A: a ≤ 0 show a ≤ − a
  proof (rule order-trans)
    show 0 ≤ − a using A by (simp add: minus-le-iff)
  next
    show a ≤ 0 using A .
  qed
qed

lemma equal-neg-zero:
  a = − a ⟷ a = 0
proof
  assume a = 0 then show a = − a by simp
next
  assume A: a = − a show a = 0
  proof (cases 0 ≤ a)
    case True with A have 0 ≤ − a by auto
    with le-minus-iff have a ≤ 0 by simp
    with True show ?thesis by (auto intro: order-trans)
  next
    case False then have B: a ≤ 0 by auto
    with A have − a ≤ 0 by auto
    with B show ?thesis by (auto intro: order-trans)
  qed
qed
```

lemma *neg-equal-zero*:
  $-\ a = a \longleftrightarrow a = 0$
  unfolding *equal-neg-zero* [*symmetric*] **by** *auto*

end

— FIXME localize the following

lemma *add-increasing*:
  fixes $c :: 'a::\{$*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*$\}$
  shows $[|0 \leq a;\ b \leq c|] ==> b \leq a + c$
by (*insert add-mono* [*of 0 a b c*], *simp*)

lemma *add-increasing2*:
  fixes $c :: 'a::\{$*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*$\}$
  shows $[|0 \leq c;\ b \leq a|] ==> b \leq a + c$
by (*simp add*:*add-increasing add-commute*[*of a*])

lemma *add-strict-increasing*:
  fixes $c :: 'a::\{$*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*$\}$
  shows $[|0 < a;\ b \leq c|] ==> b < a + c$
by (*insert add-less-le-mono* [*of 0 a b c*], *simp*)

lemma *add-strict-increasing2*:
  fixes $c :: 'a::\{$*pordered-ab-semigroup-add-imp-le*, *comm-monoid-add*$\}$
  shows $[|0 \leq a;\ b < c|] ==> b < a + c$
by (*insert add-le-less-mono* [*of 0 a b c*], *simp*)

class *pordered-ab-group-add-abs* = *pordered-ab-group-add* + *abs* +
  assumes *abs-ge-zero* [*simp*]: $|a| \geq 0$
    and *abs-ge-self*: $a \leq |a|$
    and *abs-leI*: $a \leq b \Longrightarrow -\ a \leq b \Longrightarrow |a| \leq b$
    and *abs-minus-cancel* [*simp*]: $|-a| = |a|$
    and *abs-triangle-ineq*: $|a + b| \leq |a| + |b|$
begin

lemma *abs-minus-le-zero*: $-\ |a| \leq 0$
  unfolding *neg-le-0-iff-le* **by** *simp*

lemma *abs-of-nonneg* [*simp*]:
  assumes *nonneg*: $0 \leq a$ **shows** $|a| = a$
proof (*rule antisym*)
  from *nonneg le-imp-neg-le* **have** $-\ a \leq 0$ **by** *simp*
  from *this nonneg* **have** $-\ a \leq a$ **by** (*rule order-trans*)
  then show $|a| \leq a$ **by** (*auto intro*: *abs-leI*)
qed (*rule abs-ge-self*)

lemma *abs-idempotent* [*simp*]: $||a|| = |a|$
by (*rule antisym*)
  (*auto intro*!: *abs-ge-self abs-leI order-trans* [*of uminus* (*abs a*) *zero abs a*])

lemma *abs-eq-0* [*simp*]: $|a| = 0 \longleftrightarrow a = 0$
proof −
  have $|a| = 0 \Longrightarrow a = 0$
  proof (*rule antisym*)
    assume *zero*: $|a| = 0$
    with *abs-ge-self* **show** $a \leq 0$ **by** *auto*
    from *zero* **have** $|-a| = 0$ **by** *simp*
    with *abs-ge-self* [*of uminus a*] **have** $- a \leq 0$ **by** *auto*
    with *neg-le-0-iff-le* **show** $0 \leq a$ **by** *auto*
  qed
  then show *?thesis* **by** *auto*
qed

lemma *abs-zero* [*simp*]: $|0| = 0$
by *simp*

lemma *abs-0-eq* [*simp*, *noatp*]: $0 = |a| \longleftrightarrow a = 0$
proof −
  have $0 = |a| \longleftrightarrow |a| = 0$ **by** (*simp only*: *eq-ac*)
  thus *?thesis* **by** *simp*
qed

lemma *abs-le-zero-iff* [*simp*]: $|a| \leq 0 \longleftrightarrow a = 0$
proof
  assume $|a| \leq 0$
  then have $|a| = 0$ **by** (*rule antisym*) *simp*
  thus $a = 0$ **by** *simp*
next
  assume $a = 0$
  thus $|a| \leq 0$ **by** *simp*
qed

lemma *zero-less-abs-iff* [*simp*]: $0 < |a| \longleftrightarrow a \neq 0$
by (*simp add*: *less-le*)

lemma *abs-not-less-zero* [*simp*]: $\neg\ |a| < 0$
proof −
  have *a*: $\bigwedge x\ y.\ x \leq y \Longrightarrow \neg\ y < x$ **by** *auto*
  show *?thesis* **by** (*simp add*: *a*)
qed

lemma *abs-ge-minus-self*: $- a \leq |a|$
proof −
  have $- a \leq |-a|$ **by** (*rule abs-ge-self*)
  then show *?thesis* **by** *simp*

qed

lemma *abs-minus-commute*:
$|a - b| = |b - a|$
proof −
have $|a - b| = |-(a - b)|$ **by** (*simp only*: *abs-minus-cancel*)
also have ... $= |b - a|$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *abs-of-pos*: $0 < a \Longrightarrow |a| = a$
by (*rule abs-of-nonneg*, *rule less-imp-le*)

lemma *abs-of-nonpos* [*simp*]:
assumes $a \leq 0$ **shows** $|a| = - a$
proof −
let *?b* $= - a$
have $- ?b \leq 0 \Longrightarrow |- ?b| = - (- ?b)$
unfolding *abs-minus-cancel* [*of ?b*]
unfolding *neg-le-0-iff-le* [*of ?b*]
unfolding *minus-minus* **by** (*erule abs-of-nonneg*)
then show *?thesis* **using** *assms* **by** *auto*
qed

lemma *abs-of-neg*: $a < 0 \Longrightarrow |a| = - a$
by (*rule abs-of-nonpos*, *rule less-imp-le*)

lemma *abs-le-D1*: $|a| \leq b \Longrightarrow a \leq b$
by (*insert abs-ge-self*, *blast intro*: *order-trans*)

lemma *abs-le-D2*: $|a| \leq b \Longrightarrow - a \leq b$
by (*insert abs-le-D1* [*of uminus a*], *simp*)

lemma *abs-le-iff*: $|a| \leq b \longleftrightarrow a \leq b \wedge - a \leq b$
by (*blast intro*: *abs-leI dest*: *abs-le-D1 abs-le-D2*)

lemma *abs-triangle-ineq2*: $|a| - |b| \leq |a - b|$
apply (*simp add*: *algebra-simps*)
apply (*subgoal-tac abs a = abs* (*plus b* (*minus a b*)))
apply (*erule ssubst*)
apply (*rule abs-triangle-ineq*)
apply (*rule arg-cong*[*of - - abs*])
apply (*simp add*: *algebra-simps*)
done

lemma *abs-triangle-ineq3*: $||a| - |b|| \leq |a - b|$
apply (*subst abs-le-iff*)
apply *auto*
apply (*rule abs-triangle-ineq2*)

```
  apply (subst abs-minus-commute)
  apply (rule abs-triangle-ineq2)
done

lemma abs-triangle-ineq4: |a − b| ≤ |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)| ≤ |a − c| + |b − d|
proof −
  have |a + b − (c+d)| = |(a−c) + (b−d)| by (simp add: diff-minus add-ac)
  also have ... ≤ |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 ≥ ?R by(rule abs-ge-self)
next
  have ?L ≤ ||a|| + ||b|| by(rule abs-triangle-ineq)
  also have ... = ?R by simp
  finally show ?L ≤ ?R .
qed

end
```

10.5 Lattice Ordered (Abelian) Groups

```
class lordered-ab-group-add-meet = pordered-ab-group-add + lower-semilattice
begin

lemma add-inf-distrib-left:
  a + inf b c = inf (a + b) (a + c)
apply (rule antisym)
apply (simp-all add: le-infI)
apply (rule add-le-imp-le-left [of uminus a])
apply (simp only: add-assoc [symmetric], simp)
apply rule
apply (rule add-le-imp-le-left[of a], simp only: add-assoc[symmetric], simp)+
done

lemma add-inf-distrib-right:
  inf a b + c = inf (a + c) (b + c)
proof −
  have c + inf a b = inf (c+a) (c+b) by (simp add: add-inf-distrib-left)
```

thus *?thesis* **by** (*simp add*: *add-commute*)
qed

end

class *lordered-ab-group-add-join* = *pordered-ab-group-add* + *upper-semilattice*
begin

lemma *add-sup-distrib-left*:
a + *sup b c* = *sup* (*a* + *b*) (*a* + *c*)
apply (*rule antisym*)
apply (*rule add-le-imp-le-left* [*of uminus a*])
apply (*simp only*: *add-assoc*[*symmetric*], *simp*)
apply *rule*
apply (*rule add-le-imp-le-left* [*of a*], *simp only*: *add-assoc*[*symmetric*], *simp*)+
apply (*rule le-supI*)
apply (*simp-all*)
done

lemma *add-sup-distrib-right*:
sup a b + *c* = *sup* (*a*+*c*) (*b*+*c*)
proof −
have *c* + *sup a b* = *sup* (*c*+*a*) (*c*+*b*) **by** (*simp add*: *add-sup-distrib-left*)
thus *?thesis* **by** (*simp add*: *add-commute*)
qed

end

class *lordered-ab-group-add* = *pordered-ab-group-add* + *lattice*
begin

subclass *lordered-ab-group-add-meet* ..
subclass *lordered-ab-group-add-join* ..

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

lemma *inf-eq-neg-sup*: *inf a b* = − *sup* (−*a*) (−*b*)
proof (*rule inf-unique*)
fix *a b* :: *'a*
show − *sup* (−*a*) (−*b*) ≤ *a*
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
next
fix *a b* :: *'a*
show − *sup* (−*a*) (−*b*) ≤ *b*
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
next

fix $a\ b\ c :: 'a$
assume $a \leq b\ a \leq c$
then show $a \leq -\ sup\ (-b)\ (-c)$ **by** (*subst neg-le-iff-le* [*symmetric*])
(*simp add*: *le-supI*)
qed

lemma *sup-eq-neg-inf*: $sup\ a\ b = -\ inf\ (-a)\ (-b)$
proof (*rule sup-unique*)
fix $a\ b :: 'a$
show $a \leq -\ inf\ (-a)\ (-b)$
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
next
fix $a\ b :: 'a$
show $b \leq -\ inf\ (-a)\ (-b)$
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
next
fix $a\ b\ c :: 'a$
assume $a \leq c\ b \leq c$
then show $-\ inf\ (-a)\ (-b) \leq c$ **by** (*subst neg-le-iff-le* [*symmetric*])
(*simp add*: *le-infI*)
qed

lemma *neg-inf-eq-sup*: $-\ inf\ a\ b = sup\ (-a)\ (-b)$
by (*simp add*: *inf-eq-neg-sup*)

lemma *neg-sup-eq-inf*: $-\ sup\ a\ b = inf\ (-a)\ (-b)$
by (*simp add*: *sup-eq-neg-inf*)

lemma *add-eq-inf-sup*: $a + b = sup\ a\ b + inf\ a\ b$
proof −
have $0 = -\ inf\ 0\ (a-b) + inf\ (a-b)\ 0$ **by** (*simp add*: *inf-commute*)
hence $0 = sup\ 0\ (b-a) + inf\ (a-b)\ 0$ **by** (*simp add*: *inf-eq-neg-sup*)
hence $0 = (-a + sup\ a\ b) + (inf\ a\ b + (-b))$
by (*simp add*: *add-sup-distrib-left add-inf-distrib-right*)
(*simp add*: *algebra-simps*)
thus *?thesis* **by** (*simp add*: *algebra-simps*)
qed

10.6 Positive Part, Negative Part, Absolute Value

definition
$nprt :: 'a \Rightarrow 'a$ **where**
$nprt\ x = inf\ x\ 0$

definition
$pprt :: 'a \Rightarrow 'a$ **where**
$pprt\ x = sup\ x\ 0$

lemma *pprt-neg*: *pprt* (− *x*) = − *nprt x*
proof −
have *sup* (− *x*) *0* = *sup* (− *x*) (− *0*) **unfolding** *minus-zero* **..**
also have ... = − *inf x 0* **unfolding** *neg-inf-eq-sup* **..**
finally have *sup* (− *x*) *0* = − *inf x 0* **.**
then show *?thesis* **unfolding** *pprt-def nprt-def* **.**
qed

lemma *nprt-neg*: *nprt* (− *x*) = − *pprt x*
proof −
from *pprt-neg* **have** *pprt* (− (− *x*)) = − *nprt* (− *x*) **.**
then have *pprt x* = − *nprt* (− *x*) **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *prts*: *a* = *pprt a* + *nprt a*
by (*simp add*: *pprt-def nprt-def add-eq-inf-sup*[*symmetric*])

lemma *zero-le-pprt*[*simp*]: *0* ≤ *pprt a*
by (*simp add*: *pprt-def*)

lemma *nprt-le-zero*[*simp*]: *nprt a* ≤ *0*
by (*simp add*: *nprt-def*)

lemma *le-eq-neg*: *a* ≤ − *b* ⟷ *a* + *b* ≤ *0* (**is** *?l* = *?r*)
proof −
have *a*: *?l* ⟶ *?r*
apply (*auto*)
apply (*rule add-le-imp-le-right*[*of* - *uminus b* -])
apply (*simp add*: *add-assoc*)
done
have *b*: *?r* ⟶ *?l*
apply (*auto*)
apply (*rule add-le-imp-le-right*[*of* - *b* -])
apply (*simp*)
done
from *a b* **show** *?thesis* **by** *blast*
qed

lemma *pprt-0*[*simp*]: *pprt 0* = *0* **by** (*simp add*: *pprt-def*)
lemma *nprt-0*[*simp*]: *nprt 0* = *0* **by** (*simp add*: *nprt-def*)

lemma *pprt-eq-id* [*simp*, *noatp*]: *0* ≤ *x* ⟹ *pprt x* = *x*
by (*simp add*: *pprt-def le-iff-sup sup-ACI*)

lemma *nprt-eq-id* [*simp*, *noatp*]: *x* ≤ *0* ⟹ *nprt x* = *x*
by (*simp add*: *nprt-def le-iff-inf inf-ACI*)

lemma *pprt-eq-0* [*simp*, *noatp*]: $x \leq 0 \Longrightarrow pprt\ x = 0$
by (*simp add*: *pprt-def le-iff-sup sup-ACI*)

lemma *nprt-eq-0* [*simp*, *noatp*]: $0 \leq x \Longrightarrow nprt\ x = 0$
by (*simp add*: *nprt-def le-iff-inf inf-ACI*)

lemma *sup-0-imp-0*: $sup\ a\ (-\ a) = 0 \Longrightarrow a = 0$
proof −
  {
    fix *a*::'*a*
    assume *hyp*: $sup\ a\ (-a) = 0$
    hence $sup\ a\ (-a) + a = a$ **by** (*simp*)
    hence $sup\ (a+a)\ 0 = a$ **by** (*simp add*: *add-sup-distrib-right*)
    hence $sup\ (a+a)\ 0 <= a$ **by** (*simp*)
    hence $0 <= a$ **by** (*blast intro*: *order-trans inf-sup-ord*)
  }
  note $p = this$
  assume *hyp*:$sup\ a\ (-a) = 0$
  hence *hyp2*:$sup\ (-a)\ (-(-a)) = 0$ **by** (*simp add*: *sup-commute*)
  from *p*[*OF hyp*] *p*[*OF hyp2*] **show** $a = 0$ **by** *simp*
qed

lemma *inf-0-imp-0*: $inf\ a\ (-a) = 0 \Longrightarrow a = 0$
apply (*simp add*: *inf-eq-neg-sup*)
apply (*simp add*: *sup-commute*)
apply (*erule sup-0-imp-0*)
done

lemma *inf-0-eq-0* [*simp*, *noatp*]: $inf\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
by (*rule*, *erule inf-0-imp-0*) *simp*

lemma *sup-0-eq-0* [*simp*, *noatp*]: $sup\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
by (*rule*, *erule sup-0-imp-0*) *simp*

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]:
  $0 \leq a + a \longleftrightarrow 0 \leq a$
proof
  assume $0 <= a + a$
  hence *a*:$inf\ (a+a)\ 0 = 0$ **by** (*simp add*: *le-iff-inf inf-commute*)
  have $(inf\ a\ 0)+(inf\ a\ 0) = inf\ (inf\ (a+a)\ 0)\ a$ (**is** *?l*=-)
    by (*simp add*: *add-sup-inf-distribs inf-ACI*)
  hence $?l = 0 + inf\ a\ 0$ **by** (*simp add*: *a*, *simp add*: *inf-commute*)
  hence $inf\ a\ 0 = 0$ **by** (*simp only*: *add-right-cancel*)
  then show $0 <= a$ **by** (*simp add*: *le-iff-inf inf-commute*)
next
  assume *a*: $0 <= a$
  show $0 <= a + a$ **by** (*simp add*: *add-mono*[*OF a a*, *simplified*])
qed

lemma *double-zero*: $a + a = 0 \longleftrightarrow a = 0$
proof
assume *assm*: $a + a = 0$
then have $a + a + - a = - a$ **by** *simp*
then have $a + (a + - a) = - a$ **by** (*simp only*: *add-assoc*)
then have *a*: $- a = a$ **by** *simp*
show $a = 0$ **apply** (*rule antisym*)
apply (*unfold neg-le-iff-le* [*symmetric*, *of a*])
unfolding *a* **apply** *simp*
unfolding *zero-le-double-add-iff-zero-le-single-add* [*symmetric*, *of a*]
unfolding *assm* **unfolding** *le-less* **apply** *simp-all* **done**
next
assume $a = 0$ **then show** $a + a = 0$ **by** *simp*
qed

lemma *zero-less-double-add-iff-zero-less-single-add*:
$0 < a + a \longleftrightarrow 0 < a$
proof (*cases* $a = 0$)
case *True* **then show** *?thesis* **by** *auto*
next
case *False* **then show** *?thesis*
unfolding *less-le* **apply** *simp* **apply** *rule*
apply *clarify*
apply *rule*
apply *assumption*
apply (*rule notI*)
unfolding *double-zero* [*symmetric*, *of a*] **apply** *simp*
done
qed

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]:
$a + a \leq 0 \longleftrightarrow a \leq 0$
proof −
have $a + a \leq 0 \longleftrightarrow 0 \leq - (a + a)$ **by** (*subst le-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a \leq 0$ **by** (*simp add*: *zero-le-double-add-iff-zero-le-single-add*)
ultimately show *?thesis* **by** *blast*
qed

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]:
$a + a < 0 \longleftrightarrow a < 0$
proof −
have $a + a < 0 \longleftrightarrow 0 < - (a + a)$ **by** (*subst less-minus-iff*, *simp*)
moreover have $\ldots \longleftrightarrow a < 0$ **by** (*simp add*: *zero-less-double-add-iff-zero-less-single-add*)
ultimately show *?thesis* **by** *blast*
qed

declare *neg-inf-eq-sup* [*simp*] *neg-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq - a \longleftrightarrow a \leq 0$

proof −
from *add-le-cancel-left* [*of uminus a plus a a zero*]
have (*a* <= −*a*) = (*a*+*a* <= *0*)
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *minus-le-self-iff*: − *a* ≤ *a* ⟷ *0* ≤ *a*
proof −
from *add-le-cancel-left* [*of uminus a zero plus a a*]
have (−*a* <= *a*) = (*0* <= *a*+*a*)
by (*simp add*: *add-assoc*[*symmetric*])
thus *?thesis* **by** *simp*
qed

lemma *zero-le-iff-zero-nprt*: *0* ≤ *a* ⟷ *nprt a* = *0*
by (*simp add*: *le-iff-inf nprt-def inf-commute*)

lemma *le-zero-iff-zero-pprt*: *a* ≤ *0* ⟷ *pprt a* = *0*
by (*simp add*: *le-iff-sup pprt-def sup-commute*)

lemma *le-zero-iff-pprt-id*: *0* ≤ *a* ⟷ *pprt a* = *a*
by (*simp add*: *le-iff-sup pprt-def sup-commute*)

lemma *zero-le-iff-nprt-id*: *a* ≤ *0* ⟷ *nprt a* = *a*
by (*simp add*: *le-iff-inf nprt-def inf-commute*)

lemma *pprt-mono* [*simp*, *noatp*]: *a* ≤ *b* ⟹ *pprt a* ≤ *pprt b*
by (*simp add*: *le-iff-sup pprt-def sup-ACI sup-assoc* [*symmetric*, *of a*])

lemma *nprt-mono* [*simp*, *noatp*]: *a* ≤ *b* ⟹ *nprt a* ≤ *nprt b*
by (*simp add*: *le-iff-inf nprt-def inf-ACI inf-assoc* [*symmetric*, *of a*])

end

lemmas *add-sup-inf-distribs* = *add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left*

class *lordered-ab-group-add-abs* = *lordered-ab-group-add* + *abs* +
assumes *abs-lattice*: |*a*| = *sup a* (− *a*)
begin

lemma *abs-prts*: |*a*| = *pprt a* − *nprt a*
proof −
have *0* ≤ |*a*|
proof −
have *a*: *a* ≤ |*a*| **and** *b*: − *a* ≤ |*a*| **by** (*auto simp add*: *abs-lattice*)
show *?thesis* **by** (*rule add-mono* [*OF a b*, *simplified*])

```
  qed
  then have 0 ≤ sup a (− a) unfolding abs-lattice .
  then have sup (sup a (− a)) 0 = sup a (− a) by (rule sup-absorb1)
  then show ?thesis
    by (simp add: add-sup-inf-distribs sup-ACI
      pprt-def nprt-def diff-minus abs-lattice)
qed

subclass pordered-ab-group-add-abs
proof
  have abs-ge-zero [simp]: ⋀a. 0 ≤ |a|
  proof −
    fix a b
    have a: a ≤ |a| and b: − a ≤ |a| by (auto simp add: abs-lattice)
    show 0 ≤ |a| by (rule add-mono [OF a b, simplified])
  qed
  have abs-leI: ⋀a b. a ≤ b ⟹ − a ≤ b ⟹ |a| ≤ b
    by (simp add: abs-lattice le-supI)
  fix a b
  show 0 ≤ |a| by simp
  show a ≤ |a|
    by (auto simp add: abs-lattice)
  show |−a| = |a|
    by (simp add: abs-lattice sup-commute)
  show a ≤ b ⟹ − a ≤ b ⟹ |a| ≤ b by (fact abs-leI)
  show |a + b| ≤ |a| + |b|
  proof −
    have g:abs a + abs b = sup (a+b) (sup (−a−b) (sup (−a+b) (a + (−b))))
(is -=sup ?m ?n)
      by (simp add: abs-lattice add-sup-inf-distribs sup-ACI diff-minus)
    have a:a+b <= sup ?m ?n by (simp)
    have b:−a−b <= ?n by (simp)
    have c:?n <= sup ?m ?n by (simp)
    from b c have d: −a−b <= sup ?m ?n by(rule order-trans)
    have e:−a−b = −(a+b) by (simp add: diff-minus)
    from a d e have abs(a+b) <= sup ?m ?n
      by (drule-tac abs-leI, auto)
    with g[symmetric] show ?thesis by simp
  qed
qed

end

lemma sup-eq-if:
  fixes a :: 'a::{lordered-ab-group-add, linorder}
  shows sup a (− a) = (if a < 0 then − a else a)
proof −
  note add-le-cancel-right [of a a − a, symmetric, simplified]
  moreover note add-le-cancel-right [of −a a a, symmetric, simplified]
```

then show *?thesis* **by** (*auto simp*: *sup-max max-def*)
qed

lemma *abs-if-lattice*:
fixes *a* :: *'a*::{*lordered-ab-group-add-abs*, *linorder*}
shows |*a*| = (*if a* < *0 then* − *a else a*)
by *auto*

Needed for abelian cancellation simprocs:

lemma *add-cancel-21*: ((*x*::*'a*::*ab-group-add*) + (*y* + *z*) = *y* + *u*) = (*x* + *z* = *u*)
apply (*subst add-left-commute*)
apply (*subst add-left-cancel*)
apply *simp*
done

lemma *add-cancel-end*: (*x* + (*y* + *z*) = *y*) = (*x* = − (*z*::*'a*::*ab-group-add*))
apply (*subst add-cancel-21*[*of - - - 0*, *simplified*])
apply (*simp add*: *add-right-cancel*[*symmetric*, *of x* −*z z*, *simplified*])
done

lemma *less-eqI*: (*x*::*'a*::*pordered-ab-group-add*) − *y* = *x'* − *y'* ⟹ (*x* < *y*) = (*x'* < *y'*)
by (*simp add*: *less-iff-diff-less-0*[*of x y*] *less-iff-diff-less-0*[*of x' y'*])

lemma *le-eqI*: (*x*::*'a*::*pordered-ab-group-add*) − *y* = *x'* − *y'* ⟹ (*y* <= *x*) = (*y'* <= *x'*)
apply (*simp add*: *le-iff-diff-le-0*[*of y x*] *le-iff-diff-le-0*[*of y' x'*])
apply (*simp add*: *neg-le-iff-le*[*symmetric*, *of y*−*x 0*] *neg-le-iff-le*[*symmetric*, *of y'*−*x' 0*])
done

lemma *eq-eqI*: (*x*::*'a*::*ab-group-add*) − *y* = *x'* − *y'* ⟹ (*x* = *y*) = (*x'* = *y'*)
by (*simp only*: *eq-iff-diff-eq-0*[*of x y*] *eq-iff-diff-eq-0*[*of x' y'*])

lemma *diff-def*: (*x*::*'a*::*ab-group-add*) − *y* == *x* + (−*y*)
by (*simp add*: *diff-minus*)

lemma *add-minus-cancel*: (*a*::*'a*::*ab-group-add*) + (−*a* + *b*) = *b*
by (*simp add*: *add-assoc*[*symmetric*])

lemma *le-add-right-mono*:
assumes
a <= *b* + (*c*::*'a*::*pordered-ab-group-add*)
c <= *d*
shows *a* <= *b* + *d*
apply (*rule-tac order-trans*[**where** *y* = *b*+*c*])
apply (*simp-all add*: *prems*)
done

lemma *estimate-by-abs*:
  $a + b <= (c::'a::lordered\text{-}ab\text{-}group\text{-}add\text{-}abs) \Longrightarrow a <= c + abs\ b$
proof −
  assume $a+b <= c$
  hence *2*: $a <= c+(-b)$ **by** (*simp add*: *algebra-simps*)
  have *3*: $(-b) <= abs\ b$ **by** (*rule abs-ge-minus-self*)
  show *?thesis* **by** (*rule le-add-right-mono*[*OF 2 3*])
qed

10.7 Tools setup

lemma *add-mono-thms-ordered-semiring* [*noatp*]:
  fixes $i\ j\ k :: 'a::pordered\text{-}ab\text{-}semigroup\text{-}add$
  shows $i \le j \wedge k \le l \Longrightarrow i + k \le j + l$
    and $i = j \wedge k \le l \Longrightarrow i + k \le j + l$
    and $i \le j \wedge k = l \Longrightarrow i + k \le j + l$
    and $i = j \wedge k = l \Longrightarrow i + k = j + l$
by (*rule add-mono*, *clarify*+)+

lemma *add-mono-thms-ordered-field* [*noatp*]:
  fixes $i\ j\ k :: 'a::pordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add$
  shows $i < j \wedge k = l \Longrightarrow i + k < j + l$
    and $i = j \wedge k < l \Longrightarrow i + k < j + l$
    and $i < j \wedge k \le l \Longrightarrow i + k < j + l$
    and $i \le j \wedge k < l \Longrightarrow i + k < j + l$
    and $i < j \wedge k < l \Longrightarrow i + k < j + l$
by (*auto intro*: *add-strict-right-mono add-strict-left-mono*
  add-less-le-mono add-le-less-mono add-strict-mono)

Simplification of $x - y < (0::'a)$, etc.

lemmas *diff-less-0-iff-less* [*simp*, *noatp*] = *less-iff-diff-less-0* [*symmetric*]
lemmas *diff-le-0-iff-le* [*simp*, *noatp*] = *le-iff-diff-le-0* [*symmetric*]

ML ⟪
structure ab-group-add-cancel = Abel-Cancel
(

(∗ *term order for abelian groups* ∗)

fun agrp-ord (*Const* (*a*, -)) = *find-index* (*fn a' => a = a'*)
      [@{*const-name HOL.zero*}, @{*const-name HOL.plus*},
        @{*const-name HOL.uminus*}, @{*const-name HOL.minus*}]
  | *agrp-ord* - = ~*1*;

fun termless-agrp (*a*, *b*) = (*TermOrd.term-lpo agrp-ord* (*a*, *b*) = *LESS*);

local
  val ac1 = mk-meta-eq @{*thm add-assoc*};
  val ac2 = mk-meta-eq @{*thm add-commute*};

```
  val ac3 = mk-meta-eq @{thm add-left-commute};
  fun solve-add-ac thy - (- $ (Const (@{const-name HOL.plus},-) $ - $ -) $ -) =
        SOME ac1
    | solve-add-ac thy - (- $ x $ (Const (@{const-name HOL.plus},-) $ y $ z)) =
        if termless-agrp (y, x) then SOME ac3 else NONE
    | solve-add-ac thy - (- $ x $ y) =
        if termless-agrp (y, x) then SOME ac2 else NONE
    | solve-add-ac thy - - = NONE
in
  val add-ac-proc = Simplifier.simproc (the-context ())
    add-ac-proc [x + y::'a::ab-semigroup-add] solve-add-ac;
end;

val eq-reflection = @{thm eq-reflection};

val T = @{typ 'a::ab-group-add};

val cancel-ss = HOL-basic-ss settermless termless-agrp
  addsimprocs [add-ac-proc] addsimps
  [@{thm add-0-left}, @{thm add-0-right}, @{thm diff-def},
   @{thm minus-add-distrib}, @{thm minus-minus}, @{thm minus-zero},
   @{thm right-minus}, @{thm left-minus}, @{thm add-minus-cancel},
   @{thm minus-add-cancel}];

val sum-pats = [@{cterm x + y::'a::ab-group-add}, @{cterm x - y::'a::ab-group-add}];

val eqI-rules = [@{thm less-eqI}, @{thm le-eqI}, @{thm eq-eqI}];

val dest-eqI =
  fst o HOLogic.dest-bin op = HOLogic.boolT o HOLogic.dest-Trueprop o concl-of;

);
⟩⟩

ML ⟨⟨
  Addsimprocs [ab-group-add-cancel.sum-conv, ab-group-add-cancel.rel-conv];
⟩⟩

end
```

11 Ring-and-Field: (Ordered) Rings and Fields

theory *Ring-and-Field*
imports *OrderedGroup*
begin

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer.

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +
  assumes *left-distrib*[*algebra-simps*]: $(a + b) * c = a * c + b * c$
  assumes *right-distrib*[*algebra-simps*]: $a * (b + c) = a * b + a * c$
begin

For the *combine-numerals* simproc

lemma *combine-common-factor*:
  $a * e + (b * e + c) = (a + b) * e + c$
by (*simp add*: *left-distrib add-ac*)

end

class *mult-zero* = *times* + *zero* +
  assumes *mult-zero-left* [*simp*]: $0 * a = 0$
  assumes *mult-zero-right* [*simp*]: $a * 0 = 0$

class *semiring-0* = *semiring* + *comm-monoid-add* + *mult-zero*

class *semiring-0-cancel* = *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0*
proof
  fix $a :: 'a$
  have $0 * a + 0 * a = 0 * a + 0$ **by** (*simp add*: *left-distrib* [*symmetric*])
  thus $0 * a = 0$ **by** (*simp only*: *add-left-cancel*)
next
  fix $a :: 'a$
  have $a * 0 + a * 0 = a * 0 + 0$ **by** (*simp add*: *right-distrib* [*symmetric*])
  thus $a * 0 = 0$ **by** (*simp only*: *add-left-cancel*)
qed

end

class *comm-semiring* = *ab-semigroup-add* + *ab-semigroup-mult* +
  assumes *distrib*: $(a + b) * c = a * c + b * c$
begin

subclass *semiring*
proof
  fix *a b c* :: *'a*
  show $(a + b) * c = a * c + b * c$ **by** (*simp add*: *distrib*)
  have $a * (b + c) = (b + c) * a$ **by** (*simp add*: *mult-ac*)
  also have $... = b * a + c * a$ **by** (*simp only*: *distrib*)
  also have $... = a * b + a * c$ **by** (*simp add*: *mult-ac*)
  finally show $a * (b + c) = a * b + a * c$ **by** *blast*
qed

end

class *comm-semiring-0* = *comm-semiring* + *comm-monoid-add* + *mult-zero*
begin

subclass *semiring-0* **..**

end

class *comm-semiring-0-cancel* = *comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* **..**

subclass *comm-semiring-0* **..**

end

class *zero-neq-one* = *zero* + *one* +
  assumes *zero-neq-one* [*simp*]: $0 \neq 1$
begin

lemma *one-neq-zero* [*simp*]: $1 \neq 0$
by (*rule not-sym*) (*rule zero-neq-one*)

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*

Abstract divisibility

class *dvd* = *times*
begin

definition *dvd* :: *'a* ⇒ *'a* ⇒ *bool* (**infixl** *dvd 50*) **where**
  [*code del*]: *b dvd a* ⟷ $(\exists k.\ a = b * k)$

lemma *dvdI* [*intro?*]: $a = b * k \Longrightarrow b$ *dvd a*
  unfolding *dvd-def* **..**

lemma *dvdE* [*elim?*]: *b dvd a* $\Longrightarrow$ ($\bigwedge k.$ *a* = *b* * *k* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
unfolding *dvd-def* **by** *blast*

end

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult* + *dvd*

begin

subclass *semiring-1* **..**

lemma *dvd-refl*[*simp*]: *a dvd a*
proof
show *a* = *a* * *1* **by** *simp*
qed

lemma *dvd-trans*:
assumes *a dvd b* **and** *b dvd c*
shows *a dvd c*
proof −
from *assms* **obtain** *v* **where** *b* = *a* * *v* **by** (*auto elim*!: *dvdE*)
moreover from *assms* **obtain** *w* **where** *c* = *b* * *w* **by** (*auto elim*!: *dvdE*)
ultimately have *c* = *a* * (*v* * *w*) **by** (*simp add*: *mult-assoc*)
then show *?thesis* **..**
qed

lemma *dvd-0-left-iff* [*noatp*, *simp*]: *0 dvd a* $\longleftrightarrow$ *a* = *0*
by (*auto intro*: *dvd-refl elim*!: *dvdE*)

lemma *dvd-0-right* [*iff*]: *a dvd 0*
proof
show *0* = *a* * *0* **by** *simp*
qed

lemma *one-dvd* [*simp*]: *1 dvd a*
by (*auto intro*!: *dvdI*)

lemma *dvd-mult*[*simp*]: *a dvd c* $\Longrightarrow$ *a dvd* (*b* * *c*)
by (*auto intro*!: *mult-left-commute dvdI elim*!: *dvdE*)

lemma *dvd-mult2*[*simp*]: *a dvd b* $\Longrightarrow$ *a dvd* (*b* * *c*)
apply (*subst mult-commute*)
apply (*erule dvd-mult*)
done

lemma *dvd-triv-right* [*simp*]: *a dvd b* * *a*
by (*rule dvd-mult*) (*rule dvd-refl*)

lemma *dvd-triv-left* [*simp*]: *a dvd a * b*
by (*rule dvd-mult2*) (*rule dvd-refl*)

lemma *mult-dvd-mono*:
assumes *a dvd b*
and *c dvd d*
shows *a * c dvd b * d*
proof −
from ‹*a dvd b*› **obtain** *b'* **where** *b = a * b'* **..**
moreover from ‹*c dvd d*› **obtain** *d'* **where** *d = c * d'* **..**
ultimately have *b * d = (a * c) * (b' * d')* **by** (*simp add*: *mult-ac*)
then show *?thesis* **..**
qed

lemma *dvd-mult-left*: *a * b dvd c ⟹ a dvd c*
by (*simp add*: *dvd-def mult-assoc*, *blast*)

lemma *dvd-mult-right*: *a * b dvd c ⟹ b dvd c*
unfolding *mult-ac* [*of a*] **by** (*rule dvd-mult-left*)

lemma *dvd-0-left*: *0 dvd a ⟹ a = 0*
by *simp*

lemma *dvd-add*[*simp*]:
assumes *a dvd b* **and** *a dvd c* **shows** *a dvd (b + c)*
proof −
from ‹*a dvd b*› **obtain** *b'* **where** *b = a * b'* **..**
moreover from ‹*a dvd c*› **obtain** *c'* **where** *c = a * c'* **..**
ultimately have *b + c = a * (b' + c')* **by** (*simp add*: *right-distrib*)
then show *?thesis* **..**
qed

end

class *no-zero-divisors = zero + times +*
assumes *no-zero-divisors*: *a ≠ 0 ⟹ b ≠ 0 ⟹ a * b ≠ 0*

class *semiring-1-cancel = semiring + cancel-comm-monoid-add*
+ zero-neq-one + monoid-mult
begin

subclass *semiring-0-cancel* **..**

subclass *semiring-1* **..**

end

class *comm-semiring-1-cancel = comm-semiring + cancel-comm-monoid-add*

+ zero-neq-one + comm-monoid-mult
begin

subclass *semiring-1-cancel* **..**
subclass *comm-semiring-0-cancel* **..**
subclass *comm-semiring-1* **..**

end

class *ring* = *semiring + ab-group-add*
begin

subclass *semiring-0-cancel* **..**

Distribution rules

lemma *minus-mult-left*: $- (a * b) = - a * b$
by (*rule equals-zero-I*) (*simp add*: *left-distrib* [*symmetric*])

lemma *minus-mult-right*: $- (a * b) = a * - b$
by (*rule equals-zero-I*) (*simp add*: *right-distrib* [*symmetric*])

Extract signs from products

lemmas *mult-minus-left* [*simp*, *noatp*] = *minus-mult-left* [*symmetric*]
lemmas *mult-minus-right* [*simp*,*noatp*] = *minus-mult-right* [*symmetric*]

lemma *minus-mult-minus* [*simp*]: $- a * - b = a * b$
by *simp*

lemma *minus-mult-commute*: $- a * b = a * - b$
by *simp*

lemma *right-diff-distrib*[*algebra-simps*]: $a * (b - c) = a * b - a * c$
by (*simp add*: *right-distrib diff-minus*)

lemma *left-diff-distrib*[*algebra-simps*]: $(a - b) * c = a * c - b * c$
by (*simp add*: *left-distrib diff-minus*)

lemmas *ring-distribs*[*noatp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

Legacy - use *algebra-simps*

lemmas *ring-simps*[*noatp*] = *algebra-simps*

lemma *eq-add-iff1*:
$a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
by (*simp add*: *algebra-simps*)

lemma *eq-add-iff2*:
$a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$

by (*simp add*: *algebra-simps*)

end

lemmas *ring-distribs*[*noatp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

class *comm-ring* = *comm-semiring* + *ab-group-add*
begin

subclass *ring* ..
subclass *comm-semiring-0-cancel* ..

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* ..

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*

begin

subclass *ring-1* ..
subclass *comm-semiring-1-cancel* ..

lemma *dvd-minus-iff* [*simp*]: *x dvd* − *y* ⟷ *x dvd y*
proof
assume *x dvd* − *y*
then have *x dvd* − *1* ∗ − *y* **by** (*rule dvd-mult*)
then show *x dvd y* **by** *simp*
next
assume *x dvd y*
then have *x dvd* − *1* ∗ *y* **by** (*rule dvd-mult*)
then show *x dvd* − *y* **by** *simp*
qed

lemma *minus-dvd-iff* [*simp*]: − *x dvd y* ⟷ *x dvd y*
proof
assume − *x dvd y*
then obtain *k* **where** *y* = − *x* ∗ *k* ..
then have *y* = *x* ∗ − *k* **by** *simp*
then show *x dvd y* ..
next
assume *x dvd y*
then obtain *k* **where** *y* = *x* ∗ *k* ..

then have $y = - x * - k$ **by** *simp*
then show $- x$ *dvd* y **..**
qed

lemma *dvd-diff*[*simp*]: x *dvd* $y \Longrightarrow x$ *dvd* $z \Longrightarrow x$ *dvd* $(y - z)$
by (*simp add*: *diff-minus dvd-minus-iff*)

end

class *ring-no-zero-divisors* = *ring* + *no-zero-divisors*
begin

lemma *mult-eq-0-iff* [*simp*]:
shows $a * b = 0 \longleftrightarrow (a = 0 \vee b = 0)$
proof (*cases* $a = 0 \vee b = 0$)
case *False* **then have** $a \neq 0$ **and** $b \neq 0$ **by** *auto*
then show *?thesis* **using** *no-zero-divisors* **by** *simp*
next
case *True* **then show** *?thesis* **by** *auto*
qed

Cancellation of equalities with a common factor

lemma *mult-cancel-right* [*simp*, *noatp*]:
$a * c = b * c \longleftrightarrow c = 0 \vee a = b$
proof −
have $(a * c = b * c) = ((a - b) * c = 0)$
by (*simp add*: *algebra-simps right-minus-eq*)
thus *?thesis* **by** (*simp add*: *disj-commute right-minus-eq*)
qed

lemma *mult-cancel-left* [*simp*, *noatp*]:
$c * a = c * b \longleftrightarrow c = 0 \vee a = b$
proof −
have $(c * a = c * b) = (c * (a - b) = 0)$
by (*simp add*: *algebra-simps right-minus-eq*)
thus *?thesis* **by** (*simp add*: *right-minus-eq*)
qed

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

lemma *mult-cancel-right1* [*simp*]:
$c = b * c \longleftrightarrow c = 0 \vee b = 1$
by (*insert mult-cancel-right* [*of 1 c b*], *force*)

lemma *mult-cancel-right2* [*simp*]:
$a * c = c \longleftrightarrow c = 0 \vee a = 1$

by (*insert mult-cancel-right* [*of a c 1*], *simp*)

lemma *mult-cancel-left1* [*simp*]:
  $c = c * b \longleftrightarrow c = 0 \vee b = 1$
by (*insert mult-cancel-left* [*of c 1 b*], *force*)

lemma *mult-cancel-left2* [*simp*]:
  $c * a = c \longleftrightarrow c = 0 \vee a = 1$
by (*insert mult-cancel-left* [*of c a 1*], *simp*)

end

class *idom* = *comm-ring-1* + *no-zero-divisors*
begin

subclass *ring-1-no-zero-divisors* **..**

lemma *square-eq-iff*: $a * a = b * b \longleftrightarrow (a = b \vee a = - b)$
proof
  assume $a * a = b * b$
  then have $(a - b) * (a + b) = 0$
    by (*simp add*: *algebra-simps*)
  then show $a = b \vee a = - b$
    by (*simp add*: *right-minus-eq eq-neg-iff-add-eq-0*)
next
  assume $a = b \vee a = - b$
  then show $a * a = b * b$ **by** *auto*
qed

lemma *dvd-mult-cancel-right* [*simp*]:
  $a * c \; dvd \; b * c \longleftrightarrow c = 0 \vee a \; dvd \; b$
proof −
  have $a * c \; dvd \; b * c \longleftrightarrow (\exists k.\ b * c = (a * k) * c)$
    unfolding *dvd-def* **by** (*simp add*: *mult-ac*)
  also have $(\exists k.\ b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \; dvd \; b$
    unfolding *dvd-def* **by** *simp*
  finally show *?thesis* **.**
qed

lemma *dvd-mult-cancel-left* [*simp*]:
  $c * a \; dvd \; c * b \longleftrightarrow c = 0 \vee a \; dvd \; b$
proof −
  have $c * a \; dvd \; c * b \longleftrightarrow (\exists k.\ b * c = (a * k) * c)$
    unfolding *dvd-def* **by** (*simp add*: *mult-ac*)
  also have $(\exists k.\ b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \; dvd \; b$
    unfolding *dvd-def* **by** *simp*
  finally show *?thesis* **.**
qed

end

class *division-ring* = *ring-1* + *inverse* +
  assumes *left-inverse* [*simp*]: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
  assumes *right-inverse* [*simp*]: $a \neq 0 \Longrightarrow a * inverse\ a = 1$
begin

subclass *ring-1-no-zero-divisors*
proof
  fix $a\ b :: 'a$
  assume *a*: $a \neq 0$ **and** *b*: $b \neq 0$
  show $a * b \neq 0$
  proof
    assume *ab*: $a * b = 0$
    hence $0 = inverse\ a * (a * b) * inverse\ b$ **by** *simp*
    also have $\ldots = (inverse\ a * a) * (b * inverse\ b)$
      by (*simp only*: *mult-assoc*)
    also have $\ldots = 1$ **using** *a b* **by** *simp*
    finally show *False* **by** *simp*
  qed
qed

lemma *nonzero-imp-inverse-nonzero*:
  $a \neq 0 \Longrightarrow inverse\ a \neq 0$
proof
  assume *ianz*: $inverse\ a = 0$
  assume $a \neq 0$
  hence $1 = a * inverse\ a$ **by** *simp*
  also have ... $= 0$ **by** (*simp add*: *ianz*)
  finally have $1 = 0$ **.**
  thus *False* **by** (*simp add*: *eq-commute*)
qed

lemma *inverse-zero-imp-zero*:
  $inverse\ a = 0 \Longrightarrow a = 0$
apply (*rule classical*)
apply (*drule nonzero-imp-inverse-nonzero*)
apply *auto*
done

lemma *inverse-unique*:
  assumes *ab*: $a * b = 1$
  shows $inverse\ a = b$
proof −
  have $a \neq 0$ **using** *ab* **by** (*cases* $a = 0$) *simp-all*
  moreover have $inverse\ a * (a * b) = inverse\ a$ **by** (*simp add*: *ab*)
  ultimately show *?thesis* **by** (*simp add*: *mult-assoc* [*symmetric*])
qed

lemma *nonzero-inverse-minus-eq*:
  $a \neq 0 \Longrightarrow inverse\ (-\ a) = -\ inverse\ a$
by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-inverse-eq*:
  $a \neq 0 \Longrightarrow inverse\ (inverse\ a) = a$
by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-eq-imp-eq*:
  assumes $inverse\ a = inverse\ b$ **and** $a \neq 0$ **and** $b \neq 0$
  shows $a = b$
proof −
  from ‹$inverse\ a = inverse\ b$›
  have $inverse\ (inverse\ a) = inverse\ (inverse\ b)$ **by** (*rule arg-cong*)
  with ‹$a \neq 0$› **and** ‹$b \neq 0$› **show** $a = b$
    by (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-1* [*simp*]: $inverse\ 1 = 1$
by (*rule inverse-unique*) *simp*

lemma *nonzero-inverse-mult-distrib*:
  assumes $a \neq 0$ **and** $b \neq 0$
  shows $inverse\ (a * b) = inverse\ b * inverse\ a$
proof −
  have $a * (b * inverse\ b) * inverse\ a = 1$ **using** *assms* **by** *simp*
  hence $a * b * (inverse\ b * inverse\ a) = 1$ **by** (*simp only*: *mult-assoc*)
  thus *?thesis* **by** (*rule inverse-unique*)
qed

lemma *division-ring-inverse-add*:
  $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ a + inverse\ b = inverse\ a * (a + b) * inverse\ b$
by (*simp add*: *algebra-simps*)

lemma *division-ring-inverse-diff*:
  $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ a - inverse\ b = inverse\ a * (b - a) * inverse\ b$
by (*simp add*: *algebra-simps*)

end

class *field* = *comm-ring-1* + *inverse* +
  assumes *field-inverse*: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
  assumes *divide-inverse*: $a\ /\ b = a * inverse\ b$
begin

subclass *division-ring*
proof
  fix $a :: 'a$
  assume $a \neq 0$

thus *inverse a * a = 1* **by** (*rule field-inverse*)
thus *a * inverse a = 1* **by** (*simp only*: *mult-commute*)
qed

subclass *idom* **..**

lemma *right-inverse-eq*: *b ≠ 0 ⟹ a / b = 1 ⟷ a = b*
proof
assume *neq*: *b ≠ 0*
{
hence *a = (a / b) * b* **by** (*simp add*: *divide-inverse mult-ac*)
also assume *a / b = 1*
finally show *a = b* **by** *simp*
next
assume *a = b*
with *neq* **show** *a / b = 1* **by** (*simp add*: *divide-inverse*)
}
qed

lemma *nonzero-inverse-eq-divide*: *a ≠ 0 ⟹ inverse a = 1 / a*
by (*simp add*: *divide-inverse*)

lemma *divide-self* [*simp*]: *a ≠ 0 ⟹ a / a = 1*
by (*simp add*: *divide-inverse*)

lemma *divide-zero-left* [*simp*]: *0 / a = 0*
by (*simp add*: *divide-inverse*)

lemma *inverse-eq-divide*: *inverse a = 1 / a*
by (*simp add*: *divide-inverse*)

lemma *add-divide-distrib*: *(a+b) / c = a/c + b/c*
by (*simp add*: *divide-inverse algebra-simps*)

There is no slick version using division by zero.

lemma *inverse-add*:
[| a ≠ 0; b ≠ 0 |]
*==> inverse a + inverse b = (a + b) * inverse a * inverse b*
by (*simp add*: *division-ring-inverse-add mult-ac*)

lemma *nonzero-mult-divide-mult-cancel-left* [*simp*, *noatp*]:
assumes [*simp*]: *b≠0* **and** [*simp*]: *c≠0* **shows** *(c*a)/(c*b) = a/b*
proof −
have *(c*a)/(c*b) = c * a * (inverse b * inverse c)*
by (*simp add*: *divide-inverse nonzero-inverse-mult-distrib*)
also have *... = a * inverse b * (inverse c * c)*
by (*simp only*: *mult-ac*)
also have *... = a * inverse b* **by** *simp*
finally show *?thesis* **by** (*simp add*: *divide-inverse*)

qed

lemma *nonzero-mult-divide-mult-cancel-right* [*simp*, *noatp*]:
$[\![b \neq 0;\ c \neq 0]\!] \Longrightarrow (a * c) \mathbin{/} (b * c) = a \mathbin{/} b$
by (*simp add*: *mult-commute* [*of* - *c*])

lemma *divide-1* [*simp*]: $a \mathbin{/} 1 = a$
by (*simp add*: *divide-inverse*)

lemma *times-divide-eq-right*: $a * (b \mathbin{/} c) = (a * b) \mathbin{/} c$
by (*simp add*: *divide-inverse mult-assoc*)

lemma *times-divide-eq-left*: $(b \mathbin{/} c) * a = (b * a) \mathbin{/} c$
by (*simp add*: *divide-inverse mult-ac*)

These are later declared as simp rules.

lemmas *times-divide-eq* [*noatp*] = *times-divide-eq-right times-divide-eq-left*

lemma *add-frac-eq*:
assumes $y \neq 0$ **and** $z \neq 0$
shows $x \mathbin{/} y + w \mathbin{/} z = (x * z + w * y) \mathbin{/} (y * z)$
proof −
have $x \mathbin{/} y + w \mathbin{/} z = (x * z) \mathbin{/} (y * z) + (y * w) \mathbin{/} (y * z)$
using *assms* **by** *simp*
also have $\ldots = (x * z + y * w) \mathbin{/} (y * z)$
by (*simp only*: *add-divide-distrib*)
finally show *?thesis*
by (*simp only*: *mult-commute*)
qed

Special Cancellation Simprules for Division

lemma *nonzero-mult-divide-cancel-right* [*simp*, *noatp*]:
$b \neq 0 \Longrightarrow a * b \mathbin{/} b = a$
using *nonzero-mult-divide-mult-cancel-right* [*of 1 b a*] **by** *simp*

lemma *nonzero-mult-divide-cancel-left* [*simp*, *noatp*]:
$a \neq 0 \Longrightarrow a * b \mathbin{/} a = b$
using *nonzero-mult-divide-mult-cancel-left* [*of 1 a b*] **by** *simp*

lemma *nonzero-divide-mult-cancel-right* [*simp*, *noatp*]:
$[\![a \neq 0;\ b \neq 0]\!] \Longrightarrow b \mathbin{/} (a * b) = 1 \mathbin{/} a$
using *nonzero-mult-divide-mult-cancel-right* [*of a b 1*] **by** *simp*

lemma *nonzero-divide-mult-cancel-left* [*simp*, *noatp*]:
$[\![a \neq 0;\ b \neq 0]\!] \Longrightarrow a \mathbin{/} (a * b) = 1 \mathbin{/} b$
using *nonzero-mult-divide-mult-cancel-left* [*of b a 1*] **by** *simp*

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*, *noatp*]:
$[\![b \neq 0;\ c \neq 0]\!] \Longrightarrow (c * a) \mathbin{/} (b * c) = a \mathbin{/} b$

using *nonzero-mult-divide-mult-cancel-left* [*of b c a*] **by** (*simp add*: *mult-ac*)

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*, *noatp*]:
$[\![b \neq 0; c \neq 0]\!] \Longrightarrow (a * c) / (c * b) = a / b$
using *nonzero-mult-divide-mult-cancel-right* [*of b c a*] **by** (*simp add*: *mult-ac*)

lemma *minus-divide-left*: $-(a / b) = (-a) / b$
by (*simp add*: *divide-inverse*)

lemma *nonzero-minus-divide-right*: $b \neq 0 ==> -(a / b) = a / (-b)$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *nonzero-minus-divide-divide*: $b \neq 0 ==> (-a) / (-b) = a / b$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *divide-minus-left* [*simp*, *noatp*]: $(-a) / b = -(a / b)$
by (*simp add*: *divide-inverse*)

lemma *diff-divide-distrib*: $(a - b) / c = a / c - b / c$
by (*simp add*: *diff-minus add-divide-distrib*)

lemma *add-divide-eq-iff*:
$z \neq 0 \Longrightarrow x + y / z = (z * x + y) / z$
by (*simp add*: *add-divide-distrib*)

lemma *divide-add-eq-iff*:
$z \neq 0 \Longrightarrow x / z + y = (x + z * y) / z$
by (*simp add*: *add-divide-distrib*)

lemma *diff-divide-eq-iff*:
$z \neq 0 \Longrightarrow x - y / z = (z * x - y) / z$
by (*simp add*: *diff-divide-distrib*)

lemma *divide-diff-eq-iff*:
$z \neq 0 \Longrightarrow x / z - y = (x - z * y) / z$
by (*simp add*: *diff-divide-distrib*)

lemma *nonzero-eq-divide-eq*: $c \neq 0 \Longrightarrow a = b / c \longleftrightarrow a * c = b$
proof −
assume [*simp*]: $c \neq 0$
have $a = b / c \longleftrightarrow a * c = (b / c) * c$ **by** *simp*
also have ... $\longleftrightarrow a * c = b$ **by** (*simp add*: *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *nonzero-divide-eq-eq*: $c \neq 0 \Longrightarrow b / c = a \longleftrightarrow b = a * c$
proof −
assume [*simp*]: $c \neq 0$
have $b / c = a \longleftrightarrow (b / c) * c = a * c$ **by** *simp*

also have ... $\longleftrightarrow$ $b = a * c$ **by** (*simp add*: *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *divide-eq-imp*: $c \neq 0 \Longrightarrow b = a * c \Longrightarrow b \mathbin{/} c = a$
by *simp*

lemma *eq-divide-imp*: $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b \mathbin{/} c$
by (*erule subst*, *simp*)

lemmas *field-eq-simps*[*noatp*] = *algebra-simps*

add-divide-eq-iff divide-add-eq-iff
diff-divide-eq-iff divide-diff-eq-iff

nonzero-eq-divide-eq nonzero-divide-eq-eq

An example:

lemma $[\![a{\neq}b;\ c{\neq}d;\ e{\neq}f]\!] \Longrightarrow ((a-b)*(c-d)*(e-f))/((c-d)*(e-f)*(a-b)) = 1$
apply(*subgoal-tac* $(c-d)*(e-f)*(a-b) \neq 0$)
apply(*simp add*:*field-eq-simps*)
apply(*simp*)
done

lemma *diff-frac-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y - w \mathbin{/} z = (x * z - w * y) \mathbin{/} (y * z)$
by (*simp add*: *field-eq-simps times-divide-eq*)

lemma *frac-eq-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow (x \mathbin{/} y = w \mathbin{/} z) = (x * z = w * y)$
by (*simp add*: *field-eq-simps times-divide-eq*)

end

class *division-by-zero* = *zero* + *inverse* +
assumes *inverse-zero* [*simp*]: *inverse 0* = *0*

lemma *divide-zero* [*simp*]:
$a \mathbin{/} 0 = (0::'a::\{field, division\text{-}by\text{-}zero\})$
by (*simp add*: *divide-inverse*)

lemma *divide-self-if* [*simp*]:
$a \mathbin{/} (a::'a::\{field, division\text{-}by\text{-}zero\}) = (if\ a{=}0\ then\ 0\ else\ 1)$
by *simp*

class *mult-mono* = *times* + *zero* + *ord* +
assumes *mult-left-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
assumes *mult-right-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow a * c \leq b * c$

class *pordered-semiring* = *mult-mono* + *semiring-0* + *pordered-ab-semigroup-add*

begin

lemma *mult-mono*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq b \Longrightarrow 0 \leq c$
$\Longrightarrow a * c \leq b * d$
apply (*erule mult-right-mono* [*THEN order-trans*], *assumption*)
apply (*erule mult-left-mono*, *assumption*)
done

lemma *mult-mono'*:
$a \leq b \Longrightarrow c \leq d \Longrightarrow 0 \leq a \Longrightarrow 0 \leq c$
$\Longrightarrow a * c \leq b * d$
apply (*rule mult-mono*)
apply (*fast intro*: *order-trans*)+
done

end

class *pordered-cancel-semiring* = *mult-mono* + *pordered-ab-semigroup-add*
+ *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* **..**
subclass *pordered-semiring* **..**

lemma *mult-nonneg-nonneg*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * b$
using *mult-left-mono* [*of zero b a*] **by** *simp*

lemma *mult-nonneg-nonpos*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow a * b \leq 0$
using *mult-left-mono* [*of b zero a*] **by** *simp*

lemma *mult-nonpos-nonneg*: $a \leq 0 \Longrightarrow 0 \leq b \Longrightarrow a * b \leq 0$
using *mult-right-mono* [*of a zero b*] **by** *simp*

Legacy - use *mult-nonpos-nonneg*

lemma *mult-nonneg-nonpos2*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow b * a \leq 0$
by (*drule mult-right-mono* [*of b zero*], *auto*)

lemma *split-mult-neg-le*: $(0 \leq a \;\&\; b \leq 0) \mid (a \leq 0 \;\&\; 0 \leq b) \Longrightarrow a * b \leq 0$
by (*auto simp add*: *mult-nonneg-nonpos mult-nonneg-nonpos2*)

end

class *ordered-semiring* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add*
+ *mult-mono*
begin

subclass *pordered-cancel-semiring* **..**

subclass *pordered-comm-monoid-add* **..**

lemma *mult-left-less-imp-less*:
  $c * a < c * b \Longrightarrow 0 \leq c \Longrightarrow a < b$
by (*force simp add*: *mult-left-mono not-le* [*symmetric*])

lemma *mult-right-less-imp-less*:
  $a * c < b * c \Longrightarrow 0 \leq c \Longrightarrow a < b$
by (*force simp add*: *mult-right-mono not-le* [*symmetric*])

end

class *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add* +
  assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
  assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* **..**

subclass *ordered-semiring*
proof
  fix *a b c* :: $'a$
  assume *A*: $a \leq b$ $0 \leq c$
  from *A* **show** $c * a \leq c * b$
    unfolding *le-less*
    using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
  from *A* **show** $a * c \leq b * c$
    unfolding *le-less*
    using *mult-strict-right-mono* **by** (*cases* $c = 0$) *auto*
qed

lemma *mult-left-le-imp-le*:
  $c * a \leq c * b \Longrightarrow 0 < c \Longrightarrow a \leq b$
by (*force simp add*: *mult-strict-left-mono -not-less* [*symmetric*])

lemma *mult-right-le-imp-le*:
  $a * c \leq b * c \Longrightarrow 0 < c \Longrightarrow a \leq b$
by (*force simp add*: *mult-strict-right-mono not-less* [*symmetric*])

lemma *mult-pos-pos*: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
using *mult-strict-left-mono* [*of zero b a*] **by** *simp*

lemma *mult-pos-neg*: $0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$
using *mult-strict-left-mono* [*of b zero a*] **by** *simp*

lemma *mult-neg-pos*: $a < 0 \Longrightarrow 0 < b \Longrightarrow a * b < 0$

using *mult-strict-right-mono* [*of a zero b*] **by** *simp*

Legacy - use *mult-neg-pos*

lemma *mult-pos-neg2*: $0 < a \Longrightarrow b < 0 \Longrightarrow b * a < 0$
by (*drule mult-strict-right-mono* [*of b zero*], *auto*)

lemma *zero-less-mult-pos*:
$0 < a * b \Longrightarrow 0 < a \Longrightarrow 0 < b$
apply (*cases b≤0*)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

lemma *zero-less-mult-pos2*:
$0 < b * a \Longrightarrow 0 < a \Longrightarrow 0 < b$
apply (*cases b≤0*)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg2* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes $a < b$ **and** $c < d$ **and** $0 < b$ **and** $0 \leq c$
shows $a * c < b * d$
using *assms* **apply** (*cases c=0*)
apply (*simp add*: *mult-pos-pos*)
apply (*erule mult-strict-right-mono* [*THEN less-trans*])
apply (*force simp add*: *le-less*)
apply (*erule mult-strict-left-mono*, *assumption*)
done

This weaker variant has more natural premises

lemma *mult-strict-mono'*:
assumes $a < b$ **and** $c < d$ **and** $0 \leq a$ **and** $0 \leq c$
shows $a * c < b * d$
by (*rule mult-strict-mono*) (*insert assms*, *auto*)

lemma *mult-less-le-imp-less*:
assumes $a < b$ **and** $c \leq d$ **and** $0 \leq a$ **and** $0 < c$
shows $a * c < b * d$
using *assms* **apply** (*subgoal-tac* $a * c < b * c$)
apply (*erule less-le-trans*)
apply (*erule mult-left-mono*)
apply *simp*
apply (*erule mult-strict-right-mono*)
apply *assumption*
done

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
using *assms* **apply** (*subgoal-tac* $a * c \leq b * c$)
apply (*erule le-less-trans*)
apply (*erule mult-strict-left-mono*)
apply *simp*
apply (*erule mult-right-mono*)
apply *simp*
done

lemma *mult-less-imp-less-left*:
assumes *less*: $c * a < c * b$ **and** *nonneg*: $0 \leq c$
shows $a < b$
proof (*rule ccontr*)
assume $\neg\ a < b$
hence $b \leq a$ **by** (*simp add*: *linorder-not-less*)
hence $c * b \leq c * a$ **using** *nonneg* **by** (*rule mult-left-mono*)
with *this* **and** *less* **show** *False* **by** (*simp add*: *not-less* [*symmetric*])
qed

lemma *mult-less-imp-less-right*:
assumes *less*: $a * c < b * c$ **and** *nonneg*: $0 \leq c$
shows $a < b$
proof (*rule ccontr*)
assume $\neg\ a < b$
hence $b \leq a$ **by** (*simp add*: *linorder-not-less*)
hence $b * c \leq a * c$ **using** *nonneg* **by** (*rule mult-right-mono*)
with *this* **and** *less* **show** *False* **by** (*simp add*: *not-less* [*symmetric*])
qed

end

class *mult-mono1* = *times* + *zero* + *ord* +
assumes *mult-mono1*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$

class *pordered-comm-semiring* = *comm-semiring-0*
+ *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-semiring*
proof
fix *a b c* :: $'a$
assume $a \leq b$ $0 \leq c$
thus $c * a \leq c * b$ **by** (*rule mult-mono1*)
thus $a * c \leq b * c$ **by** (*simp only*: *mult-commute*)
qed

end

class *pordered-cancel-comm-semiring* = *comm-semiring-0-cancel*
+ *pordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *pordered-comm-semiring* ..
subclass *pordered-cancel-semiring* ..

end

class *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add*
+
assumes *mult-strict-left-mono-comm*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *ordered-semiring-strict*
proof
fix *a b c* :: $'a$
assume $a < b$ $0 < c$
thus $c * a < c * b$ **by** (*rule mult-strict-left-mono-comm*)
thus $a * c < b * c$ **by** (*simp only*: *mult-commute*)
qed

subclass *pordered-cancel-comm-semiring*
proof
fix *a b c* :: $'a$
assume $a \leq b$ $0 \leq c$
thus $c * a \leq c * b$
unfolding *le-less*
using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
qed

end

class *pordered-ring* = *ring* + *pordered-cancel-semiring*
begin

subclass *pordered-ab-group-add* ..

Legacy - use *algebra-simps*

lemmas *ring-simps*[*noatp*] = *algebra-simps*

lemma *less-add-iff1*:
$a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$
by (*simp add*: *algebra-simps*)

lemma *less-add-iff2*:
$a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$

by (*simp add*: *algebra-simps*)

lemma *le-add-iff1*:
  $a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$
by (*simp add*: *algebra-simps*)

lemma *le-add-iff2*:
  $a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$
by (*simp add*: *algebra-simps*)

lemma *mult-left-mono-neg*:
  $b \leq a \Longrightarrow c \leq 0 \Longrightarrow c * a \leq c * b$
  apply (*drule mult-left-mono* [*of* - - *uminus c*])
  apply (*simp-all add*: *minus-mult-left* [*symmetric*])
  done

lemma *mult-right-mono-neg*:
  $b \leq a \Longrightarrow c \leq 0 \Longrightarrow a * c \leq b * c$
  apply (*drule mult-right-mono* [*of* - - *uminus c*])
  apply (*simp-all add*: *minus-mult-right* [*symmetric*])
  done

lemma *mult-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a * b$
using *mult-right-mono-neg* [*of a zero b*] **by** *simp*

lemma *split-mult-pos-le*:
  $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \Longrightarrow 0 \leq a * b$
by (*auto simp add*: *mult-nonneg-nonneg mult-nonpos-nonpos*)

end

class *abs-if* = *minus* + *uminus* + *ord* + *zero* + *abs* +
  assumes *abs-if*: $|a| = (if\ a < 0\ then\ -a\ else\ a)$

class *sgn-if* = *minus* + *uminus* + *zero* + *one* + *ord* + *sgn* +
  assumes *sgn-if*: $sgn\ x = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -1)$

lemma (**in** *sgn-if*) *sgn0*[*simp*]: *sgn 0 = 0*
by(*simp add*:*sgn-if*)

class *ordered-ring* = *ring* + *ordered-semiring*
  + *ordered-ab-group-add* + *abs-if*
begin

subclass *pordered-ring* **..**

subclass *pordered-ab-group-add-abs*
proof
  fix *a b*

show $|a + b| \leq |a| + |b|$
by (*auto simp add*: *abs-if not-less neg-less-eq-nonneg less-eq-neg-nonpos*)
(*auto simp del*: *minus-add-distrib simp add*: *minus-add-distrib* [*symmetric*]
neg-less-eq-nonneg less-eq-neg-nonpos, *auto intro*: *add-nonneg-nonneg*,
auto intro!: *less-imp-le add-neg-neg*)
qed (*auto simp add*: *abs-if less-eq-neg-nonpos neg-equal-zero*)

end

class *ordered-ring-strict* = *ring* + *ordered-semiring-strict*
+ *ordered-ab-group-add* + *abs-if*
begin

subclass *ordered-ring* **..**

lemma *mult-strict-left-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow c * a < c * b$
using *mult-strict-left-mono* [*of b a* $-$ *c*] **by** *simp*

lemma *mult-strict-right-mono-neg*: $b < a \Longrightarrow c < 0 \Longrightarrow a * c < b * c$
using *mult-strict-right-mono* [*of b a* $-$ *c*] **by** *simp*

lemma *mult-neg-neg*: $a < 0 \Longrightarrow b < 0 \Longrightarrow 0 < a * b$
using *mult-strict-right-mono-neg* [*of a zero b*] **by** *simp*

subclass *ring-no-zero-divisors*
proof
fix *a b*
assume $a \neq 0$ **then have** *A*: $a < 0 \vee 0 < a$ **by** (*simp add*: *neq-iff*)
assume $b \neq 0$ **then have** *B*: $b < 0 \vee 0 < b$ **by** (*simp add*: *neq-iff*)
have $a * b < 0 \vee 0 < a * b$
proof (*cases* $a < 0$)
case *True* **note** $A' = this$
show *?thesis* **proof** (*cases* $b < 0$)
case *True* **with** A'
show *?thesis* **by** (*auto dest*: *mult-neg-neg*)
next
case *False* **with** *B* **have** $0 < b$ **by** *auto*
with A' **show** *?thesis* **by** (*auto dest*: *mult-strict-right-mono*)
qed
next
case *False* **with** *A* **have** A': $0 < a$ **by** *auto*
show *?thesis* **proof** (*cases* $b < 0$)
case *True* **with** A'
show *?thesis* **by** (*auto dest*: *mult-strict-right-mono-neg*)
next
case *False* **with** *B* **have** $0 < b$ **by** *auto*
with A' **show** *?thesis* **by** (*auto dest*: *mult-pos-pos*)
qed

```
  qed
  then show a * b ≠ 0 by (simp add: neq-iff)
qed

lemma zero-less-mult-iff:
  0 < a * b ⟷ 0 < a ∧ 0 < b ∨ a < 0 ∧ b < 0
  apply (auto simp add: mult-pos-pos mult-neg-neg)
  apply (simp-all add: not-less le-less)
  apply (erule disjE) apply assumption defer
  apply (erule disjE) defer apply (drule sym) apply simp
  apply (erule disjE) defer apply (drule sym) apply simp
  apply (erule disjE) apply assumption apply (drule sym) apply simp
  apply (drule sym) apply simp
  apply (blast dest: zero-less-mult-pos)
  apply (blast dest: zero-less-mult-pos2)
  done

lemma zero-le-mult-iff:
  0 ≤ a * b ⟷ 0 ≤ a ∧ 0 ≤ b ∨ a ≤ 0 ∧ b ≤ 0
by (auto simp add: eq-commute [of 0] le-less not-less zero-less-mult-iff)

lemma mult-less-0-iff:
  a * b < 0 ⟷ 0 < a ∧ b < 0 ∨ a < 0 ∧ 0 < b
  apply (insert zero-less-mult-iff [of -a b])
  apply (force simp add: minus-mult-left[symmetric])
  done

lemma mult-le-0-iff:
  a * b ≤ 0 ⟷ 0 ≤ a ∧ b ≤ 0 ∨ a ≤ 0 ∧ 0 ≤ b
  apply (insert zero-le-mult-iff [of -a b])
  apply (force simp add: minus-mult-left[symmetric])
  done

lemma zero-le-square [simp]: 0 ≤ a * a
by (simp add: zero-le-mult-iff linear)

lemma not-square-less-zero [simp]: ¬ (a * a < 0)
by (simp add: not-less)
```

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations $\leq$ and equality.

These "disjunction" versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

```
lemma mult-less-cancel-right-disj:
  a * c < b * c ⟷ 0 < c ∧ a < b ∨ c < 0 ∧  b < a
  apply (cases c = 0)
  apply (auto simp add: neq-iff mult-strict-right-mono
                        mult-strict-right-mono-neg)
```

apply (*auto simp add*: *not-less*
not-le [*symmetric*, *of* $a*c$]
not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *less-imp-le mult-right-mono*
mult-right-mono-neg)
done

lemma *mult-less-cancel-left-disj*:
$c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge\ b < a$
apply (*cases* $c = 0$)
apply (*auto simp add*: *neq-iff mult-strict-left-mono*
mult-strict-left-mono-neg)
apply (*auto simp add*: *not-less*
not-le [*symmetric*, *of* $c*a$]
not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *less-imp-le mult-left-mono*
mult-left-mono-neg)
done

The "conjunction of implication" lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma *mult-less-cancel-right*:
$a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-right-disj* [*of* a c b] **by** *auto*

lemma *mult-less-cancel-left*:
$c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-left-disj* [*of* c a b] **by** *auto*

lemma *mult-le-cancel-right*:
$a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-right-disj*)

lemma *mult-le-cancel-left*:
$c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-left-disj*)

lemma *mult-le-cancel-left-pos*:
$0 < c \Longrightarrow c * a \leq c * b \longleftrightarrow a \leq b$
by (*auto simp*: *mult-le-cancel-left*)

lemma *mult-le-cancel-left-neg*:
$c < 0 \Longrightarrow c * a \leq c * b \longleftrightarrow b \leq a$
by (*auto simp*: *mult-le-cancel-left*)

lemma *mult-less-cancel-left-pos*:
$0 < c \Longrightarrow c * a < c * b \longleftrightarrow a < b$

by (*auto simp*: *mult-less-cancel-left*)

lemma *mult-less-cancel-left-neg*:
  $c < 0 \Longrightarrow c * a < c * b \longleftrightarrow b < a$
by (*auto simp*: *mult-less-cancel-left*)

end

Legacy - use *algebra-simps*

lemmas *ring-simps*[*noatp*] = *algebra-simps*

lemmas *mult-sign-intros* =
  mult-nonneg-nonneg mult-nonneg-nonpos
  mult-nonpos-nonneg mult-nonpos-nonpos
  mult-pos-pos mult-pos-neg
  mult-neg-pos mult-neg-neg

class *pordered-comm-ring* = *comm-ring* + *pordered-comm-semiring*
begin

subclass *pordered-ring* **..**
subclass *pordered-cancel-comm-semiring* **..**

end

class *ordered-semidom* = *comm-semiring-1-cancel* + *ordered-comm-semiring-strict*
+

  assumes *zero-less-one* [*simp*]: *0 < 1*
begin

lemma *pos-add-strict*:
  shows $0 < a \Longrightarrow b < c \Longrightarrow b < a + c$
  using *add-strict-mono* [*of zero a b c*] **by** *simp*

lemma *zero-le-one* [*simp*]: *0 ≤ 1*
by (*rule zero-less-one* [*THEN less-imp-le*])

lemma *not-one-le-zero* [*simp*]: ¬ *1 ≤ 0*
by (*simp add*: *not-le*)

lemma *not-one-less-zero* [*simp*]: ¬ *1 < 0*
by (*simp add*: *not-less*)

lemma *less-1-mult*:
  assumes *1 < m* **and** *1 < n*
  shows *1 < m * n*
  using *assms mult-strict-mono* [*of 1 m 1 n*]
    by (*simp add*: *less-trans* [*OF zero-less-one*])

end

class *ordered-idom* = *comm-ring-1* +
ordered-comm-semiring-strict + *ordered-ab-group-add* +
abs-if + *sgn-if*

begin

subclass *ordered-ring-strict* **..**
subclass *pordered-comm-ring* **..**
subclass *idom* **..**

subclass *ordered-semidom*
proof
have $0 \leq 1 * 1$ **by** (*rule zero-le-square*)
thus $0 < 1$ **by** (*simp add*: *le-less*)
qed

lemma *linorder-neqE-ordered-idom*:
assumes $x \neq y$ **obtains** $x < y \mid y < x$
using *assms* **by** (*rule neqE*)

These cancellation simprules also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*:
$c \leq b * c \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
by (*insert mult-le-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-le-cancel-right2*:
$a * c \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
by (*insert mult-le-cancel-right* [*of a c 1*], *simp*)

lemma *mult-le-cancel-left1*:
$c \leq c * b \longleftrightarrow (0 < c \longrightarrow 1 \leq b) \wedge (c < 0 \longrightarrow b \leq 1)$
by (*insert mult-le-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-le-cancel-left2*:
$c * a \leq c \longleftrightarrow (0 < c \longrightarrow a \leq 1) \wedge (c < 0 \longrightarrow 1 \leq a)$
by (*insert mult-le-cancel-left* [*of c a 1*], *simp*)

lemma *mult-less-cancel-right1*:
$c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
by (*insert mult-less-cancel-right* [*of 1 c b*], *simp*)

lemma *mult-less-cancel-right2*:
$a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
by (*insert mult-less-cancel-right* [*of a c 1*], *simp*)

lemma *mult-less-cancel-left1*:
  $c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
by (*insert mult-less-cancel-left* [*of c 1 b*], *simp*)

lemma *mult-less-cancel-left2*:
  $c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
by (*insert mult-less-cancel-left* [*of c a 1*], *simp*)

lemma *sgn-sgn* [*simp*]:
  sgn (*sgn a*) = *sgn a*
unfolding *sgn-if* **by** *simp*

lemma *sgn-0-0*:
  sgn a = *0* $\longleftrightarrow$ *a* = *0*
unfolding *sgn-if* **by** *simp*

lemma *sgn-1-pos*:
  sgn a = *1* $\longleftrightarrow$ *a* > *0*
unfolding *sgn-if* **by** (*simp add*: *neg-equal-zero*)

lemma *sgn-1-neg*:
  sgn a = − *1* $\longleftrightarrow$ *a* < *0*
unfolding *sgn-if* **by** (*auto simp add*: *equal-neg-zero*)

lemma *sgn-pos* [*simp*]:
  0 < *a* $\Longrightarrow$ *sgn a* = *1*
unfolding *sgn-1-pos* **.**

lemma *sgn-neg* [*simp*]:
  a < *0* $\Longrightarrow$ *sgn a* = − *1*
unfolding *sgn-1-neg* **.**

lemma *sgn-times*:
  sgn (*a* * *b*) = *sgn a* * *sgn b*
by (*auto simp add*: *sgn-if zero-less-mult-iff*)

lemma *abs-sgn*: *abs k* = *k* * *sgn k*
unfolding *sgn-if abs-if* **by** *auto*

lemma *sgn-greater* [*simp*]:
  0 < *sgn a* $\longleftrightarrow$ *0* < *a*
  unfolding *sgn-if* **by** *auto*

lemma *sgn-less* [*simp*]:
  sgn a < *0* $\longleftrightarrow$ *a* < *0*
  unfolding *sgn-if* **by** *auto*

lemma *abs-dvd-iff* [*simp*]: (*abs m*) *dvd k* $\longleftrightarrow$ *m dvd k*
  by (*simp add*: *abs-if*)

lemma *dvd-abs-iff* [*simp*]: *m dvd* (*abs k*) ⟷ *m dvd k*
by (*simp add*: *abs-if*)

end

class *ordered-field* = *field* + *ordered-idom*

Simprules for comparisons where common factors can be cancelled.

lemmas *mult-compare-simps*[*noatp*] =
mult-le-cancel-right mult-le-cancel-left
mult-le-cancel-right1 mult-le-cancel-right2
mult-le-cancel-left1 mult-le-cancel-left2
mult-less-cancel-right mult-less-cancel-left
mult-less-cancel-right1 mult-less-cancel-right2
mult-less-cancel-left1 mult-less-cancel-left2
mult-cancel-right mult-cancel-left
mult-cancel-right1 mult-cancel-right2
mult-cancel-left1 mult-cancel-left2

— FIXME continue localization here

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
(*inverse a* = *0*) = (*a* = (*0*::*'a*::{*division-ring*,*division-by-zero*}))
by (*force dest*: *inverse-zero-imp-zero*)

lemma *inverse-minus-eq* [*simp*]:
inverse(−*a*) = −*inverse*(*a*::*'a*::{*division-ring*,*division-by-zero*})
proof *cases*
assume *a*=*0* **thus** *?thesis* **by** (*simp add*: *inverse-zero*)
next
assume *a*≠*0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-minus-eq*)
qed

lemma *inverse-eq-imp-eq*:
inverse a = *inverse b* ==> *a* = (*b*::*'a*::{*division-ring*,*division-by-zero*})
apply (*cases a*=*0* | *b*=*0*)
apply (*force dest*!: *inverse-zero-imp-zero*
simp add: *eq-commute* [*of 0*::*'a*])
apply (*force dest*!: *nonzero-inverse-eq-imp-eq*)
done

lemma *inverse-eq-iff-eq* [*simp*]:
(*inverse a* = *inverse b*) = (*a* = (*b*::*'a*::{*division-ring*,*division-by-zero*}))
by (*force dest*!: *inverse-eq-imp-eq*)

lemma *inverse-inverse-eq* [*simp*]:
inverse(*inverse* (*a*::*'a*::{*division-ring*,*division-by-zero*})) = *a*

proof *cases*
assume *a=0* **thus** *?thesis* **by** *simp*
next
assume *a≠0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [*simp*]:
inverse(*a*∗*b*) = *inverse*(*a*) ∗ *inverse*(*b*::′*a*::{*field*,*division-by-zero*})
proof *cases*
assume *a* ≠ *0* & *b* ≠ *0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-mult-distrib mult-commute*)
next
assume ~ (*a* ≠ *0* & *b* ≠ *0*)
thus *?thesis* **by** *force*
qed

lemma *inverse-divide* [*simp*]:
inverse (*a*/*b*) = *b* / (*a*::′*a*::{*field*,*division-by-zero*})
by (*simp add*: *divide-inverse mult-commute*)

11.1 Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

lemma *mult-divide-mult-cancel-left*:
c≠0 ==> (*c*∗*a*) / (*c*∗*b*) = *a* / (*b*::′*a*::{*field*,*division-by-zero*})
apply (*cases b* = *0*)
apply (*simp-all add*: *nonzero-mult-divide-mult-cancel-left*)
done

lemma *mult-divide-mult-cancel-right*:
c≠0 ==> (*a*∗*c*) / (*b*∗*c*) = *a* / (*b*::′*a*::{*field*,*division-by-zero*})
apply (*cases b* = *0*)
apply (*simp-all add*: *nonzero-mult-divide-mult-cancel-right*)
done

lemma *divide-divide-eq-right* [*simp*,*noatp*]:
a / (*b*/*c*) = (*a*∗*c*) / (*b*::′*a*::{*field*,*division-by-zero*})
by (*simp add*: *divide-inverse mult-ac*)

lemma *divide-divide-eq-left* [*simp*,*noatp*]:
(*a* / *b*) / (*c*::′*a*::{*field*,*division-by-zero*}) = *a* / (*b*∗*c*)
by (*simp add*: *divide-inverse mult-assoc*)

11.1.1 Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if*[*simp*,*noatp*]:
fixes $c :: 'a :: \{field, division\text{-}by\text{-}zero\}$
shows $(c*a) \;/\; (c*b) = (if\ c=0\ then\ 0\ else\ a/b)$
by (*simp add*: *mult-divide-mult-cancel-left*)

11.2 Division and Unary Minus

lemma *minus-divide-right*: $-(a/b) = a \;/\; -(b::'a::\{field, division\text{-}by\text{-}zero\})$
by (*simp add*: *divide-inverse*)

lemma *divide-minus-right* [*simp*, *noatp*]:
$a \;/\; -(b::'a::\{field, division\text{-}by\text{-}zero\}) = -(a \;/\; b)$
by (*simp add*: *divide-inverse*)

lemma *minus-divide-divide*:
$(-a)/(-b) = a \;/\; (b::'a::\{field, division\text{-}by\text{-}zero\})$
apply (*cases* $b=0$, *simp*)
apply (*simp add*: *nonzero-minus-divide-divide*)
done

lemma *eq-divide-eq*:
$((a::'a::\{field, division\text{-}by\text{-}zero\}) = b/c) = (if\ c\neq 0\ then\ a*c = b\ else\ a=0)$
by (*simp add*: *nonzero-eq-divide-eq*)

lemma *divide-eq-eq*:
$(b/c = (a::'a::\{field, division\text{-}by\text{-}zero\})) = (if\ c\neq 0\ then\ b = a*c\ else\ a=0)$
by (*force simp add*: *nonzero-divide-eq-eq*)

11.3 Ordered Fields

lemma *positive-imp-inverse-positive*:
assumes *a-gt-0*: $0 < a$ **shows** $0 < inverse\ (a::'a::ordered\text{-}field)$
proof −
have $0 < a * inverse\ a$
by (*simp add*: *a-gt-0* [*THEN order-less-imp-not-eq2*] *zero-less-one*)
thus $0 < inverse\ a$
by (*simp add*: *a-gt-0* [*THEN order-less-not-sym*] *zero-less-mult-iff*)
qed

lemma *negative-imp-inverse-negative*:
$a < 0 \Longrightarrow inverse\ a < (0::'a::ordered\text{-}field)$
by (*insert positive-imp-inverse-positive* [*of* $-a$],
simp add: *nonzero-inverse-minus-eq order-less-imp-not-eq*)

lemma *inverse-le-imp-le*:
assumes *invle*: $inverse\ a \leq inverse\ b$ **and** *apos*: $0 < a$
shows $b \leq (a::'a::ordered\text{-}field)$
proof (*rule classical*)

assume $\sim\ b \leq a$
hence $a < b$ **by** (*simp add*: *linorder-not-le*)
hence *bpos*: $0 < b$ **by** (*blast intro*: *apos order-less-trans*)
hence $a * inverse\ a \leq a * inverse\ b$
by (*simp add*: *apos invle order-less-imp-le mult-left-mono*)
hence $(a * inverse\ a) * b \leq (a * inverse\ b) * b$
by (*simp add*: *bpos order-less-imp-le mult-right-mono*)
thus $b \leq a$ **by** (*simp add*: *mult-assoc apos bpos order-less-imp-not-eq2*)
qed

lemma *inverse-positive-imp-positive*:
assumes *inv-gt-0*: $0 < inverse\ a$ **and** *nz*: $a \neq 0$
shows $0 < (a::'a::ordered\text{-}field)$
proof −
have $0 < inverse\ (inverse\ a)$
using *inv-gt-0* **by** (*rule positive-imp-inverse-positive*)
thus $0 < a$
using *nz* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-positive-iff-positive* [*simp*]:
$(0 < inverse\ a) = (0 < (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
apply (*cases* $a = 0$, *simp*)
apply (*blast intro*: *inverse-positive-imp-positive positive-imp-inverse-positive*)
done

lemma *inverse-negative-imp-negative*:
assumes *inv-less-0*: $inverse\ a < 0$ **and** *nz*: $a \neq 0$
shows $a < (0::'a::ordered\text{-}field)$
proof −
have $inverse\ (inverse\ a) < 0$
using *inv-less-0* **by** (*rule negative-imp-inverse-negative*)
thus $a < 0$ **using** *nz* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-negative-iff-negative* [*simp*]:
$(inverse\ a < 0) = (a < (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
apply (*cases* $a = 0$, *simp*)
apply (*blast intro*: *inverse-negative-imp-negative negative-imp-inverse-negative*)
done

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
$(0 \leq inverse\ a) = (0 \leq (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
$(inverse\ a \leq 0) = (a \leq (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}))$
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *ordered-field-no-lb*: $\forall\ x.\ \exists\, y.\ y < (x::'a::ordered\text{-}field)$
proof
  fix $x::'a$
  have *m1*: $-\ (1::'a) < 0$ **by** *simp*
  from *add-strict-right-mono*[*OF m1*, **where** $c{=}x$]
  have $(-\ 1) + x < x$ **by** *simp*
  thus $\exists\, y.\ y < x$ **by** *blast*
qed

lemma *ordered-field-no-ub*: $\forall\ x.\ \exists\, y.\ y > (x::'a::ordered\text{-}field)$
proof
  fix $x::'a$
  have *m1*: $(1::'a) > 0$ **by** *simp*
  from *add-strict-right-mono*[*OF m1*, **where** $c{=}x$]
  have $1 + x > x$ **by** *simp*
  thus $\exists\, y.\ y > x$ **by** *blast*
qed

11.4 Anti-Monotonicity of *inverse*

lemma *less-imp-inverse-less*:
assumes *less*: $a < b$ **and** *apos*: $0 < a$
shows *inverse* $b <$ *inverse* $(a::'a::ordered\text{-}field)$
proof (*rule ccontr*)
  assume $\sim$ *inverse* $b <$ *inverse* a
  hence *inverse* $a \leq$ *inverse* b **by** (*simp add*: *linorder-not-less*)
  hence $\sim (a < b)$
    by (*simp add*: *linorder-not-less inverse-le-imp-le* [*OF* - *apos*])
  thus *False* **by** (*rule notE* [*OF* - *less*])
qed

lemma *inverse-less-imp-less*:
  $[|$*inverse* $a <$ *inverse* b; $0 < a|] ==> b < (a::'a::ordered\text{-}field)$
apply (*simp add*: *order-less-le* [*of inverse a*] *order-less-le* [*of b*])
apply (*force dest*!: *inverse-le-imp-le nonzero-inverse-eq-imp-eq*)
done

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*,*noatp*]:
  $[|0 < a;\ 0 < b|] ==>$ (*inverse* $a <$ *inverse* $b) = (b < (a::'a::ordered\text{-}field))$
by (*blast intro*: *less-imp-inverse-less dest*: *inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
  $[|a \leq b;\ 0 < a|] ==>$ *inverse* $b \leq$ *inverse* $(a::'a::ordered\text{-}field)$
by (*force simp add*: *order-le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le* [*simp*,*noatp*]:
 $[|0 < a;\ 0 < b|] ==>$ (*inverse* $a \leq$ *inverse* $b) = (b \leq (a::'a::ordered\text{-}field))$
by (*blast intro*: *le-imp-inverse-le dest*: *inverse-le-imp-le*)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
[|*inverse a* ≤ *inverse b*; *b* < *0*|] ==> *b* ≤ (*a*::'*a*::*ordered-field*)
apply (*rule classical*)
apply (*subgoal-tac a* < *0*)
prefer *2* **apply** (*force simp add*: *linorder-not-le intro*: *order-less-trans*)
apply (*insert inverse-le-imp-le* [*of* −*b* −*a*])
apply (*simp add*: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *less-imp-inverse-less-neg*:
[|*a* < *b*; *b* < *0*|] ==> *inverse b* < *inverse* (*a*::'*a*::*ordered-field*)
apply (*subgoal-tac a* < *0*)
prefer *2* **apply** (*blast intro*: *order-less-trans*)
apply (*insert less-imp-inverse-less* [*of* −*b* −*a*])
apply (*simp add*: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *inverse-less-imp-less-neg*:
[|*inverse a* < *inverse b*; *b* < *0*|] ==> *b* < (*a*::'*a*::*ordered-field*)
apply (*rule classical*)
apply (*subgoal-tac a* < *0*)
prefer *2*
apply (*force simp add*: *linorder-not-less intro*: *order-le-less-trans*)
apply (*insert inverse-less-imp-less* [*of* −*b* −*a*])
apply (*simp add*: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *inverse-less-iff-less-neg* [*simp*,*noatp*]:
[|*a* < *0*; *b* < *0*|] ==> (*inverse a* < *inverse b*) = (*b* < (*a*::'*a*::*ordered-field*))
apply (*insert inverse-less-iff-less* [*of* −*b* −*a*])
apply (*simp del*: *inverse-less-iff-less*
add: *order-less-imp-not-eq nonzero-inverse-minus-eq*)
done

lemma *le-imp-inverse-le-neg*:
[|*a* ≤ *b*; *b* < *0*|] ==> *inverse b* ≤ *inverse* (*a*::'*a*::*ordered-field*)
by (*force simp add*: *order-le-less less-imp-inverse-less-neg*)

lemma *inverse-le-iff-le-neg* [*simp*,*noatp*]:
[|*a* < *0*; *b* < *0*|] ==> (*inverse a* ≤ *inverse b*) = (*b* ≤ (*a*::'*a*::*ordered-field*))
by (*blast intro*: *le-imp-inverse-le-neg dest*: *inverse-le-imp-le-neg*)

11.5 Inverses and the Number One

lemma *one-less-inverse-iff*:
(*1* < *inverse x*) = (*0* < *x* & *x* < (*1*::'*a*::{*ordered-field*,*division-by-zero*}))
proof *cases*

```
  assume 0 < x
    with inverse-less-iff-less [OF zero-less-one, of x]
    show ?thesis by simp
next
  assume notless: ~ (0 < x)
  have ~ (1 < inverse x)
  proof
    assume 1 < inverse x
    also with notless have ... ≤ 0 by (simp add: linorder-not-less)
    also have ... < 1 by (rule zero-less-one)
    finally show False by auto
  qed
  with notless show ?thesis by simp
qed

lemma inverse-eq-1-iff [simp]:
  (inverse x = 1) = (x = (1::'a::{field,division-by-zero}))
by (insert inverse-eq-iff-eq [of x 1], simp)

lemma one-le-inverse-iff:
  (1 ≤ inverse x) = (0 < x & x ≤ (1::'a::{ordered-field,division-by-zero}))
by (force simp add: order-le-less one-less-inverse-iff zero-less-one
                    eq-commute [of 1])

lemma inverse-less-1-iff:
  (inverse x < 1) = (x ≤ 0 | 1 < (x::'a::{ordered-field,division-by-zero}))
by (simp add: linorder-not-le [symmetric] one-le-inverse-iff)

lemma inverse-le-1-iff:
  (inverse x ≤ 1) = (x ≤ 0 | 1 ≤ (x::'a::{ordered-field,division-by-zero}))
by (simp add: linorder-not-less [symmetric] one-less-inverse-iff)
```

11.6 Simplification of Inequalities Involving Literal Divisors

```
lemma pos-le-divide-eq: 0 < (c::'a::ordered-field) ==> (a ≤ b/c) = (a*c ≤ b)
proof -
  assume less: 0<c
  hence (a ≤ b/c) = (a*c ≤ (b/c)*c)
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
  also have ... = (a*c ≤ b)
    by (simp add: order-less-imp-not-eq2 [OF less] divide-inverse mult-assoc)
  finally show ?thesis .
qed

lemma neg-le-divide-eq: c < (0::'a::ordered-field) ==> (a ≤ b/c) = (b ≤ a*c)
proof -
  assume less: c<0
  hence (a ≤ b/c) = ((b/c)*c ≤ a*c)
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
```

```
  also have ... = (b ≤ a*c)
    by (simp add: order-less-imp-not-eq [OF less] divide-inverse mult-assoc)
  finally show ?thesis .
qed

lemma le-divide-eq:
  (a ≤ b/c) =
   (if 0 < c then a*c ≤ b
            else if c < 0 then b ≤ a*c
            else  a ≤ (0::'a::{ordered-field,division-by-zero}))
apply (cases c=0, simp)
apply (force simp add: pos-le-divide-eq neg-le-divide-eq linorder-neq-iff)
done

lemma pos-divide-le-eq: 0 < (c::'a::ordered-field) ==> (b/c ≤ a) = (b ≤ a*c)
proof -
  assume less: 0<c
  hence (b/c ≤ a) = ((b/c)*c ≤ a*c)
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
  also have ... = (b ≤ a*c)
    by (simp add: order-less-imp-not-eq2 [OF less] divide-inverse mult-assoc)
  finally show ?thesis .
qed

lemma neg-divide-le-eq: c < (0::'a::ordered-field) ==> (b/c ≤ a) = (a*c ≤ b)
proof -
  assume less: c<0
  hence (b/c ≤ a) = (a*c ≤ (b/c)*c)
    by (simp add: mult-le-cancel-right order-less-not-sym [OF less])
  also have ... = (a*c ≤ b)
    by (simp add: order-less-imp-not-eq [OF less] divide-inverse mult-assoc)
  finally show ?thesis .
qed

lemma divide-le-eq:
  (b/c ≤ a) =
   (if 0 < c then b ≤ a*c
            else if c < 0 then a*c ≤ b
            else 0 ≤ (a::'a::{ordered-field,division-by-zero}))
apply (cases c=0, simp)
apply (force simp add: pos-divide-le-eq neg-divide-le-eq linorder-neq-iff)
done

lemma pos-less-divide-eq:
     0 < (c::'a::ordered-field) ==> (a < b/c) = (a*c < b)
proof -
  assume less: 0<c
  hence (a < b/c) = (a*c < (b/c)*c)
    by (simp add: mult-less-cancel-right-disj order-less-not-sym [OF less])
```

also have ... = $(a*c < b)$
by (*simp add*: *order-less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-less-divide-eq*:
$c < (0::'a::ordered\text{-}field) \Longrightarrow (a < b/c) = (b < a*c)$
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*:
$0 < (c::'a::ordered\text{-}field) \Longrightarrow (b/c < a) = (b < a*c)$
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*:
$c < (0::'a::ordered\text{-}field) \Longrightarrow (b/c < a) = (a*c < b)$
proof −
assume *less*: *c<0*
hence $(b/c < a) = (a*c < (b/c)*c)$
by (*simp add*: *mult-less-cancel-right-disj order-less-not-sym* [*OF less*])
also have ... = $(a*c < b)$
by (*simp add*: *order-less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *divide-less-eq*:
$(b/c < a) =$

(*if* $0 < c$ *then* $b < a*c$
else if $c < 0$ *then* $a*c < b$
else $0 < (a::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-divide-less-eq neg-divide-less-eq linorder-neq-iff*)
done

11.7 Field simplification

Lemmas *field-simps* multiply with denominators in in(equations) if they can be proved to be non-zero (for equations) or positive/negative (for inequations). Can be too aggressive and is therefore separate from the more benign *algebra-simps*.

lemmas *field-simps*[*noatp*] = *field-eq-simps*

pos-divide-less-eq neg-divide-less-eq
pos-less-divide-eq neg-less-divide-eq
pos-divide-le-eq neg-divide-le-eq
pos-le-divide-eq neg-le-divide-eq

Lemmas *sign-simps* is a first attempt to automate proofs of positivity/negativity needed for *field-simps*. Have not added *sign-simps* to *field-simps* because the former can lead to case explosions.

lemmas *sign-simps*[*noatp*] = *group-simps*
zero-less-mult-iff *mult-less-0-iff*

11.8 Division and Signs

lemma *zero-less-divide-iff*:
$((0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}) < a/b) = (0 < a \ \&\ 0 < b \mid a < 0 \ \&\ b < 0)$
by (*simp add*: *divide-inverse zero-less-mult-iff*)

lemma *divide-less-0-iff*:
$(a/b < (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})) =$
$(0 < a \ \&\ b < 0 \mid a < 0 \ \&\ 0 < b)$
by (*simp add*: *divide-inverse mult-less-0-iff*)

lemma *zero-le-divide-iff*:
$((0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\}) \leq a/b) =$
$(0 \leq a \ \&\ 0 \leq b \mid a \leq 0 \ \&\ b \leq 0)$
by (*simp add*: *divide-inverse zero-le-mult-iff*)

lemma *divide-le-0-iff*:
$(a/b \leq (0::'a::\{ordered\text{-}field, division\text{-}by\text{-}zero\})) =$
$(0 \leq a \ \&\ b \leq 0 \mid a \leq 0 \ \&\ 0 \leq b)$
by (*simp add*: *divide-inverse mult-le-0-iff*)

lemma *divide-eq-0-iff* [*simp*,*noatp*]:
(*a*/*b* = *0*) = (*a*=*0* | *b*=(*0*::'*a*::{*field*,*division-by-zero*}))
by (*simp add*: *divide-inverse*)

lemma *divide-pos-pos*:
0 < (*x*::'*a*::*ordered-field*) ==> *0* < *y* ==> *0* < *x* / *y*
by(*simp add*:*field-simps*)

lemma *divide-nonneg-pos*:
0 <= (*x*::'*a*::*ordered-field*) ==> *0* < *y* ==> *0* <= *x* / *y*
by(*simp add*:*field-simps*)

lemma *divide-neg-pos*:
(*x*::'*a*::*ordered-field*) < *0* ==> *0* < *y* ==> *x* / *y* < *0*
by(*simp add*:*field-simps*)

lemma *divide-nonpos-pos*:
(*x*::'*a*::*ordered-field*) <= *0* ==> *0* < *y* ==> *x* / *y* <= *0*
by(*simp add*:*field-simps*)

lemma *divide-pos-neg*:
0 < (*x*::'*a*::*ordered-field*) ==> *y* < *0* ==> *x* / *y* < *0*
by(*simp add*:*field-simps*)

lemma *divide-nonneg-neg*:
0 <= (*x*::'*a*::*ordered-field*) ==> *y* < *0* ==> *x* / *y* <= *0*
by(*simp add*:*field-simps*)

lemma *divide-neg-neg*:
(*x*::'*a*::*ordered-field*) < *0* ==> *y* < *0* ==> *0* < *x* / *y*
by(*simp add*:*field-simps*)

lemma *divide-nonpos-neg*:
(*x*::'*a*::*ordered-field*) <= *0* ==> *y* < *0* ==> *0* <= *x* / *y*
by(*simp add*:*field-simps*)

11.9 Cancellation Laws for Division

lemma *divide-cancel-right* [*simp*,*noatp*]:
(*a*/*c* = *b*/*c*) = (*c* = *0* | *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)
done

lemma *divide-cancel-left* [*simp*,*noatp*]:
(*c*/*a* = *c*/*b*) = (*c* = *0* | *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)

done

11.10 Division and the Number One

Simplify expressions equated with 1

lemma *divide-eq-1-iff* [*simp*,*noatp*]:
(*a*/*b* = *1*) = (*b* ≠ *0* & *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
apply (*cases b=0*, *simp*)
apply (*simp add*: *right-inverse-eq*)
done

lemma *one-eq-divide-iff* [*simp*,*noatp*]:
(*1* = *a*/*b*) = (*b* ≠ *0* & *a* = (*b*::'*a*::{*field*,*division-by-zero*}))
by (*simp add*: *eq-commute* [*of 1*])

lemma *zero-eq-1-divide-iff* [*simp*,*noatp*]:
((*0*::'*a*::{*ordered-field*,*division-by-zero*}) = *1*/*a*) = (*a* = *0*)
apply (*cases a=0*, *simp*)
apply (*auto simp add*: *nonzero-eq-divide-eq*)
done

lemma *one-divide-eq-0-iff* [*simp*,*noatp*]:
(*1*/*a* = (*0*::'*a*::{*ordered-field*,*division-by-zero*})) = (*a* = *0*)
apply (*cases a=0*, *simp*)
apply (*insert zero-neq-one* [*THEN not-sym*])
apply (*auto simp add*: *nonzero-divide-eq-eq*)
done

Simplify expressions such as *0* < *1*/*x* to *0* < *x*

lemmas *zero-less-divide-1-iff* = *zero-less-divide-iff* [*of 1*, *simplified*]
lemmas *divide-less-0-1-iff* = *divide-less-0-iff* [*of 1*, *simplified*]
lemmas *zero-le-divide-1-iff* = *zero-le-divide-iff* [*of 1*, *simplified*]
lemmas *divide-le-0-1-iff* = *divide-le-0-iff* [*of 1*, *simplified*]

declare *zero-less-divide-1-iff* [*simp*,*noatp*]
declare *divide-less-0-1-iff* [*simp*,*noatp*]
declare *zero-le-divide-1-iff* [*simp*,*noatp*]
declare *divide-le-0-1-iff* [*simp*,*noatp*]

11.11 Ordering Rules for Division

lemma *divide-strict-right-mono*:
[|*a* < *b*; *0* < *c*|] ==> *a* / *c* < *b* / (*c*::'*a*::*ordered-field*)
by (*simp add*: *order-less-imp-not-eq2 divide-inverse mult-strict-right-mono positive-imp-inverse-positive*)

lemma *divide-right-mono*:
[|*a* ≤ *b*; *0* ≤ *c*|] ==> *a*/*c* ≤ *b*/(*c*::'*a*::{*ordered-field*,*division-by-zero*})
by (*force simp add*: *divide-strict-right-mono order-le-less*)

lemma *divide-right-mono-neg*: (*a*::'*a*::{*division-by-zero*,*ordered-field*}) <= *b*
==> *c* <= *0* ==> *b* / *c* <= *a* / *c*
apply (*drule divide-right-mono* [*of* - - − *c*])
apply *auto*
done

lemma *divide-strict-right-mono-neg*:
[|*b* < *a*; *c* < *0*|] ==> *a* / *c* < *b* / (*c*::'*a*::*ordered-field*)
apply (*drule divide-strict-right-mono* [*of* - - −*c*], *simp*)
apply (*simp add*: *order-less-imp-not-eq nonzero-minus-divide-right* [*symmetric*])
done

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:
[|*b* < *a*; *0* < *c*; *0* < *a*∗*b*|] ==> *c* / *a* < *c* / (*b*::'*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono*)

lemma *divide-left-mono*:
[|*b* ≤ *a*; *0* ≤ *c*; *0* < *a*∗*b*|] ==> *c* / *a* ≤ *c* / (*b*::'*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-right-mono*)

lemma *divide-left-mono-neg*: (*a*::'*a*::{*division-by-zero*,*ordered-field*}) <= *b*
==> *c* <= *0* ==> *0* < *a* ∗ *b* ==> *c* / *a* <= *c* / *b*
apply (*drule divide-left-mono* [*of* - - − *c*])
apply (*auto simp add*: *mult-commute*)
done

lemma *divide-strict-left-mono-neg*:
[|*a* < *b*; *c* < *0*; *0* < *a*∗*b*|] ==> *c* / *a* < *c* / (*b*::'*a*::*ordered-field*)
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono-neg*)

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*1* ≤ *b* / *a*) = ((*0* < *a* & *a* ≤ *b*) | (*a* < *0* & *b* ≤ *a*))
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*b* / *a* ≤ *1*) = ((*0* < *a* & *b* ≤ *a*) | (*a* < *0* & *a* ≤ *b*) | *a*=*0*)
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1* [*noatp*]:
fixes *a* :: '*a* :: {*ordered-field*,*division-by-zero*}
shows (*1* < *b* / *a*) = ((*0* < *a* & *a* < *b*) | (*a* < *0* & *b* < *a*))
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1* [*noatp*]:

fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(b \ / \ a < 1) = ((0 < a \ \& \ b < a) \mid (a < 0 \ \& \ a < b) \mid a=0)$
by (*auto simp add*: *divide-less-eq*)

11.12 Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
by (*auto simp add*: *le-divide-eq*)

lemma *le-divide-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
by (*auto simp add*: *divide-le-eq*)

lemma *divide-le-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (1 < b/a) = (a < b)$
by (*auto simp add*: *less-divide-eq*)

lemma *less-divide-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow (1 < b/a) = (b < a)$
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1-pos* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $0 < a \Longrightarrow (b/a < 1) = (b < a)$
by (*auto simp add*: *divide-less-eq*)

lemma *divide-less-eq-1-neg* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $a < 0 \Longrightarrow b/a < 1 <-> a < b$
by (*auto simp add*: *divide-less-eq*)

lemma *eq-divide-eq-1* [*simp*,*noatp*]:
fixes $a :: 'a :: \{ordered\text{-}field, division\text{-}by\text{-}zero\}$
shows $(1 = b/a) = ((a \neq 0 \ \& \ a = b))$

by (*auto simp add*: *eq-divide-eq*)

lemma *divide-eq-eq-1* [*simp*,*noatp*]:
  fixes *a* :: *'a* :: {*ordered-field*,*division-by-zero*}
  shows (*b*/*a* = *1*) = ((*a* ≠ *0* & *a* = *b*))
by (*auto simp add*: *divide-eq-eq*)

11.13 Reasoning about inequalities with division

lemma *mult-right-le-one-le*: *0* <= (*x*::*'a*::*ordered-idom*) ==> *0* <= *y* ==> *y* <= *1*
    ==> *x* * *y* <= *x*
by (*auto simp add*: *mult-compare-simps*)

lemma *mult-left-le-one-le*: *0* <= (*x*::*'a*::*ordered-idom*) ==> *0* <= *y* ==> *y* <= *1*
    ==> *y* * *x* <= *x*
by (*auto simp add*: *mult-compare-simps*)

lemma *mult-imp-div-pos-le*: *0* < (*y*::*'a*::*ordered-field*) ==> *x* <= *z* * *y* ==>
    x / *y* <= *z*
by (*subst pos-divide-le-eq*, *assumption*+)

lemma *mult-imp-le-div-pos*: *0* < (*y*::*'a*::*ordered-field*) ==> *z* * *y* <= *x* ==>
    z <= *x* / *y*
by(*simp add*:*field-simps*)

lemma *mult-imp-div-pos-less*: *0* < (*y*::*'a*::*ordered-field*) ==> *x* < *z* * *y* ==>
    x / *y* < *z*
by(*simp add*:*field-simps*)

lemma *mult-imp-less-div-pos*: *0* < (*y*::*'a*::*ordered-field*) ==> *z* * *y* < *x* ==>
    z < *x* / *y*
by(*simp add*:*field-simps*)

lemma *frac-le*: (*0*::*'a*::*ordered-field*) <= *x* ==>
    x <= *y* ==> *0* < *w* ==> *w* <= *z* ==> *x* / *z* <= *y* / *w*
  apply (*rule mult-imp-div-pos-le*)
  apply *simp*
  apply (*subst times-divide-eq-left*)
  apply (*rule mult-imp-le-div-pos*, *assumption*)
  apply (*rule mult-mono*)
  apply *simp-all*
done

lemma *frac-less*: (*0*::*'a*::*ordered-field*) <= *x* ==>
    x < *y* ==> *0* < *w* ==> *w* <= *z* ==> *x* / *z* < *y* / *w*
  apply (*rule mult-imp-div-pos-less*)
  apply *simp*

apply (*subst times-divide-eq-left*)
apply (*rule mult-imp-less-div-pos*, *assumption*)
apply (*erule mult-less-le-imp-less*)
apply *simp-all*
done

lemma *frac-less2*: (*0*::*'a*::*ordered-field*) < *x* ==>
x <= *y* ==> *0* < *w* ==> *w* < *z* ==> *x* / *z* < *y* / *w*
apply (*rule mult-imp-div-pos-less*)
apply *simp-all*
apply (*subst times-divide-eq-left*)
apply (*rule mult-imp-less-div-pos*, *assumption*)
apply (*erule mult-le-less-imp-less*)
apply *simp-all*
done

It's not obvious whether these should be simprules or not. Their effect is to gather terms into one big fraction, like a*b*c / x*y*z. The rationale for that is unclear, but many proofs seem to need them.

declare *times-divide-eq* [*simp*]

11.14 Ordered Fields are Dense

context *ordered-semidom*
begin

lemma *less-add-one*: *a* < *a* + *1*
proof −
have *a* + *0* < *a* + *1*
by (*blast intro*: *zero-less-one add-strict-left-mono*)
thus *?thesis* **by** *simp*
qed

lemma *zero-less-two*: *0* < *1* + *1*
by (*blast intro*: *less-trans zero-less-one less-add-one*)

end

lemma *less-half-sum*: *a* < *b* ==> *a* < (*a*+*b*) / (*1*+*1*::*'a*::*ordered-field*)
by (*simp add*: *field-simps zero-less-two*)

lemma *gt-half-sum*: *a* < *b* ==> (*a*+*b*)/(*1*+*1*::*'a*::*ordered-field*) < *b*
by (*simp add*: *field-simps zero-less-two*)

instance *ordered-field* < *dense-linear-order*
proof
fix *x y* :: *'a*
have *x* < *x* + *1* **by** *simp*
then show ∃ *y*. *x* < *y* **..**

have $x - 1 < x$ **by** *simp*
then show $\exists y.\ y < x$ **..**
show $x < y \Longrightarrow \exists z{>}x.\ z < y$ **by** (*blast intro*!: *less-half-sum gt-half-sum*)
qed

11.15 Absolute Value

context *ordered-idom*
begin

lemma *mult-sgn-abs*: *sgn x * abs x = x*
unfolding *abs-if sgn-if* **by** *auto*

end

lemma *abs-one* [*simp*]: *abs 1 = (1::'a::ordered-idom)*
by (*simp add*: *abs-if zero-less-one* [*THEN order-less-not-sym*])

class *pordered-ring-abs = pordered-ring + pordered-ab-group-add-abs +*
assumes *abs-eq-mult*:
$(0 \le a \vee a \le 0) \wedge (0 \le b \vee b \le 0) \Longrightarrow |a * b| = |a| * |b|$

class *lordered-ring = pordered-ring + lordered-ab-group-add-abs*
begin

subclass *lordered-ab-group-add-meet* **..**
subclass *lordered-ab-group-add-join* **..**

end

lemma *abs-le-mult*: *abs (a * b) ≤ (abs a) * (abs (b::'a::lordered-ring))*
proof −
let *?x = pprt a * pprt b − pprt a * nprt b − nprt a * pprt b + nprt a * nprt b*
let *?y = pprt a * pprt b + pprt a * nprt b + nprt a * pprt b + nprt a * nprt b*
have *a*: *(abs a) * (abs b) = ?x*
by (*simp only*: *abs-prts*[*of a*] *abs-prts*[*of b*] *algebra-simps*)
{
fix *u v* :: *'a*
have *bh*: $[\![u = a;\ v = b]\!] \Longrightarrow$
*u * v = pprt a * pprt b + pprt a * nprt b +*
*nprt a * pprt b + nprt a * nprt b*
apply (*subst prts*[*of u*], *subst prts*[*of v*])
apply (*simp add*: *algebra-simps*)
done
}
note *b = this*[*OF refl*[*of a*] *refl*[*of b*]]
note *addm = add-mono*[*of 0::'a - 0::'a, simplified*]
note *addm2 = add-mono*[*of - 0::'a - 0::'a, simplified*]

```
  have xy: − ?x <= ?y
    apply (simp)
    apply (rule-tac y=0::'a in order-trans)
    apply (rule addm2)
    apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
    apply (rule addm)
    apply (simp-all add: mult-nonneg-nonneg mult-nonpos-nonpos)
    done
  have yx: ?y <= ?x
    apply (simp add:diff-def)
    apply (rule-tac y=0 in order-trans)
    apply (rule addm2, (simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)+)
    apply (rule addm, (simp add: mult-nonneg-nonpos mult-nonneg-nonpos2)+)
    done
  have i1: a*b <= abs a * abs b by (simp only: a b yx)
  have i2: − (abs a * abs b) <= a*b by (simp only: a b xy)
  show ?thesis
    apply (rule abs-leI)
    apply (simp add: i1)
    apply (simp add: i2[simplified minus-le-iff])
    done
qed

instance lordered-ring ⊆ pordered-ring-abs
proof
  fix a b :: 'a:: lordered-ring
  assume (0 ≤ a ∨ a ≤ 0) ∧ (0 ≤ b ∨ b ≤ 0)
  show abs (a*b) = abs a * abs b
proof −
  have s: (0 <= a*b) | (a*b <= 0)
    apply (auto)
    apply (rule-tac split-mult-pos-le)
    apply (rule-tac contrapos-np[of a*b <= 0])
    apply (simp)
    apply (rule-tac split-mult-neg-le)
    apply (insert prems)
    apply (blast)
    done
  have mulprts: a * b = (pprt a + nprt a) * (pprt b + nprt b)
    by (simp add: prts[symmetric])
  show ?thesis
  proof cases
    assume 0 <= a * b
    then show ?thesis
      apply (simp-all add: mulprts abs-prts)
      apply (insert prems)
      apply (auto simp add:
        algebra-simps
        iffD1[OF zero-le-iff-zero-nprt] iffD1[OF le-zero-iff-zero-pprt]
```

```
      iffD1[OF le-zero-iff-pprt-id] iffD1[OF zero-le-iff-nprt-id])
      apply(drule (1) mult-nonneg-nonpos[of a b], simp)
      apply(drule (1) mult-nonneg-nonpos2[of b a], simp)
     done
  next
    assume ~(0 <= a*b)
    with s have a*b <= 0 by simp
    then show ?thesis
     apply (simp-all add: mulprts abs-prts)
     apply (insert prems)
     apply (auto simp add: algebra-simps)
     apply(drule (1) mult-nonneg-nonneg[of a b],simp)
     apply(drule (1) mult-nonpos-nonpos[of a b],simp)
     done
  qed
qed
qed

instance ordered-idom ⊆ pordered-ring-abs
by default (auto simp add: abs-if not-less
  equal-neg-zero neg-equal-zero mult-less-0-iff)

lemma abs-mult: abs (a * b) = abs a * abs (b::'a::ordered-idom)
by (simp add: abs-eq-mult linorder-linear)

lemma abs-mult-self: abs a * abs a = a * (a::'a::ordered-idom)
by (simp add: abs-if)

lemma nonzero-abs-inverse:
    a ≠ 0 ==> abs (inverse (a::'a::ordered-field)) = inverse (abs a)
apply (auto simp add: linorder-neq-iff abs-if nonzero-inverse-minus-eq
                  negative-imp-inverse-negative)
apply (blast intro: positive-imp-inverse-positive elim: order-less-asym)
done

lemma abs-inverse [simp]:
    abs (inverse (a::'a::{ordered-field,division-by-zero})) =
     inverse (abs a)
apply (cases a=0, simp)
apply (simp add: nonzero-abs-inverse)
done

lemma nonzero-abs-divide:
    b ≠ 0 ==> abs (a / (b::'a::ordered-field)) = abs a / abs b
by (simp add: divide-inverse abs-mult nonzero-abs-inverse)

lemma abs-divide [simp]:
    abs (a / (b::'a::{ordered-field,division-by-zero})) = abs a / abs b
apply (cases b=0, simp)
```

apply (*simp add*: *nonzero-abs-divide*)
done

lemma *abs-mult-less*:
  [| *abs a* < *c*; *abs b* < *d* |] ==> *abs a* * *abs b* < *c*(*d*::'*a*::*ordered-idom*)
proof −
  assume *ac*: *abs a* < *c*
  hence *cpos*: *0*<*c* **by** (*blast intro*: *order-le-less-trans abs-ge-zero*)
  assume *abs b* < *d*
  thus *?thesis* **by** (*simp add*: *ac cpos mult-strict-mono*)
qed

lemmas *eq-minus-self-iff*[*noatp*] = *equal-neg-zero*

lemma *less-minus-self-iff*: (*a* < −*a*) = (*a* < (*0*::'*a*::*ordered-idom*))
  unfolding *order-less-le less-eq-neg-nonpos equal-neg-zero* **..**

lemma *abs-less-iff*: (*abs a* < *b*) = (*a* < *b* & −*a* < (*b*::'*a*::*ordered-idom*))
apply (*simp add*: *order-less-le abs-le-iff*)
apply (*auto simp add*: *abs-if neg-less-eq-nonneg less-eq-neg-nonpos*)
done

lemma *abs-mult-pos*: (*0*::'*a*::*ordered-idom*) <= *x* ==>
    (*abs y*) * *x* = *abs* (*y* * *x*)
  apply (*subst abs-mult*)
  apply *simp*
done

lemma *abs-div-pos*: (*0*::'*a*::{*division-by-zero*,*ordered-field*}) < *y* ==>
    abs x / *y* = *abs* (*x* / *y*)
  apply (*subst abs-divide*)
  apply (*simp add*: *order-less-imp-le*)
done

11.16 Bounds of products via negative and positive Part

lemma *mult-le-prts*:
  assumes
  a1 <= (*a*::'*a*::*lordered-ring*)
  a <= *a2*
  b1 <= *b*
  b <= *b2*
  shows
  a * *b* <= *pprt a2* * *pprt b2* + *pprt a1* * *nprt b2* + *nprt a2* * *pprt b1* + *nprt a1*
* *nprt b1*
proof −
  have *a* * *b* = (*pprt a* + *nprt a*) * (*pprt b* + *nprt b*)
    apply (*subst prts*[*symmetric*])+
    apply *simp*

```
    done
  then have a * b = pprt a * pprt b + pprt a * nprt b + nprt a * pprt b + nprt
a * nprt b
    by (simp add: algebra-simps)
  moreover have pprt a * pprt b <= pprt a2 * pprt b2
    by (simp-all add: prems mult-mono)
  moreover have pprt a * nprt b <= pprt a1 * nprt b2
  proof -
    have pprt a * nprt b <= pprt a * nprt b2
      by (simp add: mult-left-mono prems)
    moreover have pprt a * nprt b2 <= pprt a1 * nprt b2
      by (simp add: mult-right-mono-neg prems)
    ultimately show ?thesis
      by simp
  qed
  moreover have nprt a * pprt b <= nprt a2 * pprt b1
  proof -
    have nprt a * pprt b <= nprt a2 * pprt b
      by (simp add: mult-right-mono prems)
    moreover have nprt a2 * pprt b <= nprt a2 * pprt b1
      by (simp add: mult-left-mono-neg prems)
    ultimately show ?thesis
      by simp
  qed
  moreover have nprt a * nprt b <= nprt a1 * nprt b1
  proof -
    have nprt a * nprt b <= nprt a * nprt b1
      by (simp add: mult-left-mono-neg prems)
    moreover have nprt a * nprt b1 <= nprt a1 * nprt b1
      by (simp add: mult-right-mono-neg prems)
    ultimately show ?thesis
      by simp
  qed
  ultimately show ?thesis
    by - (rule add-mono | simp)+
qed

lemma mult-ge-prts:
  assumes
  a1 <= (a::'a::lordered-ring)
  a <= a2
  b1 <= b
  b <= b2
  shows
  a * b >= nprt a1 * pprt b2 + nprt a2 * nprt b2 + pprt a1 * pprt b1 + pprt a2
* nprt b1
proof -
  from prems have a1:- a2 <= -a by auto
  from prems have a2: -a <= -a1 by auto
```

from *mult-le-prts*[*of* $-a2$ $-a$ $-a1$ $b1$ b $b2$, *OF a1 a2 prems*(*3*) *prems*(*4*), *simplified nprt-neg pprt-neg*]
have *le*: $-(a * b) <= -\ nprt\ a1 * pprt\ b2 + -\ nprt\ a2 * nprt\ b2 + -\ pprt\ a1 * pprt\ b1 + -\ pprt\ a2 * nprt\ b1$ **by** *simp*
then have $-(-\ nprt\ a1 * pprt\ b2 + -\ nprt\ a2 * nprt\ b2 + -\ pprt\ a1 * pprt\ b1 + -\ pprt\ a2 * nprt\ b1) <= a * b$
by (*simp only*: *minus-le-iff*)
then show *?thesis* **by** *simp*
qed

end

12 Nat: Natural numbers

theory *Nat*
imports *Inductive Ring-and-Field*
uses
~~/*src*/*Tools*/*rat.ML*
~~/*src*/*Provers*/*Arith*/*cancel-sums.ML*
Tools/*arith-data.ML*
(*Tools*/*nat-arith.ML*)
~~/*src*/*Provers*/*Arith*/*fast-lin-arith.ML*
(*Tools*/*lin-arith.ML*)
begin

12.1 Type *ind*

typedecl *ind*

axiomatization
Zero-Rep :: *ind* **and**
Suc-Rep :: *ind* => *ind*
where
— the axiom of infinity in 2 parts
inj-Suc-Rep: *inj Suc-Rep* **and**
Suc-Rep-not-Zero-Rep: *Suc-Rep x* $\neq$ *Zero-Rep*

12.2 Type nat

Type definition

inductive *Nat* :: *ind* $\Rightarrow$ *bool*
where
Zero-RepI: *Nat Zero-Rep*
| *Suc-RepI*: *Nat i* $\Longrightarrow$ *Nat* (*Suc-Rep i*)

global

typedef (**open** *Nat*)
nat = *Nat*
by (*rule exI*, *unfold mem-def*, *rule Nat.Zero-RepI*)

constdefs
Suc :: *nat* => *nat*
Suc-def: *Suc* == (%*n*. *Abs-Nat* (*Suc-Rep* (*Rep-Nat n*)))

local

instantiation *nat* :: *zero*
begin

definition *Zero-nat-def* [*code del*]:
0 = *Abs-Nat Zero-Rep*

instance ..

end

lemma *Suc-not-Zero*: *Suc m* ≠ *0*
by (*simp add*: *Zero-nat-def Suc-def Abs-Nat-inject* [*unfolded mem-def*]
Rep-Nat [*unfolded mem-def*] *Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep* [*unfolded mem-def*])

lemma *Zero-not-Suc*: *0* ≠ *Suc m*
by (*rule not-sym*, *rule Suc-not-Zero not-sym*)

rep-datatype *0* :: *nat Suc*
apply (*unfold Zero-nat-def Suc-def*)
apply (*rule Rep-Nat-inverse* [*THEN subst*]) — types force good instantiation
apply (*erule Rep-Nat* [*unfolded mem-def*, *THEN Nat.induct*])
apply (*iprover elim*: *Abs-Nat-inverse* [*unfolded mem-def*, *THEN subst*])
apply (*simp-all add*: *Abs-Nat-inject* [*unfolded mem-def*] *Rep-Nat* [*unfolded mem-def*]
Suc-RepI Zero-RepI Suc-Rep-not-Zero-Rep [*unfolded mem-def*]
Suc-Rep-not-Zero-Rep [*unfolded mem-def*, *symmetric*]
inj-Suc-Rep [*THEN inj-eq*] *Rep-Nat-inject*)
done

lemma *nat-induct* [*case-names 0 Suc*, *induct type*: *nat*]:
— for backward compatibility – names of variables differ
fixes *n*
assumes *P 0*
and ⋀*n*. *P n* ⟹ *P* (*Suc n*)
shows *P n*
using *assms* **by** (*rule nat.induct*)

declare *nat.exhaust* [*case-names 0 Suc*, *cases type*: *nat*]

lemmas *nat-rec-0* = *nat.recs*(*1*)
  and *nat-rec-Suc* = *nat.recs*(*2*)

lemmas *nat-case-0* = *nat.cases*(*1*)
  and *nat-case-Suc* = *nat.cases*(*2*)

Injectiveness and distinctness lemmas

lemma *inj-Suc*[*simp*]: *inj-on Suc N*
  by (*simp add*: *inj-on-def*)

lemma *Suc-neq-Zero*: *Suc m* = *0* ⟹ *R*
by (*rule notE*, *rule Suc-not-Zero*)

lemma *Zero-neq-Suc*: *0* = *Suc m* ⟹ *R*
by (*rule Suc-neq-Zero*, *erule sym*)

lemma *Suc-inject*: *Suc x* = *Suc y* ⟹ *x* = *y*
by (*rule inj-Suc* [*THEN injD*])

lemma *n-not-Suc-n*: *n* ≠ *Suc n*
by (*induct n*) *simp-all*

lemma *Suc-n-not-n*: *Suc n* ≠ *n*
by (*rule not-sym*, *rule n-not-Suc-n*)

A special form of induction for reasoning about *m* < *n* and *m* − *n*

lemma *diff-induct*: (!!*x*. *P x 0*) ==> (!!*y*. *P 0* (*Suc y*)) ==>
    (!!*x y*. *P x y* ==> *P* (*Suc x*) (*Suc y*)) ==> *P m n*
  apply (*rule-tac x* = *m* **in** *spec*)
  apply (*induct n*)
  prefer *2*
  apply (*rule allI*)
  apply (*induct-tac x*, *iprover*+)
  done

12.3 Arithmetic operators

instantiation *nat* :: {*minus*, *comm-monoid-add*}
begin

primrec *plus-nat*
where
  add-0: *0* + *n* = (*n*::*nat*)
  | *add-Suc*: *Suc m* + *n* = *Suc* (*m* + *n*)

lemma *add-0-right* [*simp*]: *m* + *0* = (*m*::*nat*)
  by (*induct m*) *simp-all*

lemma *add-Suc-right* [*simp*]: $m + Suc\ n = Suc\ (m + n)$
by (*induct m*) *simp-all*

declare *add-0* [*code*]

lemma *add-Suc-shift* [*code*]: $Suc\ m + n = m + Suc\ n$
by *simp*

primrec *minus-nat*
where
diff-0: $m - 0 = (m::nat)$
| *diff-Suc*: $m - Suc\ n = (case\ m - n\ of\ 0 => 0 \mid Suc\ k => k)$

declare *diff-Suc* [*simp del*]
declare *diff-0* [*code*]

lemma *diff-0-eq-0* [*simp*, *code*]: $0 - n = (0::nat)$
by (*induct n*) (*simp-all add*: *diff-Suc*)

lemma *diff-Suc-Suc* [*simp*, *code*]: $Suc\ m - Suc\ n = m - n$
by (*induct n*) (*simp-all add*: *diff-Suc*)

instance proof
fix $n\ m\ q :: nat$
show $(n + m) + q = n + (m + q)$ **by** (*induct n*) *simp-all*
show $n + m = m + n$ **by** (*induct n*) *simp-all*
show $0 + n = n$ **by** *simp*
qed

end

instantiation *nat* :: *comm-semiring-1-cancel*
begin

definition
One-nat-def [*simp*]: $1 = Suc\ 0$

primrec *times-nat*
where
mult-0: $0 * n = (0::nat)$
| *mult-Suc*: $Suc\ m * n = n + (m * n)$

lemma *mult-0-right* [*simp*]: $(m::nat) * 0 = 0$
by (*induct m*) *simp-all*

lemma *mult-Suc-right* [*simp*]: $m * Suc\ n = m + (m * n)$
by (*induct m*) (*simp-all add*: *add-left-commute*)

lemma *add-mult-distrib*: $(m + n) * k = (m * k) + ((n * k)::nat)$

by (*induct m*) (*simp-all add*: *add-assoc*)

instance proof
fix *n m q* :: *nat*
show *0* ≠ (*1*::*nat*) **unfolding** *One-nat-def* **by** *simp*
show *1* ∗ *n* = *n* **unfolding** *One-nat-def* **by** *simp*
show *n* ∗ *m* = *m* ∗ *n* **by** (*induct n*) *simp-all*
show (*n* ∗ *m*) ∗ *q* = *n* ∗ (*m* ∗ *q*) **by** (*induct n*) (*simp-all add*: *add-mult-distrib*)
show (*n* + *m*) ∗ *q* = *n* ∗ *q* + *m* ∗ *q* **by** (*rule add-mult-distrib*)
assume *n* + *m* = *n* + *q* **thus** *m* = *q* **by** (*induct n*) *simp-all*
qed

end

12.3.1 Addition

lemma *nat-add-assoc*: (*m* + *n*) + *k* = *m* + ((*n* + *k*)::*nat*)
by (*rule add-assoc*)

lemma *nat-add-commute*: *m* + *n* = *n* + (*m*::*nat*)
by (*rule add-commute*)

lemma *nat-add-left-commute*: *x* + (*y* + *z*) = *y* + ((*x* + *z*)::*nat*)
by (*rule add-left-commute*)

lemma *nat-add-left-cancel* [*simp*]: (*k* + *m* = *k* + *n*) = (*m* = (*n*::*nat*))
by (*rule add-left-cancel*)

lemma *nat-add-right-cancel* [*simp*]: (*m* + *k* = *n* + *k*) = (*m*=(*n*::*nat*))
by (*rule add-right-cancel*)

Reasoning about *m* + *0* = *0*, etc.

lemma *add-is-0* [*iff*]:
fixes *m n* :: *nat*
shows (*m* + *n* = *0*) = (*m* = *0* & *n* = *0*)
by (*cases m*) *simp-all*

lemma *add-is-1*:
(*m*+*n*= *Suc 0*) = (*m*= *Suc 0* & *n*=*0* | *m*=*0* & *n*= *Suc 0*)
by (*cases m*) *simp-all*

lemma *one-is-add*:
(*Suc 0* = *m* + *n*) = (*m* = *Suc 0* & *n* = *0* | *m* = *0* & *n* = *Suc 0*)
by (*rule trans*, *rule eq-commute*, *rule add-is-1*)

lemma *add-eq-self-zero*:
fixes *m n* :: *nat*
shows *m* + *n* = *m* ⟹ *n* = *0*
by (*induct m*) *simp-all*

lemma *inj-on-add-nat*[*simp*]: *inj-on* (%*n*::*nat*. *n*+*k*) *N*
apply (*induct k*)
apply *simp*
apply(*drule comp-inj-on*[*OF - inj-Suc*])
apply (*simp add*:*o-def*)
done

12.3.2 Difference

lemma *diff-self-eq-0* [*simp*]: (*m*::*nat*) − *m* = *0*
by (*induct m*) *simp-all*

lemma *diff-diff-left*: (*i*::*nat*) − *j* − *k* = *i* − (*j* + *k*)
by (*induct i j rule*: *diff-induct*) *simp-all*

lemma *Suc-diff-diff* [*simp*]: (*Suc m* − *n*) − *Suc k* = *m* − *n* − *k*
by (*simp add*: *diff-diff-left*)

lemma *diff-commute*: (*i*::*nat*) − *j* − *k* = *i* − *k* − *j*
by (*simp add*: *diff-diff-left add-commute*)

lemma *diff-add-inverse*: (*n* + *m*) − *n* = (*m*::*nat*)
by (*induct n*) *simp-all*

lemma *diff-add-inverse2*: (*m* + *n*) − *n* = (*m*::*nat*)
by (*simp add*: *diff-add-inverse add-commute* [*of m n*])

lemma *diff-cancel*: (*k* + *m*) − (*k* + *n*) = *m* − (*n*::*nat*)
by (*induct k*) *simp-all*

lemma *diff-cancel2*: (*m* + *k*) − (*n* + *k*) = *m* − (*n*::*nat*)
by (*simp add*: *diff-cancel add-commute*)

lemma *diff-add-0*: *n* − (*n* + *m*) = (*0*::*nat*)
by (*induct n*) *simp-all*

lemma *diff-Suc-1* [*simp*]: *Suc n* − *1* = *n*
unfolding *One-nat-def* **by** *simp*

Difference distributes over multiplication

lemma *diff-mult-distrib*: ((*m*::*nat*) − *n*) ∗ *k* = (*m* ∗ *k*) − (*n* ∗ *k*)
by (*induct m n rule*: *diff-induct*) (*simp-all add*: *diff-cancel*)

lemma *diff-mult-distrib2*: *k* ∗ ((*m*::*nat*) − *n*) = (*k* ∗ *m*) − (*k* ∗ *n*)
by (*simp add*: *diff-mult-distrib mult-commute* [*of k*])
— NOT added as rewrites, since sometimes they are used from right-to-left

12.3.3 Multiplication

lemma *nat-mult-assoc*: $(m * n) * k = m * ((n * k)::nat)$
by (*rule mult-assoc*)

lemma *nat-mult-commute*: $m * n = n * (m::nat)$
by (*rule mult-commute*)

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + ((k * n)::nat)$
by (*rule right-distrib*)

lemma *mult-is-0* [*simp*]: $((m::nat) * n = 0) = (m=0 \mid n=0)$
by (*induct m*) *auto*

lemmas *nat-distrib* =
add-mult-distrib add-mult-distrib2 diff-mult-distrib diff-mult-distrib2

lemma *mult-eq-1-iff* [*simp*]: $(m * n = Suc\ 0) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
apply (*induct m*)
apply *simp*
apply (*induct n*)
apply *auto*
done

lemma *one-eq-mult-iff* [*simp*,*noatp*]: $(Suc\ 0 = m * n) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
apply (*rule trans*)
apply (*rule-tac* [*2*] *mult-eq-1-iff*, *fastsimp*)
done

lemma *nat-mult-eq-1-iff* [*simp*]: $m * n = (1::nat) \longleftrightarrow m = 1 \wedge n = 1$
unfolding *One-nat-def* **by** (*rule mult-eq-1-iff*)

lemma *nat-1-eq-mult-iff* [*simp*]: $(1::nat) = m * n \longleftrightarrow m = 1 \wedge n = 1$
unfolding *One-nat-def* **by** (*rule one-eq-mult-iff*)

lemma *mult-cancel1* [*simp*]: $(k * m = k * n) = (m = n \mid (k = (0::nat)))$
proof −
have $k \neq 0 \Longrightarrow k * m = k * n \Longrightarrow m = n$
proof (*induct n arbitrary*: *m*)
case *0* **then show** $m = 0$ **by** *simp*
next
case (*Suc n*) **then show** $m = Suc\ n$
by (*cases m*) (*simp-all add*: *eq-commute* [*of 0*])
qed
then show *?thesis* **by** *auto*
qed

lemma *mult-cancel2* [*simp*]: $(m * k = n * k) = (m = n \mid (k = (0::nat)))$
by (*simp add*: *mult-commute*)

lemma *Suc-mult-cancel1*: $(Suc\ k * m = Suc\ k * n) = (m = n)$
by (*subst mult-cancel1*) *simp*

12.4 Orders on *nat*

12.4.1 Operation definition

instantiation *nat* :: *linorder*
begin

primrec *less-eq-nat* **where**
$(0::nat) \leq n \longleftrightarrow True$
| $Suc\ m \leq n \longleftrightarrow (case\ n\ of\ 0 \Rightarrow False \mid Suc\ n \Rightarrow m \leq n)$

declare *less-eq-nat.simps* [*simp del*]
lemma [*code*]: $(0::nat) \leq n \longleftrightarrow True$ **by** (*simp add*: *less-eq-nat.simps*)
lemma *le0* [*iff*]: $0 \leq (n::nat)$ **by** (*simp add*: *less-eq-nat.simps*)

definition *less-nat* **where**
less-eq-Suc-le: $n < m \longleftrightarrow Suc\ n \leq m$

lemma *Suc-le-mono* [*iff*]: $Suc\ n \leq Suc\ m \longleftrightarrow n \leq m$
by (*simp add*: *less-eq-nat.simps*(*2*))

lemma *Suc-le-eq* [*code*]: $Suc\ m \leq n \longleftrightarrow m < n$
unfolding *less-eq-Suc-le* **..**

lemma *le-0-eq* [*iff*]: $(n::nat) \leq 0 \longleftrightarrow n = 0$
by (*induct n*) (*simp-all add*: *less-eq-nat.simps*(*2*))

lemma *not-less0* [*iff*]: $\neg\ n < (0::nat)$
by (*simp add*: *less-eq-Suc-le*)

lemma *less-nat-zero-code* [*code*]: $n < (0::nat) \longleftrightarrow False$
by *simp*

lemma *Suc-less-eq* [*iff*]: $Suc\ m < Suc\ n \longleftrightarrow m < n$
by (*simp add*: *less-eq-Suc-le*)

lemma *less-Suc-eq-le* [*code*]: $m < Suc\ n \longleftrightarrow m \leq n$
by (*simp add*: *less-eq-Suc-le*)

lemma *le-SucI*: $m \leq n \Longrightarrow m \leq Suc\ n$
by (*induct m arbitrary*: *n*)
(*simp-all add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)

lemma *Suc-leD*: $Suc\ m \leq n \Longrightarrow m \leq n$
by (*cases n*) (*auto intro*: *le-SucI*)

lemma *less-SucI*: $m < n \Longrightarrow m < Suc\ n$
by (*simp add*: *less-eq-Suc-le*) (*erule Suc-leD*)

lemma *Suc-lessD*: $Suc\ m < n \Longrightarrow m < n$
by (*simp add*: *less-eq-Suc-le*) (*erule Suc-leD*)

instance
proof
fix $n\ m$ :: *nat*
show $n < m \longleftrightarrow n \leq m \wedge \neg\ m \leq n$
proof (*induct n arbitrary*: m)
case *0* **then show** *?case* **by** (*cases m*) (*simp-all add*: *less-eq-Suc-le*)
next
case (*Suc n*) **then show** *?case* **by** (*cases m*) (*simp-all add*: *less-eq-Suc-le*)
qed
next
fix n :: *nat* **show** $n \leq n$ **by** (*induct n*) *simp-all*
next
fix $n\ m$ :: *nat* **assume** $n \leq m$ **and** $m \leq n$
then show $n = m$
by (*induct n arbitrary*: m)
(*simp-all add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)
next
fix $n\ m\ q$ :: *nat* **assume** $n \leq m$ **and** $m \leq q$
then show $n \leq q$
proof (*induct n arbitrary*: $m\ q$)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **then show** *?case*
by (*simp-all* (*no-asm-use*) *add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*, *clarify*,
simp-all (*no-asm-use*) *add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*, *clarify*,
simp-all (*no-asm-use*) *add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)
qed
next
fix $n\ m$ :: *nat* **show** $n \leq m \vee m \leq n$
by (*induct n arbitrary*: m)
(*simp-all add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)
qed

end

instantiation *nat* :: *bot*
begin

definition *bot-nat* :: *nat* **where**
bot-nat = *0*

instance proof
qed (*simp add*: *bot-nat-def*)

end

12.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < Suc\ n$
by (*simp add*: *less-Suc-eq-le*)

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
by (*simp add*: *less-Suc-eq-le*)

12.4.3 Elimination properties

lemma *less-not-refl*: $\sim n < (n::nat)$
by (*rule order-less-irrefl*)

lemma *less-not-refl2*: $n < m \Longrightarrow m \neq (n::nat)$
by (*rule not-sym*) (*rule less-imp-neq*)

lemma *less-not-refl3*: $(s::nat) < t \Longrightarrow s \neq t$
by (*rule less-imp-neq*)

lemma *less-irrefl-nat*: $(n::nat) < n \Longrightarrow R$
by (*rule notE*, *rule less-not-refl*)

lemma *less-zeroE*: $(n::nat) < 0 \Longrightarrow R$
by (*rule notE*) (*rule not-less0*)

lemma *less-Suc-eq*: $(m < Suc\ n) = (m < n \mid m = n)$
unfolding *less-Suc-eq-le le-less* **..**

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$
by (*simp add*: *less-Suc-eq*)

lemma *less-one* [*iff*, *noatp*]: $(n < (1::nat)) = (n = 0)$
unfolding *One-nat-def* **by** (*rule less-Suc0*)

lemma *Suc-mono*: $m < n \Longrightarrow Suc\ m < Suc\ n$
by *simp*

"Less than" is antisymmetric, sort of

lemma *less-antisym*: $[\![\neg n < m;\ n < Suc\ m]\!] \Longrightarrow m = n$
unfolding *not-less less-Suc-eq-le* **by** (*rule antisym*)

lemma *nat-neq-iff*: $((m::nat) \neq n) = (m < n \mid n < m)$
by (*rule linorder-neq-iff*)

lemma *nat-less-cases*: **assumes** *major*: $(m::nat) < n \Longrightarrow P\ n\ m$
and *eqCase*: $m = n \Longrightarrow P\ n\ m$ **and** *lessCase*: $n<m \Longrightarrow P\ n\ m$
shows $P\ n\ m$

apply (*rule less-linear* [*THEN disjE*])
apply (*erule-tac* [*2*] *disjE*)
apply (*erule lessCase*)
apply (*erule sym* [*THEN eqCase*])
apply (*erule major*)
done

12.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \Longrightarrow Suc\ m \neq n \Longrightarrow Suc\ m < n$
unfolding *less-eq-Suc-le* [*of m*] *le-less* **by** *simp*

lemma *lessE*:
assumes *major*: $i < k$
and *p1*: $k = Suc\ i \Longrightarrow P$ **and** *p2*: $!!j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$
shows P
proof −
from *major* **have** $\exists j.\ i \leq j \wedge k = Suc\ j$
unfolding *less-eq-Suc-le* **by** (*induct k*) *simp-all*
then have $(\exists j.\ i < j \wedge k = Suc\ j) \vee k = Suc\ i$
by (*clarsimp simp add*: *less-le*)
with *p1 p2* **show** P **by** *auto*
qed

lemma *less-SucE*: **assumes** *major*: $m < Suc\ n$
and *less*: $m < n \Longrightarrow P$ **and** *eq*: $m = n \Longrightarrow P$ **shows** P
apply (*rule major* [*THEN lessE*])
apply (*rule eq*, *blast*)
apply (*rule less*, *blast*)
done

lemma *Suc-lessE*: **assumes** *major*: $Suc\ i < k$
and *minor*: $!!j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$ **shows** P
apply (*rule major* [*THEN lessE*])
apply (*erule lessI* [*THEN minor*])
apply (*erule Suc-lessD* [*THEN minor*], *assumption*)
done

lemma *Suc-less-SucD*: $Suc\ m < Suc\ n \Longrightarrow m < n$
by *simp*

lemma *less-trans-Suc*:
assumes *le*: $i < j$ **shows** $j < k \Longrightarrow Suc\ i < k$
apply (*induct k*, *simp-all*)
apply (*insert le*)
apply (*simp add*: *less-Suc-eq*)
apply (*blast dest*: *Suc-lessD*)
done

Can be used with *less-Suc-eq* to get $n = m \vee n < m$

lemma *not-less-eq*: $\neg\ m < n \longleftrightarrow n < Suc\ m$
unfolding *not-less less-Suc-eq-le* **..**

lemma *not-less-eq-eq*: $\neg\ m \leq n \longleftrightarrow Suc\ n \leq m$
unfolding *not-le Suc-le-eq* **..**

Properties of "less than or equal"

lemma *le-imp-less-Suc*: $m \leq n \Longrightarrow m < Suc\ n$
unfolding *less-Suc-eq-le* **.**

lemma *Suc-n-not-le-n*: $\sim\ Suc\ n \leq n$
unfolding *not-le less-Suc-eq-le* **..**

lemma *le-Suc-eq*: $(m \leq Suc\ n) = (m \leq n \mid m = Suc\ n)$
by (*simp add*: *less-Suc-eq-le* [*symmetric*] *less-Suc-eq*)

lemma *le-SucE*: $m \leq Suc\ n \Longrightarrow (m \leq n \Longrightarrow R) \Longrightarrow (m = Suc\ n \Longrightarrow R) \Longrightarrow R$
by (*drule le-Suc-eq* [*THEN iffD1*], *iprover*+)

lemma *Suc-leI*: $m < n \Longrightarrow Suc(m) \leq n$
unfolding *Suc-le-eq* **.**

Stronger version of *Suc-leD*

lemma *Suc-le-lessD*: $Suc\ m \leq n \Longrightarrow m < n$
unfolding *Suc-le-eq* **.**

lemma *less-imp-le-nat*: $m < n \Longrightarrow m \leq (n::nat)$
unfolding *less-eq-Suc-le* **by** (*rule Suc-leD*)

For instance, $(Suc\ m < Suc\ n) = (Suc\ m \leq n) = (m < n)$

lemmas *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *less-or-eq-imp-le*: $m < n \mid m = n \Longrightarrow m \leq (n::nat)$
unfolding *le-less* **.**

lemma *le-eq-less-or-eq*: $(m \leq (n::nat)) = (m < n \mid m=n)$
by (*rule le-less*)

Useful with *blast*.

lemma *eq-imp-le*: $(m::nat) = n \Longrightarrow m \leq n$
by *auto*

lemma *le-refl*: $n \leq (n::nat)$
by *simp*

lemma *le-trans*: $[|\ i \leq j;\ j \leq k\ |] \Longrightarrow i \leq (k::nat)$

by (*rule order-trans*)

lemma *le-anti-sym*: [| $m \leq n$; $n \leq m$ |] ==> $m = (n::nat)$
by (*rule antisym*)

lemma *nat-less-le*: $((m::nat) < n) = (m \leq n \;\&\; m \neq n)$
by (*rule less-le*)

lemma *le-neq-implies-less*: $(m::nat) \leq n$ ==> $m \neq n$ ==> $m < n$
unfolding *less-le* **..**

lemma *nat-le-linear*: $(m::nat) \leq n \mid n \leq m$
by (*rule linear*)

lemmas *linorder-neqE-nat* = *linorder-neqE* [**where** $'a = nat$]

lemma *le-less-Suc-eq*: $m \leq n$ ==> $(n < Suc\ m) = (n = m)$
unfolding *less-Suc-eq-le* **by** *auto*

lemma *not-less-less-Suc-eq*: ~ $n < m$ ==> $(n < Suc\ m) = (n = m)$
unfolding *not-less* **by** (*rule le-less-Suc-eq*)

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

These two rules ease the use of primitive recursion. NOTE USE OF ==

lemma *def-nat-rec-0*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n)$ ==> $f\ 0 = c$
by *simp*

lemma *def-nat-rec-Suc*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n)$ ==> $f\ (Suc\ n) = h\ n\ (f\ n)$
by *simp*

lemma *not0-implies-Suc*: $n \neq 0$ ==> $\exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr0-implies-Suc*: $n > 0$ ==> $\exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr-implies-not0*: **fixes** $n :: nat$ **shows** $m<n$ ==> $n \neq 0$
by (*cases n*) *simp-all*

lemma *neq0-conv*[*iff*]: **fixes** $n :: nat$ **shows** $(n \neq 0) = (0 < n)$
by (*cases n*) *simp-all*

This theorem is useful with *blast*

lemma *gr0I*: $((n::nat) = 0$ ==> $False)$ ==> $0 < n$
by (*rule neq0-conv*[*THEN iffD1*], *iprover*)

lemma *gr0-conv-Suc*: $(0 < n) = (\exists\, m.\ n = Suc\ m)$
by (*fast intro*: *not0-implies-Suc*)

lemma *not-gr0* [*iff*,*noatp*]: !!*n*::*nat*. (~ (*0* < *n*)) = (*n* = *0*)
using *neq0-conv* **by** *blast*

lemma *Suc-le-D*: (*Suc n* ≤ *m'*) ==> (*? m*. *m'* = *Suc m*)
by (*induct m'*) *simp-all*

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: (*m* < *Suc n*) = (*m* = *0* | (∃ *j*. *m* = *Suc j* & *j* < *n*))
by (*cases m*) *simp-all*

12.4.5 *min* and *max*

lemma *mono-Suc*: *mono Suc*
by (*rule monoI*) *simp*

lemma *min-0L* [*simp*]: *min 0 n* = (*0*::*nat*)
by (*rule min-leastL*) *simp*

lemma *min-0R* [*simp*]: *min n 0* = (*0*::*nat*)
by (*rule min-leastR*) *simp*

lemma *min-Suc-Suc* [*simp*]: *min* (*Suc m*) (*Suc n*) = *Suc* (*min m n*)
by (*simp add*: *mono-Suc min-of-mono*)

lemma *min-Suc1*:
min (*Suc n*) *m* = (*case m of 0* => *0* | *Suc m'* => *Suc*(*min n m'*))
by (*simp split*: *nat.split*)

lemma *min-Suc2*:
min m (*Suc n*) = (*case m of 0* => *0* | *Suc m'* => *Suc*(*min m' n*))
by (*simp split*: *nat.split*)

lemma *max-0L* [*simp*]: *max 0 n* = (*n*::*nat*)
by (*rule max-leastL*) *simp*

lemma *max-0R* [*simp*]: *max n 0* = (*n*::*nat*)
by (*rule max-leastR*) *simp*

lemma *max-Suc-Suc* [*simp*]: *max* (*Suc m*) (*Suc n*) = *Suc*(*max m n*)
by (*simp add*: *mono-Suc max-of-mono*)

lemma *max-Suc1*:
max (*Suc n*) *m* = (*case m of 0* => *Suc n* | *Suc m'* => *Suc*(*max n m'*))
by (*simp split*: *nat.split*)

lemma *max-Suc2*:
max m (*Suc n*) = (*case m of 0* => *Suc n* | *Suc m'* => *Suc*(*max m' n*))
by (*simp split*: *nat.split*)

12.4.6 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: *n>0 ==> Suc (n − Suc 0) = n*
by (*simp add*: *diff-Suc split*: *nat.split*)

lemma *Suc-diff-1* [*simp*]: *0 < n ==> Suc (n − 1) = n*
unfolding *One-nat-def* **by** (*rule Suc-pred*)

lemma *nat-add-left-cancel-le* [*simp*]: *(k + m ≤ k + n) = (m≤(n::nat))*
by (*induct k*) *simp-all*

lemma *nat-add-left-cancel-less* [*simp*]: *(k + m < k + n) = (m<(n::nat))*
by (*induct k*) *simp-all*

lemma *add-gr-0* [*iff*]: *!!m::nat. (m + n > 0) = (m>0 | n>0)*
by(*auto dest*:*gr0-implies-Suc*)

strict, in 1st argument

lemma *add-less-mono1*: *i < j ==> i + k < j + (k::nat)*
by (*induct k*) *simp-all*

strict, in both arguments

lemma *add-less-mono*: *[|i < j; k < l|] ==> i + k < j + (l::nat)*
  apply (*rule add-less-mono1* [*THEN less-trans*], *assumption*+)
  apply (*induct j*, *simp-all*)
  done

Deleted *less-natE*; use *less-imp-Suc-add RS exE*

lemma *less-imp-Suc-add*: *m < n ==> (∃ k. n = Suc (m + k))*
  apply (*induct n*)
  apply (*simp-all add*: *order-le-less*)
  apply (*blast elim*!: *less-SucE*
        intro!: *add-0-right* [*symmetric*] *add-Suc-right* [*symmetric*])
  done

strict, in 1st argument; proof is by induction on *k > 0*

lemma *mult-less-mono2*: *(i::nat) < j ==> 0<k ==> k * i < k * j*
apply(*auto simp*: *gr0-conv-Suc*)
apply (*induct-tac m*)
apply (*simp-all add*: *add-less-mono*)
done

The naturals form an ordered *comm-semiring-1-cancel*

instance *nat* :: *ordered-semidom*
proof
  fix *i j k* :: *nat*
  show *0 < (1::nat)* **by** *simp*
  show *i ≤ j ==> k + i ≤ k + j* **by** *simp*
  show *i < j ==> 0 < k ==> k * i < k * j* **by** (*simp add*: *mult-less-mono2*)

qed

instance *nat* :: *no-zero-divisors*
proof
  fix *a*::*nat* **and** *b*::*nat* **show** $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$ **by** *auto*
qed

lemma *nat-mult-1*: $(1::nat) * n = n$
by *simp*

lemma *nat-mult-1-right*: $n * (1::nat) = n$
by *simp*

12.4.7 Additional theorems about *op* ≤

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder* **proof**
  fix *P* **and** *n* :: *nat*
  assume *step*: $\bigwedge n::nat.\ (\bigwedge m.\ m < n \Longrightarrow P\ m) \Longrightarrow P\ n$
  have $\bigwedge q.\ q \leq n \Longrightarrow P\ q$
  proof (*induct n*)
    case (*0 n*)
    have *P 0* **by** (*rule step*) *auto*
    thus *?case* **using** *0* **by** *auto*
  next
    case (*Suc m n*)
    then have $n \leq m \vee n = Suc\ m$ **by** (*simp add*: *le-Suc-eq*)
    thus *?case*
    proof
      assume $n \leq m$ **thus** *P n* **by** (*rule Suc*(*1*))
    next
      assume *n*: $n = Suc\ m$
      show *P n*
        by (*rule step*) (*rule Suc*(*1*), *simp add*: *n le-simps*)
    qed
  qed
  then show *P n* **by** *auto*
qed

lemma *Least-Suc*:
    [| *P n*; ~ *P 0* |] ==> (*LEAST n. P n*) = *Suc* (*LEAST m. P*(*Suc m*))
  apply (*case-tac n*, *auto*)
  apply (*frule LeastI*)
  apply (*drule-tac P* = %*x. P* (*Suc x*) **in** *LeastI*)
  apply (*subgoal-tac* (*LEAST x. P x*) ≤ *Suc* (*LEAST x. P* (*Suc x*)))
  apply (*erule-tac* [*2*] *Least-le*)
  apply (*case-tac LEAST x. P x*, *auto*)
  apply (*drule-tac P* = %*x. P* (*Suc x*) **in** *Least-le*)
  apply (*blast intro*: *order-antisym*)

done

lemma *Least-Suc2*:
[|*P n*; *Q m*; ~*P 0*; !*k*. *P* (*Suc k*) = *Q k*|] ==> *Least P* = *Suc* (*Least Q*)
apply (*erule* (*1*) *Least-Suc* [*THEN ssubst*])
apply *simp*
done

lemma *ex-least-nat-le*: ¬*P*(*0*) ⟹ *P*(*n*::*nat*) ⟹ ∃*k*≤*n*. (∀*i*<*k*. ¬*P i*) & *P*(*k*)
apply (*cases n*)
apply *blast*
apply (*rule-tac x*=*LEAST k*. *P*(*k*) **in** *exI*)
apply (*blast intro*: *Least-le dest*: *not-less-Least intro*: *LeastI-ex*)
done

lemma *ex-least-nat-less*: ¬*P*(*0*) ⟹ *P*(*n*::*nat*) ⟹ ∃*k*<*n*. (∀*i*≤*k*. ¬*P i*) & *P*(*k*+*1*)
unfolding *One-nat-def*
apply (*cases n*)
apply *blast*
apply (*frule* (*1*) *ex-least-nat-le*)
apply (*erule exE*)
apply (*case-tac k*)
apply *simp*
apply (*rename-tac k1*)
apply (*rule-tac x*=*k1* **in** *exI*)
apply (*auto simp add*: *less-eq-Suc-le*)
done

lemma *nat-less-induct*:
assumes !!*n*. ∀*m*::*nat*. *m* < *n* --> *P m* ==> *P n* **shows** *P n*
using *assms less-induct* **by** *blast*

lemma *measure-induct-rule* [*case-names less*]:
fixes *f* :: *'a* ⇒ *nat*
assumes *step*: ⋀*x*. (⋀*y*. *f y* < *f x* ⟹ *P y*) ⟹ *P x*
shows *P a*
by (*induct m*≡*f a arbitrary*: *a rule*: *less-induct*) (*auto intro*: *step*)

old style induction rules:

lemma *measure-induct*:
fixes *f* :: *'a* ⇒ *nat*
shows (⋀*x*. ∀*y*. *f y* < *f x* ⟶ *P y* ⟹ *P x*) ⟹ *P a*
by (*rule measure-induct-rule* [*of f P a*]) *iprover*

lemma *full-nat-induct*:
assumes *step*: (!!*n*. (*ALL m*. *Suc m* <= *n* --> *P m*) ==> *P n*)
shows *P n*
by (*rule less-induct*) (*auto intro*: *step simp*:*le-simps*)

An induction rule for estabilishing binary relations

lemma *less-Suc-induct*:
assumes *less*: $i < j$
and *step*: $!!i.\ P\ i\ (Suc\ i)$
and *trans*: $!!i\ j\ k.\ P\ i\ j \Longrightarrow P\ j\ k \Longrightarrow P\ i\ k$
shows $P\ i\ j$
proof −
from *less* **obtain** k **where** j: $j = Suc(i+k)$ **by** (*auto dest*: *less-imp-Suc-add*)
have $P\ i\ (Suc\ (i + k))$
proof (*induct k*)
case *0*
show *?case* **by** (*simp add*: *step*)
next
case (*Suc k*)
thus *?case* **by** (*auto intro*: *assms*)
qed
thus $P\ i\ j$ **by** (*simp add*: *j*)
qed

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. $P(n)$ is true for all $n \in \mathbb{N}$ if

- case "0": given $n = 0$ prove $P(n)$,
- case "smaller": given $n > 0$ and $\neg P(n)$ prove there exists a smaller integer m such that $\neg P(m)$.

A compact version without explicit base case:

lemma *infinite-descent*:
$[\![\ !!n::nat.\ \neg\ P\ n \Longrightarrow\ \exists\, m<n.\ \neg\ \ P\ m\ ]\!] \Longrightarrow\ \ P\ n$
by (*induct n rule*: *less-induct*, *auto*)

lemma *infinite-descent0*[*case-names 0 smaller*]:
$[\![\ P\ 0;\ !!n.\ n>0 \Longrightarrow \neg\ P\ n \Longrightarrow (\exists\, m::nat.\ m < n \wedge \neg P\ m)\ ]\!] \Longrightarrow P\ n$
by (*rule infinite-descent*) (*case-tac n>0*, *auto*)

Infinite descent using a mapping to $\mathbb{N}$: $P(x)$ is true for all $x \in D$ if there exists a $V : D \to \mathbb{N}$ and

- case "0": given $V(x) = 0$ prove $P(x)$,
- case "smaller": given $V(x) > 0$ and $\neg P(x)$ prove there exists a $y \in D$ such that $V(y) < V(x)$ and $\neg P(y)$.

NB: the proof also shows how to use the previous lemma.

corollary *infinite-descent0-measure* [*case-names 0 smaller*]:
assumes *A0*: $!!x.\ V\ x = (0::nat) \Longrightarrow P\ x$
and *A1*: $!!x.\ V\ x > 0 \Longrightarrow \neg P\ x \Longrightarrow (\exists\, y.\ V\ y < V\ x \wedge \neg P\ y)$
shows $P\ x$

proof −
obtain n **where** $n = V\ x$ **by** *auto*
moreover have $\bigwedge x.\ V\ x = n \Longrightarrow P\ x$
proof (*induct n rule*: *infinite-descent0*)
case *0* — i.e. $V(x) = 0$
with *A0* **show** $P\ x$ **by** *auto*
next — now $n > 0$ and $P(x)$ does not hold for some x with $V(x) = n$
case (*smaller n*)
then obtain x **where** *vxn*: $V\ x = n$ **and** $V\ x > 0 \wedge \neg\ P\ x$ **by** *auto*
with *A1* **obtain** y **where** $V\ y < V\ x \wedge \neg\ P\ y$ **by** *auto*
with *vxn* **obtain** m **where** $m = V\ y \wedge m<n \wedge \neg\ P\ y$ **by** *auto*
then show *?case* **by** *auto*
qed
ultimately show $P\ x$ **by** *auto*
qed

Again, without explicit base case:

lemma *infinite-descent-measure*:
assumes $!!x.\ \neg\ P\ x \Longrightarrow \exists\, y.\ (V::'a\Rightarrow nat)\ y < V\ x \wedge \neg\ P\ y$ **shows** $P\ x$
proof −
from *assms* **obtain** n **where** $n = V\ x$ **by** *auto*
moreover have $!!x.\ V\ x = n \Longrightarrow P\ x$
proof (*induct n rule*: *infinite-descent*, *auto*)
fix x **assume** $\neg\ P\ x$
with *assms* **show** $\exists\, m < V\ x.\ \exists\, y.\ V\ y = m \wedge \neg\ P\ y$ **by** *auto*
qed
ultimately show $P\ x$ **by** *auto*
qed

A [clumsy] way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma *less-mono-imp-le-mono*:
$[\![\ !!i\ j::nat.\ i < j \Longrightarrow f\ i < f\ j;\ i \leq j\ ]\!] \Longrightarrow f\ i \leq ((f\ j)::nat)$
by (*simp add*: *order-le-less*) (*blast*)

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j$ ==> $i + k \leq j + (k::nat)$
by (*rule add-right-mono*)

non-strict, in both arguments

lemma *add-le-mono*: [| $i \leq j$; $k \leq l$ |] ==> $i + k \leq j + (l::nat)$
by (*rule add-mono*)

lemma *le-add2*: $n \leq ((m + n)::nat)$
by (*insert add-right-mono* [*of 0 m n*], *simp*)

lemma *le-add1*: $n \leq ((n + m)::nat)$
by (*simp add*: *add-commute*, *rule le-add2*)

lemma *less-add-Suc1*: $i < Suc\ (i + m)$
by (*rule le-less-trans*, *rule le-add1*, *rule lessI*)

lemma *less-add-Suc2*: $i < Suc\ (m + i)$
by (*rule le-less-trans*, *rule le-add2*, *rule lessI*)

lemma *less-iff-Suc-add*: $(m < n) = (\exists k.\ n = Suc\ (m + k))$
by (*iprover intro*!: *less-add-Suc1 less-imp-Suc-add*)

lemma *trans-le-add1*: $(i::nat) \leq j \Longrightarrow i \leq j + m$
by (*rule le-trans*, *assumption*, *rule le-add1*)

lemma *trans-le-add2*: $(i::nat) \leq j \Longrightarrow i \leq m + j$
by (*rule le-trans*, *assumption*, *rule le-add2*)

lemma *trans-less-add1*: $(i::nat) < j \Longrightarrow i < j + m$
by (*rule less-le-trans*, *assumption*, *rule le-add1*)

lemma *trans-less-add2*: $(i::nat) < j \Longrightarrow i < m + j$
by (*rule less-le-trans*, *assumption*, *rule le-add2*)

lemma *add-lessD1*: $i + j < (k::nat) \Longrightarrow i < k$
apply (*rule le-less-trans* [*of* - *i+j*])
apply (*simp-all add*: *le-add1*)
done

lemma *not-add-less1* [*iff*]: $\sim (i + j < (i::nat))$
apply (*rule notI*)
apply (*drule add-lessD1*)
apply (*erule less-irrefl* [*THEN notE*])
done

lemma *not-add-less2* [*iff*]: $\sim (j + i < (i::nat))$
by (*simp add*: *add-commute*)

lemma *add-leD1*: $m + k \leq n \Longrightarrow m \leq (n::nat)$
apply (*rule order-trans* [*of* - *m+k*])
apply (*simp-all add*: *le-add1*)
done

lemma *add-leD2*: $m + k \leq n \Longrightarrow k \leq (n::nat)$
apply (*simp add*: *add-commute*)
apply (*erule add-leD1*)
done

lemma *add-leE*: $(m::nat) + k \leq n \Longrightarrow (m \leq n \Longrightarrow k \leq n \Longrightarrow R) \Longrightarrow R$
by (*blast dest*: *add-leD1 add-leD2*)

needs !!k for *add-ac* to work

lemma *less-add-eq-less*: !!k::*nat*. $k < l$ ==> $m + l = k + n$ ==> $m < n$
by (*force simp del*: *add-Suc-right*
simp add: *less-iff-Suc-add add-Suc-right* [*symmetric*] *add-ac*)

12.4.8 More results about difference

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add-diff-inverse*: ~ $m < n$ ==> $n + (m - n) = (m::nat)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *le-add-diff-inverse* [*simp*]: $n \leq m$ ==> $n + (m - n) = (m::nat)$
by (*simp add*: *add-diff-inverse linorder-not-less*)

lemma *le-add-diff-inverse2* [*simp*]: $n \leq m$ ==> $(m - n) + n = (m::nat)$
by (*simp add*: *add-commute*)

lemma *Suc-diff-le*: $n \leq m$ ==> *Suc* $m - n$ = *Suc* $(m - n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-less-Suc*: $m - n <$ *Suc* m
apply (*induct m n rule*: *diff-induct*)
apply (*erule-tac* [*3*] *less-SucE*)
apply (*simp-all add*: *less-Suc-eq*)
done

lemma *diff-le-self* [*simp*]: $m - n \leq (m::nat)$
by (*induct m n rule*: *diff-induct*) (*simp-all add*: *le-SucI*)

lemma *le-iff-add*: $(m::nat) \leq n = (\exists k.\ n = m + k)$
by (*auto simp*: *le-add1 dest*!: *le-add-diff-inverse sym* [*of* - *n*])

lemma *less-imp-diff-less*: $(j::nat) < k$ ==> $j - n < k$
by (*rule le-less-trans*, *rule diff-le-self*)

lemma *diff-Suc-less* [*simp*]: *0*<n ==> $n -$ *Suc* $i < n$
by (*cases n*) (*auto simp add*: *le-simps*)

lemma *diff-add-assoc*: $k \leq (j::nat)$ ==> $(i + j) - k = i + (j - k)$
by (*induct j k rule*: *diff-induct*) *simp-all*

lemma *diff-add-assoc2*: $k \leq (j::nat)$ ==> $(j + i) - k = (j - k) + i$
by (*simp add*: *add-commute diff-add-assoc*)

lemma *le-imp-diff-is-add*: $i \leq (j::nat)$ ==> $(j - i = k) = (j = k + i)$
by (*auto simp add*: *diff-add-inverse2*)

lemma *diff-is-0-eq* [*simp*]: $((m::nat) - n = 0) = (m \leq n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-is-0-eq'* [*simp*]: $m \leq n \Longrightarrow (m::nat) - n = 0$
by (*rule iffD2*, *rule diff-is-0-eq*)

lemma *zero-less-diff* [*simp*]: $(0 < n - (m::nat)) = (m < n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *less-imp-add-positive*:
assumes $i < j$
shows $\exists k::nat.\ 0 < k\ \&\ i + k = j$
proof
from *assms* **show** $0 < j - i\ \&\ i + (j - i) = j$
by (*simp add*: *order-less-imp-le*)
qed

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*:
fixes *n m* :: *nat*
shows $n - m + m = max\ n\ m$
by (*simp add*: *max-def not-le order-less-imp-le*)

lemma *nat-diff-split*:
$P(a - b::nat) = ((a<b \longrightarrow P\ 0)\ \&\ (ALL\ d.\ a = b + d \longrightarrow P\ d))$
— elimination of $-$ on *nat*
by (*cases* $a < b$)
(*auto simp add*: *diff-is-0-eq* [*THEN iffD2*] *diff-add-inverse*
not-less le-less dest!: *sym* [*of a*] *sym* [*of b*] *add-eq-self-zero*)

lemma *nat-diff-split-asm*:
$P(a - b::nat) = (\sim\ (a < b\ \&\ \sim\ P\ 0 \mid (EX\ d.\ a = b + d\ \&\ \sim\ P\ d)))$
— elimination of $-$ on *nat* in assumptions
by (*auto split*: *nat-diff-split*)

12.4.9 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \leq (j::nat) \Longrightarrow i * k \leq j * k$
by (*simp add*: *mult-right-mono*)

lemma *mult-le-mono2*: $i \leq (j::nat) \Longrightarrow k * i \leq k * j$
by (*simp add*: *mult-left-mono*)

$\leq$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq (j::nat) \Longrightarrow k \leq l \Longrightarrow i * k \leq j * l$
by (*simp add*: *mult-mono*)

lemma *mult-less-mono1*: $(i::nat) < j \Longrightarrow 0 < k \Longrightarrow i * k < j * k$
by (*simp add*: *mult-strict-right-mono*)

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $(0 < (m::nat) * n) = (0 < m \ \& \ 0 < n)$
apply (*induct m*)
apply *simp*
apply (*case-tac n*)
apply *simp-all*
done

lemma *one-le-mult-iff* [*simp*]: $(Suc\ 0 \leq m * n) = (Suc\ 0 \leq m \ \& \ Suc\ 0 \leq n)$
apply (*induct m*)
apply *simp*
apply (*case-tac n*)
apply *simp-all*
done

lemma *mult-less-cancel2* [*simp*]: $((m::nat) * k < n * k) = (0 < k \ \& \ m < n)$
apply (*safe intro*!: *mult-less-mono1*)
apply (*case-tac k*, *auto*)
apply (*simp del*: *le-0-eq add*: *linorder-not-le* [*symmetric*])
apply (*blast intro*: *mult-le-mono1*)
done

lemma *mult-less-cancel1* [*simp*]: $(k * (m::nat) < k * n) = (0 < k \ \& \ m < n)$
by (*simp add*: *mult-commute* [*of k*])

lemma *mult-le-cancel1* [*simp*]: $(k * (m::nat) \leq k * n) = (0 < k \longrightarrow m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *mult-le-cancel2* [*simp*]: $((m::nat) * k \leq n * k) = (0 < k \longrightarrow m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *Suc-mult-less-cancel1*: $(Suc\ k * m < Suc\ k * n) = (m < n)$
by (*subst mult-less-cancel1*) *simp*

lemma *Suc-mult-le-cancel1*: $(Suc\ k * m \leq Suc\ k * n) = (m \leq n)$
by (*subst mult-le-cancel1*) *simp*

lemma *le-square*: $m \leq m * (m::nat)$
by (*cases m*) (*auto intro*: *le-add1*)

lemma *le-cube*: $(m::nat) \leq m * (m * m)$
by (*cases m*) (*auto intro*: *le-add1*)

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $(m::nat) = m * n \Longrightarrow n = 1 \mid m = 0$
apply (*drule sym*)
apply (*rule disjCI*)
apply (*rule nat-less-cases*, *erule-tac* [*2*] -)
apply (*drule-tac* [*2*] *mult-less-mono2*)
apply (*auto*)

done

the lattice order on *nat*

instantiation *nat* :: *distrib-lattice*
begin

definition
(*inf* :: *nat* ⇒ *nat* ⇒ *nat*) = *min*

definition
(*sup* :: *nat* ⇒ *nat* ⇒ *nat*) = *max*

instance by *intro-classes*
(*auto simp add*: *inf-nat-def sup-nat-def max-def not-le min-def*
intro: *order-less-imp-le antisym elim*!: *order-trans order-less-trans*)

end

12.5 Embedding of the Naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

primrec
of-nat :: *nat* ⇒ *'a*
where
of-nat-0: *of-nat 0 = 0*
| *of-nat-Suc*: *of-nat* (*Suc m*) = *1* + *of-nat m*

lemma *of-nat-1* [*simp*]: *of-nat 1 = 1*
unfolding *One-nat-def* **by** *simp*

lemma *of-nat-add* [*simp*]: *of-nat* (*m* + *n*) = *of-nat m* + *of-nat n*
by (*induct m*) (*simp-all add*: *add-ac*)

lemma *of-nat-mult*: *of-nat* (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
by (*induct m*) (*simp-all add*: *add-ac left-distrib*)

primrec *of-nat-aux* :: (*'a* ⇒ *'a*) ⇒ *nat* ⇒ *'a* ⇒ *'a* **where**
of-nat-aux inc 0 i = i
| *of-nat-aux inc* (*Suc n*) *i* = *of-nat-aux inc n* (*inc i*) — tail recursive

lemma *of-nat-code* [*code*, *code unfold*, *code inline del*]:
of-nat n = of-nat-aux (λ*i*. *i* + *1*) *n 0*
proof (*induct n*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc n*)
have ⋀*i*. *of-nat-aux* (λ*i*. *i* + *1*) *n* (*i* + *1*) = *of-nat-aux* (λ*i*. *i* + *1*) *n i* + *1*

by (*induct n*) *simp-all*
from *this* [*of 0*] **have** *of-nat-aux* (λ*i*. *i* + *1*) *n 1* = *of-nat-aux* (λ*i*. *i* + *1*) *n 0* + *1*
by *simp*
with *Suc* **show** *?case* **by** (*simp add*: *add-commute*)
qed

end

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *semiring-char-0* = *semiring-1* +
assumes *of-nat-eq-iff* [*simp*]: *of-nat m* = *of-nat n* ⟷ *m* = *n*
begin

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*, *noatp*]: *0* = *of-nat n* ⟷ *0* = *n*
by (*rule of-nat-eq-iff* [*of 0*, *simplified*])

lemma *of-nat-eq-0-iff* [*simp*, *noatp*]: *of-nat m* = *0* ⟷ *m* = *0*
by (*rule of-nat-eq-iff* [*of - 0*, *simplified*])

lemma *inj-of-nat*: *inj of-nat*
by (*simp add*: *inj-on-def*)

end

context *ordered-semidom*
begin

lemma *zero-le-imp-of-nat*: *0* ≤ *of-nat m*
apply (*induct m*, *simp-all*)
apply (*erule order-trans*)
apply (*rule ord-le-eq-trans* [*OF - add-commute*])
apply (*rule less-add-one* [*THEN less-imp-le*])
done

lemma *less-imp-of-nat-less*: *m* < *n* ⟹ *of-nat m* < *of-nat n*
apply (*induct m n rule*: *diff-induct*, *simp-all*)
apply (*insert add-less-le-mono* [*OF zero-less-one zero-le-imp-of-nat*], *force*)
done

lemma *of-nat-less-imp-less*: *of-nat m* < *of-nat n* ⟹ *m* < *n*
apply (*induct m n rule*: *diff-induct*, *simp-all*)
apply (*insert zero-le-imp-of-nat*)
apply (*force simp add*: *not-less* [*symmetric*])
done

lemma *of-nat-less-iff* [*simp*]: *of-nat m* < *of-nat n* ⟷ *m* < *n*

by (*blast intro*: *of-nat-less-imp-less less-imp-of-nat-less*)

lemma *of-nat-le-iff* [*simp*]: *of-nat m* ≤ *of-nat n* ⟷ *m* ≤ *n*
by (*simp add*: *not-less* [*symmetric*] *linorder-not-less* [*symmetric*])

Every *ordered-semidom* has characteristic zero.

subclass *semiring-char-0*
proof qed (*simp add*: *eq-iff order-eq-iff*)

Special cases where either operand is zero

lemma *of-nat-0-le-iff* [*simp*]: *0* ≤ *of-nat n*
by (*rule of-nat-le-iff* [*of 0*, *simplified*])

lemma *of-nat-le-0-iff* [*simp*, *noatp*]: *of-nat m* ≤ *0* ⟷ *m* = *0*
by (*rule of-nat-le-iff* [*of - 0*, *simplified*])

lemma *of-nat-0-less-iff* [*simp*]: *0* < *of-nat n* ⟷ *0* < *n*
by (*rule of-nat-less-iff* [*of 0*, *simplified*])

lemma *of-nat-less-0-iff* [*simp*]: ¬ *of-nat m* < *0*
by (*rule of-nat-less-iff* [*of - 0*, *simplified*])

end

context *ring-1*
begin

lemma *of-nat-diff*: *n* ≤ *m* ⟹ *of-nat* (*m* − *n*) = *of-nat m* − *of-nat n*
by (*simp add*: *algebra-simps of-nat-add* [*symmetric*])

end

context *ordered-idom*
begin

lemma *abs-of-nat* [*simp*]: |*of-nat n*| = *of-nat n*
unfolding *abs-if* **by** *auto*

end

lemma *of-nat-id* [*simp*]: *of-nat n* = *n*
by (*induct n*) (*auto simp add*: *One-nat-def*)

lemma *of-nat-eq-id* [*simp*]: *of-nat* = *id*
by (*auto simp add*: *expand-fun-eq*)

12.6 The Set of Natural Numbers

context *semiring-1*

begin

definition
Nats :: *'a set* **where**
[*code del*]: *Nats* = *range of-nat*

notation (*xsymbols*)
Nats (ℕ)

lemma *of-nat-in-Nats* [*simp*]: *of-nat n* ∈ ℕ
by (*simp add*: *Nats-def*)

lemma *Nats-0* [*simp*]: *0* ∈ ℕ
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-0* [*symmetric*])
done

lemma *Nats-1* [*simp*]: *1* ∈ ℕ
apply (*simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-1* [*symmetric*])
done

lemma *Nats-add* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a* + *b* ∈ ℕ
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-add* [*symmetric*])
done

lemma *Nats-mult* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a* * *b* ∈ ℕ
apply (*auto simp add*: *Nats-def*)
apply (*rule range-eqI*)
apply (*rule of-nat-mult* [*symmetric*])
done

end

12.7 Further Arithmetic Facts Concerning the Natural Numbers

lemma *subst-equals*:
assumes *1*: *t* = *s* **and** *2*: *u* = *t*
shows *u* = *s*
using *2 1* **by** (*rule trans*)

setup *Arith-Data.setup*

use *Tools/nat-arith.ML*

declaration ⟪ *K Nat-Arith.setup* ⟫

use *Tools/lin-arith.ML*
declaration ⟪ *K Lin-Arith.setup* ⟫

lemmas [*arith-split*] = *nat-diff-split split-min split-max*

context *order*
begin

lemma *lift-Suc-mono-le*:
  assumes *mono*: !!*n*. *f n* ≤ *f*(*Suc n*) **and** *n*≤*n'*
  shows *f n* ≤ *f n'*
proof (*cases n* < *n'*)
  case *True*
  thus *?thesis*
    by (*induct n n' rule*: *less-Suc-induct*[*consumes 1*]) (*auto intro*: *mono*)
qed (*insert* ‹*n* ≤ *n'*›, *auto*) — trivial for *n* = *n'*

lemma *lift-Suc-mono-less*:
  assumes *mono*: !!*n*. *f n* < *f*(*Suc n*) **and** *n* < *n'*
  shows *f n* < *f n'*
using ‹*n* < *n'*›
by (*induct n n' rule*: *less-Suc-induct*[*consumes 1*]) (*auto intro*: *mono*)

lemma *lift-Suc-mono-less-iff*:
  (!!*n*. *f n* < *f*(*Suc n*)) ⟹ *f*(*n*) < *f*(*m*) ⟷ *n*<*m*
by(*blast intro*: *less-asym' lift-Suc-mono-less*[*of f*]
        dest: *linorder-not-less*[*THEN iffD1*] *le-eq-less-or-eq*[*THEN iffD1*])

end

lemma *mono-iff-le-Suc*: *mono f* = (∀ *n*. *f n* ≤ *f* (*Suc n*))
unfolding *mono-def*
by (*auto intro*:*lift-Suc-mono-le*[*of f*])

lemma *mono-nat-linear-lb*:
  (!!*m n*::*nat*. *m*<*n* ⟹ *f m* < *f n*) ⟹ *f*(*m*)+*k* ≤ *f*(*m*+*k*)
apply(*induct-tac k*)
 apply *simp*
apply(*erule-tac x*=*m*+*n* **in** *meta-allE*)
apply(*erule-tac x*=*Suc*(*m*+*n*) **in** *meta-allE*)
apply *simp*
done

Subtraction laws, mostly by Clemens Ballarin

lemma *diff-less-mono*: [| *a* < (*b*::*nat*); *c* ≤ *a* |] ==> *a*−*c* < *b*−*c*
by *arith*

lemma *less-diff-conv*: $(i < j-k) = (i+k < (j::nat))$
by *arith*

lemma *le-diff-conv*: $(j-k \leq (i::nat)) = (j \leq i+k)$
by *arith*

lemma *le-diff-conv2*: $k \leq j \Longrightarrow (i \leq j-k) = (i+k \leq (j::nat))$
by *arith*

lemma *diff-diff-cancel* [*simp*]: $i \leq (n::nat) \Longrightarrow n - (n - i) = i$
by *arith*

lemma *le-add-diff*: $k \leq (n::nat) \Longrightarrow m \leq n + m - k$
by *arith*

lemma *diff-less*[*simp*]: $!!m::nat.\ [|\ 0<n;\ 0<m\ |] \Longrightarrow m - n < m$
by *arith*

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow ((m-k) - (n-k)) = (m-n)$
by (*simp split add*: *nat-diff-split*)

lemma *eq-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k = n-k) = (m=n)$
by (*auto split add*: *nat-diff-split*)

lemma *less-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k < n-k) = (m<n)$
by (*auto split add*: *nat-diff-split*)

lemma *le-diff-iff*: $[|\ k \leq m;\ \ k \leq (n::nat)\ |] \Longrightarrow (m-k \leq n-k) = (m \leq n)$
by (*auto split add*: *nat-diff-split*)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq (n::nat) \Longrightarrow (m-l) \leq (n-l)$
by (*simp split add*: *nat-diff-split*)

lemma *diff-le-mono2*: $m \leq (n::nat) \Longrightarrow (l-n) \leq (l-m)$
by (*simp split add*: *nat-diff-split*)

lemma *diff-less-mono2*: $[|\ m < (n::nat);\ m<l\ |] \Longrightarrow (l-n) < (l-m)$
by (*simp split add*: *nat-diff-split*)

lemma *diffs0-imp-equal*: $!!m::nat.\ [|\ m-n = 0;\ n-m = 0\ |] \Longrightarrow\ m=n$
by (*simp split add*: *nat-diff-split*)

lemma *min-diff*: $min\ (m - (i::nat))\ (n - i) = min\ m\ n - i$
unfolding *min-def* **by** *auto*

lemma *inj-on-diff-nat*:

assumes *k-le-n*: $\forall n \in N.\ k \leq (n::nat)$
shows *inj-on* $(\lambda n.\ n - k)\ N$
proof (*rule inj-onI*)
fix $x\ y$
assume a: $x \in N\ y \in N\ x - k = y - k$
with *k-le-n* **have** $x - k + k = y - k + k$ **by** *auto*
with a *k-le-n* **show** $x = y$ **by** *auto*
qed

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j \longrightarrow i - (j - k) = i + (k::nat) - j$
by *arith*

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j \Longrightarrow m - Suc\ (j - k) = m + k - Suc\ j$
by *arith*

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j \Longrightarrow Suc\ (j - k) - m = Suc\ j - (k + m)$
by *arith*

Lemmas for ex/Factorization

lemma *one-less-mult*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow Suc\ 0 < m*n$
by (*cases* m) *auto*

lemma *n-less-m-mult-n*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow n<m*n$
by (*cases* m) *auto*

lemma *n-less-n-mult-m*: $[|\ Suc\ 0 < n;\ Suc\ 0 < m\ |] \Longrightarrow n<n*m$
by (*cases* m) *auto*

Specialized induction principles that work "backwards":

lemma *inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: $i <= j$
assumes *base*: $P\ j$
assumes *step*: $!!i.\ [|\ i < j;\ P\ (Suc\ i)\ |] \Longrightarrow P\ i$
shows $P\ i$
using *less*
proof (*induct* $d{==}j - i$ *arbitrary*: i)
case (*0* i)
hence $i = j$ **by** *simp*
with *base* **show** *?case* **by** *simp*
next
case (*Suc* $d\ i$)
hence $i < j\ P\ (Suc\ i)$
by *simp-all*
thus $P\ i$ **by** (*rule step*)
qed

lemma *strict-inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: $i < j$

assumes *base*: !!*i*. *j* = *Suc i* ==> *P i*
assumes *step*: !!*i*. [| *i* < *j*; *P* (*Suc i*) |] ==> *P i*
shows *P i*
using *less*
proof (*induct d*==*j* − *i* − *1 arbitrary*: *i*)
case (*0 i*)
with ‹*i* < *j*› **have** *j* = *Suc i* **by** *simp*
with *base* **show** *?case* **by** *simp*
next
case (*Suc d i*)
hence *i* < *j P* (*Suc i*)
by *simp-all*
thus *P i* **by** (*rule step*)
qed

lemma *zero-induct-lemma*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P* (*k* − *i*)
using *inc-induct*[*of k* − *i k P*, *simplified*] **by** *blast*

lemma *zero-induct*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P 0*
using *inc-induct*[*of 0 k P*] **by** *blast*

lemma *nat-not-singleton*: (∀ *x*. *x* = (*0*::*nat*)) = *False*
by *auto*

lemmas *add-diff-assoc* = *diff-add-assoc* [*symmetric*]
lemmas *add-diff-assoc2* = *diff-add-assoc2*[*symmetric*]
declare *diff-diff-left* [*simp*] *add-diff-assoc* [*simp*] *add-diff-assoc2*[*simp*]

At present we prove no analogue of *not-less-Least* or *Least-Suc*, since there appears to be no need.

12.8 size of a datatype value

class *size* =
fixes *size* :: *'a* ⇒ *nat* — see further theory *Wellfounded*

end

13 Product-Type: Cartesian products

theory *Product-Type*
imports *Inductive*
uses
(*Tools/split-rule.ML*)
(*Tools/inductive-set-package.ML*)
(*Tools/inductive-realizer.ML*)
(*Tools/datatype-realizer.ML*)

begin

13.1 *bool* is a datatype

rep-datatype *True False* **by** (*auto intro*: *bool-induct*)

declare *case-split* [*cases type*: *bool*]
— prefer plain propositional version

lemma
shows [*code*]: *eq-class.eq False P* ⟷ ¬ *P*
and [*code*]: *eq-class.eq True P* ⟷ *P*
and [*code*]: *eq-class.eq P False* ⟷ ¬ *P*
and [*code*]: *eq-class.eq P True* ⟷ *P*
and [*code nbe*]: *eq-class.eq P P* ⟷ *True*
by (*simp-all add*: *eq*)

code-const *eq-class.eq* :: *bool* ⇒ *bool* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-instance *bool* :: *eq*
(*Haskell* −)

13.2 Unit

typedef *unit* = {*True*}
proof
show *True* : *?unit* **..**
qed

definition
Unity :: *unit* ('())
where
() = *Abs-unit True*

lemma *unit-eq* [*noatp*]: *u* = ()
by (*induct u*) (*simp add*: *unit-def Unity-def*)

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

ML ⟪
val unit-eq-proc =
let val unit-meta-eq = *mk-meta-eq* @{*thm unit-eq*} *in*
Simplifier.simproc @{*theory*} *unit-eq* [*x*::*unit*]
(*fn* - => *fn* - => *fn t* => *if HOLogic.is-unit t then NONE else SOME unit-meta-eq*)
end;

Addsimprocs [*unit-eq-proc*];

⟫

rep-datatype () **by** *simp*

lemma *unit-all-eq1*: (!!*x*::*unit*. *PROP P x*) == *PROP P* ()
by *simp*

lemma *unit-all-eq2*: (!!*x*::*unit*. *PROP P*) == *PROP P*
by (*rule triv-forall-equality*)

This rewrite counters the effect of *unit-eq-proc* on %*u*::*unit*. *f u*, replacing it by *f* rather than by %*u*. *f* ().

lemma *unit-abs-eta-conv* [*simp*,*noatp*]: (%*u*::*unit*. *f* ()) = *f*
by (*rule ext*) *simp*

code generator setup

instance *unit* :: *eq* **..**

lemma [*code*]:
eq-class.eq (*u*::*unit*) *v* ⟷ *True* **unfolding** *eq unit-eq* [*of u*] *unit-eq* [*of v*] **by** *rule*+

code-type *unit*
(*SML unit*)
(*OCaml unit*)
(*Haskell* ())

code-instance *unit* :: *eq*
(*Haskell* −)

code-const *eq-class.eq* :: *unit* ⇒ *unit* ⇒ *bool*
(*Haskell* **infixl** 4 ==)

code-const *Unity*
(*SML* ())
(*OCaml* ())
(*Haskell* ())

code-reserved *SML*
unit

code-reserved *OCaml*
unit

13.3 Pairs

13.3.1 Product type, basic operations and concrete syntax

definition

Pair-Rep :: $'a \Rightarrow 'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
where
Pair-Rep a b = $(\lambda x\ y.\ x = a \wedge y = b)$

global

typedef (*Prod*)
($'a$, $'b$) $*$ (**infixr** $*$ *20*)
= $\{f.\ \exists\ a\ b.\ f = Pair\text{-}Rep\ (a::'a)\ (b::'b)\}$
proof
fix *a b* **show** *Pair-Rep a b* $\in$ *?Prod*
by *rule+*
qed

syntax (*xsymbols*)
$*$ :: [*type*, *type*] => *type* ((- $\times$/ -) [*21*, *20*] *20*)
syntax (*HTML* **output**)
$*$ :: [*type*, *type*] => *type* ((- $\times$/ -) [*21*, *20*] *20*)

consts
Pair :: $'a \Rightarrow 'b \Rightarrow 'a \times 'b$
fst :: $'a \times 'b \Rightarrow 'a$
snd :: $'a \times 'b \Rightarrow 'b$
split :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
curry :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$

local

defs
Pair-def: *Pair a b* == *Abs-Prod* (*Pair-Rep a b*)
fst-def: *fst p* == *THE a. EX b. p* = *Pair a b*
snd-def: *snd p* == *THE b. EX a. p* = *Pair a b*
split-def: *split* == (%*c p. c* (*fst p*) (*snd p*))
curry-def: *curry* == (%*c x y. c* (*Pair x y*))

Patterns – extends pre-defined type *pttrn* used in abstractions.

nonterminals
tuple-args patterns

syntax
-tuple :: $'a$ => *tuple-args* => $'a * 'b$ (($1'(-,/ -')$))
-tuple-arg :: $'a$ => *tuple-args* (-)
-tuple-args :: $'a$ => *tuple-args* => *tuple-args* (-,/ -)
-pattern :: [*pttrn*, *patterns*] => *pttrn* ($'(-,/ -')$)
:: *pttrn* => *patterns* (-)
-patterns :: [*pttrn*, *patterns*] => *patterns* (-,/ -)

translations
(*x*, *y*) == *Pair x y*

```
  -tuple x (-tuple-args y z) == -tuple x (-tuple-arg (-tuple y z))
  %(x,y,zs).b  == split(%x (y,zs).b)
  %(x,y).b     == split(%x y. b)
  -abs (Pair x y) t => %(x,y).t
```

print-translation ⟪

```
let fun split-tr' [Abs (x,T,t as (Abs abs))] =
      (* split (%x y. t) => %(x,y) t *)
      let val (y,t') = atomic-abs-tr' abs;
          val (x',t'') = atomic-abs-tr' (x,T,t');

      in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end
    | split-tr' [Abs (x,T,(s as Const (split,-)$t))] =
      (* split (%x. (split (%y z. t))) => %(x,y,z). t *)
      let val (Const (-abs,-)$(Const (-pattern,-)$y$z)$t') = split-tr' [t];
          val (x',t'') = atomic-abs-tr' (x,T,t');
      in Syntax.const -abs$
          (Syntax.const -pattern$x'$(Syntax.const -patterns$y$z))$t'' end
    | split-tr' [Const (split,-)$t] =
      (* split (split (%x y z. t)) => %((x,y),z). t *)
      split-tr' [(split-tr' [t])] (* inner split-tr' creates next pattern *)
    | split-tr' [Const (-abs,-)$x-y$(Abs abs)] =
      (* split (%pttrn z. t) => %(pttrn,z). t *)
      let val (z,t) = atomic-abs-tr' abs;
      in Syntax.const -abs $ (Syntax.const -pattern $x-y$z) $ t end
    | split-tr' - =  raise Match;
in [(split, split-tr')]
end
```

⟫

typed-print-translation ⟪

```
let
  fun split-guess-names-tr' - T [Abs (x,-,Abs -)] = raise Match
    | split-guess-names-tr' - T  [Abs (x,xT,t)] =
        (case (head-of t) of
           Const (split,-) => raise Match
         | - => let
                  val (-::yT::-) = binder-types (domain-type T) handle Bind => raise
Match;
                  val (y,t') = atomic-abs-tr' (y,yT,(incr-boundvars 1 t)$Bound 0);
                  val (x',t'') = atomic-abs-tr' (x,xT,t');
                in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end)
    | split-guess-names-tr' - T [t] =
      (case (head-of t) of
          Const (split,-) => raise Match
```

```
      | - => let
              val (xT::yT::-) = binder-types (domain-type T) handle Bind =>
raise Match;
              val (y,t') =
                  atomic-abs-tr' (y,yT,(incr-boundvars 2 t)$Bound 1$Bound 0);
              val (x',t'') = atomic-abs-tr' (x,xT,t');
            in Syntax.const -abs $ (Syntax.const -pattern $x'$y) $ t'' end)
   | split-guess-names-tr' - - - = raise Match;
in [(split, split-guess-names-tr')]
end
⟫
```

Towards a datatype declaration

```
lemma surj-pair [simp]: EX x y. p = (x, y)
  apply (unfold Pair-def)
  apply (rule Rep-Prod [unfolded Prod-def, THEN CollectE])
  apply (erule exE, erule exE, rule exI, rule exI)
  apply (rule Rep-Prod-inverse [symmetric, THEN trans])
  apply (erule arg-cong)
  done

lemma PairE [cases type: *]:
  obtains x y where p = (x, y)
  using surj-pair [of p] by blast

lemma ProdI: Pair-Rep a b ∈ Prod
  unfolding Prod-def by rule+

lemma Pair-Rep-inject: Pair-Rep a b = Pair-Rep a' b' ⟹ a = a' ∧ b = b'
  unfolding Pair-Rep-def by (drule fun-cong, drule fun-cong) blast

lemma inj-on-Abs-Prod: inj-on Abs-Prod Prod
  apply (rule inj-on-inverseI)
  apply (erule Abs-Prod-inverse)
  done

lemma Pair-inject:
  assumes (a, b) = (a', b')
    and a = a' ==> b = b' ==> R
  shows R
  apply (insert prems [unfolded Pair-def])
   apply (rule inj-on-Abs-Prod [THEN inj-onD, THEN Pair-Rep-inject, THEN
conjE])
  apply (assumption | rule ProdI)+
  done

rep-datatype (prod) Pair
proof −
  fix P p
```

assume $\bigwedge x\ y.\ P\ (x,\ y)$
then show $P\ p$ **by** (*cases* p) *simp*
qed (*auto elim*: *Pair-inject*)

lemmas *Pair-eq* = *prod.inject*

lemma *fst-conv* [*simp*, *code*]: *fst* $(a,\ b) = a$
unfolding *fst-def* **by** *blast*

lemma *snd-conv* [*simp*, *code*]: *snd* $(a,\ b) = b$
unfolding *snd-def* **by** *blast*

13.3.2 Basic rules and proof tools

lemma *fst-eqD*: *fst* $(x,\ y) = a$ ==> $x = a$
by *simp*

lemma *snd-eqD*: *snd* $(x,\ y) = a$ ==> $y = a$
by *simp*

lemma *pair-collapse* [*simp*]: (*fst* p, *snd* p) = p
by (*cases* p) *simp*

lemmas *surjective-pairing* = *pair-collapse* [*symmetric*]

lemma *split-paired-all*: (!!x. *PROP* $P\ x$) == (!!$a\ b$. *PROP* $P\ (a,\ b)$)
proof
fix $a\ b$
assume !!x. *PROP* $P\ x$
then show *PROP* $P\ (a,\ b)$ **.**
next
fix x
assume !!$a\ b$. *PROP* $P\ (a,\ b)$
from ‹*PROP* P (*fst* x, *snd* x)› **show** *PROP* $P\ x$ **by** *simp*
qed

The rule *split-paired-all* does not work with the Simplifier because it also affects premises in congrence rules, where this can lead to premises of the form !!$a\ b$. ... = *?P*$(a,\ b)$ which cannot be solved by reflexivity.

lemmas *split-tupled-all* = *split-paired-all unit-all-eq2*

ML ‹‹
(∗ *replace parameters of product type by individual component parameters* ∗)
val safe-full-simp-tac = *generic-simp-tac true* (*true*, *false*, *false*);
local (∗ *filtering with exists-paired-all is an essential optimization* ∗)
fun exists-paired-all (*Const* (*all*, -) $ *Abs* (-, *T*, *t*)) =
can HOLogic.dest-prodT T orelse exists-paired-all t
| *exists-paired-all* (*t* $ *u*) = *exists-paired-all t orelse exists-paired-all u*
| *exists-paired-all* (*Abs* (-, -, *t*)) = *exists-paired-all t*

```
    | exists-paired-all - = false;
  val ss = HOL-basic-ss
   addsimps [@{thm split-paired-all}, @{thm unit-all-eq2}, @{thm unit-abs-eta-conv}]
    addsimprocs [unit-eq-proc];
in
  val split-all-tac = SUBGOAL (fn (t, i) =>
    if exists-paired-all t then safe-full-simp-tac ss i else no-tac);
  val unsafe-split-all-tac = SUBGOAL (fn (t, i) =>
    if exists-paired-all t then full-simp-tac ss i else no-tac);
  fun split-all th =
   if exists-paired-all (Thm.prop-of th) then full-simplify ss th else th;
end;
⟫

declaration ⟪ fn - =>
  Classical.map-cs (fn cs => cs addSbefore (split-all-tac, split-all-tac))
⟫

lemma split-paired-All [simp]: (ALL x. P x) = (ALL a b. P (a, b))
  — [iff] is not a good idea because it makes blast loop
  by fast

lemma split-paired-Ex [simp]: (EX x. P x) = (EX a b. P (a, b))
  by fast

lemma Pair-fst-snd-eq: s = t ⟷ fst s = fst t ∧ snd s = snd t
  by (cases s, cases t) simp

lemma prod-eqI [intro?]: fst p = fst q ⟹ snd p = snd q ⟹ p = q
  by (simp add: Pair-fst-snd-eq)
```

13.3.3 *split* and *curry*

```
lemma split-conv [simp, code]: split f (a, b) = f a b
  by (simp add: split-def)

lemma curry-conv [simp, code]: curry f a b = f (a, b)
  by (simp add: curry-def)

lemmas split = split-conv  — for backwards compatibility

lemma splitI: f a b ⟹ split f (a, b)
  by (rule split-conv [THEN iffD2])

lemma splitD: split f (a, b) ⟹ f a b
  by (rule split-conv [THEN iffD1])

lemma curryI [intro!]: f (a, b) ⟹ curry f a b
  by (simp add: curry-def)
```

lemma *curryD* [*dest*!]: *curry f a b* $\Longrightarrow$ *f* (*a*, *b*)
by (*simp add*: *curry-def*)

lemma *curryE*: *curry f a b* $\Longrightarrow$ (*f* (*a*, *b*) $\Longrightarrow$ *Q*) $\Longrightarrow$ *Q*
by (*simp add*: *curry-def*)

lemma *curry-split* [*simp*]: *curry* (*split f*) = *f*
by (*simp add*: *curry-def split-def*)

lemma *split-curry* [*simp*]: *split* (*curry f*) = *f*
by (*simp add*: *curry-def split-def*)

lemma *split-Pair* [*simp*]: (λ(*x*, *y*). (*x*, *y*)) = *id*
by (*simp add*: *split-def id-def*)

lemma *split-eta*: (λ(*x*, *y*). *f* (*x*, *y*)) = *f*
— Subsumes the old *split-Pair* when *f* is the identity function.
by (*rule ext*) *auto*

lemma *split-comp*: *split* (*f* ∘ *g*) *x* = *f* (*g* (*fst x*)) (*snd x*)
by (*cases x*) *simp*

lemma *split-twice*: *split f* (*split g p*) = *split* (λ*x y*. *split f* (*g x y*)) *p*
unfolding *split-def* **..**

lemma *split-paired-The*: (*THE x*. *P x*) = (*THE* (*a*, *b*). *P* (*a*, *b*))
— Can't be added to simpset: loops!
by (*simp add*: *split-eta*)

lemma *The-split*: *The* (*split P*) = (*THE xy*. *P* (*fst xy*) (*snd xy*))
by (*simp add*: *split-def*)

lemma *split-weak-cong*: *p* = *q* $\Longrightarrow$ *split c p* = *split c q*
— Prevents simplification of *c*: much faster
by (*erule arg-cong*)

lemma *cond-split-eta*: (!!*x y*. *f x y* = *g* (*x*, *y*)) ==> (%(*x*, *y*). *f x y*) = *g*
by (*simp add*: *split-eta*)

Simplification procedure for *cond-split-eta*. Using *split-eta* as a rewrite rule is not general enough, and using *cond-split-eta* directly would render some existing proofs very inefficient; similarly for *split-beta*.

ML ⟪

```
local
  val cond-split-eta-ss = HOL-basic-ss addsimps [thm cond-split-eta]
  fun  Pair-pat k 0 (Bound m) = (m = k)
  |    Pair-pat k i (Const (Pair,  -) $ Bound m $ t) = i > 0 andalso
```

```
                m = k+i andalso Pair-pat k (i-1) t
  |   Pair-pat - - - = false;
  fun no-args k i (Abs (-, -, t)) = no-args (k+1) i t
  |   no-args k i (t $ u) = no-args k i t andalso no-args k i u
  |   no-args k i (Bound m) = m < k orelse m > k+i
  |   no-args - - - = true;
  fun split-pat tp i (Abs (-,-,t)) = if tp 0 i t then SOME (i,t) else NONE
  |   split-pat tp i (Const (split, -) $ Abs (-, -, t)) = split-pat tp (i+1) t
  |   split-pat tp i - = NONE;
  fun metaeq ss lhs rhs = mk-meta-eq (Goal.prove (Simplifier.the-context ss) [] []
        (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs,rhs)))
        (K (simp-tac (Simplifier.inherit-context ss cond-split-eta-ss) 1)));

  fun beta-term-pat k i (Abs (-, -, t)) = beta-term-pat (k+1) i t
  |   beta-term-pat k i (t $ u) = Pair-pat k i (t $ u) orelse
                    (beta-term-pat k i t andalso beta-term-pat k i u)
  |   beta-term-pat k i t = no-args k i t;
  fun  eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
  |    eta-term-pat - - - = false;
  fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
  |   subst arg k i (t $ u) = if Pair-pat k i (t $ u) then incr-boundvars k arg
                          else (subst arg k i t $ subst arg k i u)
  |   subst arg k i t = t;
  fun beta-proc ss (s as Const (split, -) $ Abs (-, -, t) $ arg) =
        (case split-pat beta-term-pat 1 t of
        SOME (i,f) => SOME (metaeq ss s (subst arg 0 i f))
        | NONE => NONE)
  |   beta-proc - - = NONE;
  fun eta-proc ss (s as Const (split, -) $ Abs (-, -, t)) =
        (case split-pat eta-term-pat 1 t of
          SOME (-,ft) => SOME (metaeq ss s (let val (f $ arg) = ft in f end))
        | NONE => NONE)
  |   eta-proc - - = NONE;
in
  val split-beta-proc = Simplifier.simproc (the-context ()) split-beta [split f z] (K
beta-proc);
  val split-eta-proc = Simplifier.simproc (the-context ()) split-eta [split f] (K eta-proc);
end;

Addsimprocs [split-beta-proc, split-eta-proc];
⟫
```

lemma *split-beta* [*mono*]: (%(*x*, *y*). *P x y*) *z* = *P* (*fst z*) (*snd z*)
by (*subst surjective-pairing*, *rule split-conv*)

lemma *split-split* [*noatp*]: *R*(*split c p*) = (*ALL x y*. *p* = (*x*, *y*) −−> *R*(*c x y*))
— For use with *split* and the Simplifier.
by (*insert surj-pair* [*of p*], *clarify*, *simp*)

split-split could be declared as [*split*] done after the Splitter has been speeded

up significantly; precompute the constants involved and don't do anything unless the current goal contains one of those constants.

lemma *split-split-asm* [*noatp*]: *R* (*split c p*) = (~(*EX x y. p* = (*x*, *y*) & (~*R* (*c x y*))))
by (*subst split-split*, *simp*)

split used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *split* as rewrite.

lemma *splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *split c p*
  apply (*simp only*: *split-tupled-all*)
  apply (*simp* (*no-asm-simp*))
  done

lemma *splitI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *split c p x*
  apply (*simp only*: *split-tupled-all*)
  apply (*simp* (*no-asm-simp*))
  done

lemma *splitE*: *split c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
  by (*induct p*) (*auto simp add*: *split-def*)

lemma *splitE'*: *split c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
  by (*induct p*) (*auto simp add*: *split-def*)

lemma *splitE2*:
  [| *Q* (*split P z*); !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R* |] ==> *R*
proof −
  assume *q*: *Q* (*split P z*)
  assume *r*: !!*x y*. [|*z* = (*x*, *y*); *Q* (*P x y*)|] ==> *R*
  show *R*
    apply (*rule r surjective-pairing*)+
    apply (*rule split-beta* [*THEN subst*], *rule q*)
    done
qed

lemma *splitD'*: *split R* (*a*,*b*) *c* ==> *R a b c*
  by *simp*

lemma *mem-splitI*: *z*: *c a b* ==> *z*: *split c* (*a*, *b*)
  by *simp*

lemma *mem-splitI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *z*: *c a b* |] ==> *z*: *split c p*
by (*simp only*: *split-tupled-all*, *simp*)

lemma *mem-splitE*:
  assumes *major*: *z*: *split c p*

and *cases*: !!*x y*. [| *p* = (*x*,*y*); *z*: *c x y* |] ==> *Q*
shows *Q*
by (*rule major* [*unfolded split-def*] *cases surjective-pairing*)+

declare *mem-splitI2* [*intro*!] *mem-splitI* [*intro*!] *splitI2'* [*intro*!] *splitI2* [*intro*!] *splitI* [*intro*!]
declare *mem-splitE* [*elim*!] *splitE'* [*elim*!] *splitE* [*elim*!]

ML ⟪
local (∗ *filtering with exists-p-split is an essential optimization* ∗)
fun exists-p-split (*Const* (*split*,-) $ - $ (*Const* (*Pair*,-)$-$-)) = *true*
| *exists-p-split* (*t* $ *u*) = *exists-p-split t orelse exists-p-split u*
| *exists-p-split* (*Abs* (-, -, *t*)) = *exists-p-split t*
| *exists-p-split* - = *false*;
val ss = *HOL-basic-ss addsimps* [*thm split-conv*];
in
val split-conv-tac = *SUBGOAL* (*fn* (*t*, *i*) =>
if exists-p-split t then safe-full-simp-tac ss i else no-tac);
end;
⟫

declaration ⟪ *fn* - =>
Classical.map-cs (*fn cs* => *cs addSbefore* (*split-conv-tac*, *split-conv-tac*))
⟫

lemma *split-eta-SetCompr* [*simp*,*noatp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P* (*x*, *y*)) = *P*
by (*rule ext*) *fast*

lemma *split-eta-SetCompr2* [*simp*,*noatp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P x y*) = *split P*
by (*rule ext*) *fast*

lemma *split-part* [*simp*]: (%(*a*,*b*). *P* & *Q a b*) = (%*ab*. *P* & *split Q ab*)
— Allows simplifications of nested splits in case of independent predicates.
by (*rule ext*) *blast*

lemma *split-comp-eq*:
fixes *f* :: *'a* => *'b* => *'c* **and** *g* :: *'d* => *'a*
shows (%*u*. *f* (*g* (*fst u*)) (*snd u*)) = (*split* (%*x*. *f* (*g x*)))
by (*rule ext*) *auto*

lemma *pair-imageI* [*intro*]: (*a*, *b*) : *A* ==> *f a b* : (%(*a*, *b*). *f a b*) ' *A*
apply (*rule-tac x* = (*a*, *b*) **in** *image-eqI*)
apply *auto*
done

lemma *The-split-eq* [*simp*]: (*THE* (*x'*,*y'*). *x* = *x'* & *y* = *y'*) = (*x*, *y*)
by *blast*

Setup of internal *split-rule*.

definition
internal-split :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a* × '*b* ⇒ '*c*
where
internal-split == *split*

lemma *internal-split-conv*: *internal-split c* (*a*, *b*) = *c a b*
by (*simp only*: *internal-split-def split-conv*)

hide *const internal-split*

use *Tools/split-rule.ML*
setup *SplitRule.setup*

lemmas *prod-caseI* = *prod.cases* [*THEN iffD2*, *standard*]

lemma *prod-caseI2*: !!*p*. [| !!*a b*. *p* = (*a*, *b*) ==> *c a b* |] ==> *prod-case c p*
by *auto*

lemma *prod-caseI2'*: !!*p*. [| !!*a b*. (*a*, *b*) = *p* ==> *c a b x* |] ==> *prod-case c p x*
by (*auto simp*: *split-tupled-all*)

lemma *prod-caseE*: *prod-case c p* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y* ==> *Q*) ==> *Q*
by (*induct p*) *auto*

lemma *prod-caseE'*: *prod-case c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
by (*induct p*) *auto*

lemma *prod-case-unfold*: *prod-case* = (%*c p*. *c* (*fst p*) (*snd p*))
by (*simp add*: *expand-fun-eq*)

declare *prod-caseI2'* [*intro*!] *prod-caseI2* [*intro*!] *prod-caseI* [*intro*!]
declare *prod-caseE'* [*elim*!] *prod-caseE* [*elim*!]

lemma *prod-case-split*:
prod-case = *split*
by (*auto simp add*: *expand-fun-eq*)

lemma *prod-case-beta*:
prod-case f p = *f* (*fst p*) (*snd p*)
unfolding *prod-case-split split-beta* **..**

13.4 Further cases/induct rules for tuples

lemma *prod-cases3* [*cases type*]:
obtains (*fields*) *a b c* **where** $y = (a, b, c)$
by (*cases y*, *case-tac b*) *blast*

lemma *prod-induct3* [*case-names fields*, *induct type*]:
$(!!a\ b\ c.\ P\ (a, b, c)) ==> P\ x$
by (*cases x*) *blast*

lemma *prod-cases4* [*cases type*]:
obtains (*fields*) *a b c d* **where** $y = (a, b, c, d)$
by (*cases y*, *case-tac c*) *blast*

lemma *prod-induct4* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d.\ P\ (a, b, c, d)) ==> P\ x$
by (*cases x*) *blast*

lemma *prod-cases5* [*cases type*]:
obtains (*fields*) *a b c d e* **where** $y = (a, b, c, d, e)$
by (*cases y*, *case-tac d*) *blast*

lemma *prod-induct5* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e.\ P\ (a, b, c, d, e)) ==> P\ x$
by (*cases x*) *blast*

lemma *prod-cases6* [*cases type*]:
obtains (*fields*) *a b c d e f* **where** $y = (a, b, c, d, e, f)$
by (*cases y*, *case-tac e*) *blast*

lemma *prod-induct6* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f.\ P\ (a, b, c, d, e, f)) ==> P\ x$
by (*cases x*) *blast*

lemma *prod-cases7* [*cases type*]:
obtains (*fields*) *a b c d e f g* **where** $y = (a, b, c, d, e, f, g)$
by (*cases y*, *case-tac f*) *blast*

lemma *prod-induct7* [*case-names fields*, *induct type*]:
$(!!a\ b\ c\ d\ e\ f\ g.\ P\ (a, b, c, d, e, f, g)) ==> P\ x$
by (*cases x*) *blast*

13.4.1 Derived operations

The composition-uncurry combinator.

notation *fcomp* (**infixl** *o>* *60*)

definition
scomp :: $('a \Rightarrow 'b \times 'c) \Rightarrow ('b \Rightarrow 'c \Rightarrow 'd) \Rightarrow 'a \Rightarrow 'd$ (**infixl** *o→* *60*)

where
  $f\ o{\rightarrow}\ g = (\lambda x.\ split\ g\ (f\ x))$

lemma *scomp-apply*: $(f\ o{\rightarrow}\ g)\ x = split\ g\ (f\ x)$
  by (*simp add*: *scomp-def*)

lemma *Pair-scomp*: $Pair\ x\ o{\rightarrow}\ f = f\ x$
  by (*simp add*: *expand-fun-eq scomp-apply*)

lemma *scomp-Pair*: $x\ o{\rightarrow}\ Pair = x$
  by (*simp add*: *expand-fun-eq scomp-apply*)

lemma *scomp-scomp*: $(f\ o{\rightarrow}\ g)\ o{\rightarrow}\ h = f\ o{\rightarrow}\ (\lambda x.\ g\ x\ o{\rightarrow}\ h)$
  by (*simp add*: *expand-fun-eq split-twice scomp-def*)

lemma *scomp-fcomp*: $(f\ o{\rightarrow}\ g)\ o{>}\ h = f\ o{\rightarrow}\ (\lambda x.\ g\ x\ o{>}\ h)$
  by (*simp add*: *expand-fun-eq scomp-apply fcomp-def split-def*)

lemma *fcomp-scomp*: $(f\ o{>}\ g)\ o{\rightarrow}\ h = f\ o{>}\ (g\ o{\rightarrow}\ h)$
  by (*simp add*: *expand-fun-eq scomp-apply fcomp-apply*)

no-notation *fcomp* (**infixl** $o{>}$ *60*)
no-notation *scomp* (**infixl** $o{\rightarrow}$ *60*)

prod-fun — action of the product functor upon functions.

definition *prod-fun* :: $('a \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'd) \Rightarrow 'a \times 'b \Rightarrow 'c \times 'd$ **where**
  [*code del*]: $prod\text{-}fun\ f\ g = (\lambda(x,\ y).\ (f\ x,\ g\ y))$

lemma *prod-fun* [*simp*, *code*]: $prod\text{-}fun\ f\ g\ (a,\ b) = (f\ a,\ g\ b)$
  by (*simp add*: *prod-fun-def*)

lemma *prod-fun-compose*: *prod-fun* (*f1 o f2*) (*g1 o g2*) = (*prod-fun f1 g1 o prod-fun f2 g2*)
  by (*rule ext*) *auto*

lemma *prod-fun-ident* [*simp*]: *prod-fun* (%*x*. *x*) (%*y*. *y*) = (%*z*. *z*)
  by (*rule ext*) *auto*

lemma *prod-fun-imageI* [*intro*]: $(a,\ b) : r \Longrightarrow (f\ a,\ g\ b) : prod\text{-}fun\ f\ g\ `\ r$
  apply (*rule image-eqI*)
  apply (*rule prod-fun* [*symmetric*], *assumption*)
  done

lemma *prod-fun-imageE* [*elim*!]:
  assumes *major*: *c*: (*prod-fun f g*)‘*r*
    and *cases*: !!*x y*. [| *c*=(*f*(*x*),*g*(*y*)); (*x*,*y*):*r* |] ==> *P*
  shows *P*
  apply (*rule major* [*THEN imageE*])
  apply (*rule-tac p* = *x* **in** *PairE*)

apply (*rule cases*)
apply (*blast intro*: *prod-fun*)
apply *blast*
done

definition
apfst :: ('*a* ⇒ '*c*) ⇒ '*a* × '*b* ⇒ '*c* × '*b*
where
[*code del*]: *apfst f* = *prod-fun f id*

definition
apsnd :: ('*b* ⇒ '*c*) ⇒ '*a* × '*b* ⇒ '*a* × '*c*
where
[*code del*]: *apsnd f* = *prod-fun id f*

lemma *apfst-conv* [*simp*, *code*]:
apfst f (*x*, *y*) = (*f x*, *y*)
by (*simp add*: *apfst-def*)

lemma *upd-snd-conv* [*simp*, *code*]:
apsnd f (*x*, *y*) = (*x*, *f y*)
by (*simp add*: *apsnd-def*)

Disjoint union of a family of sets – Sigma.

definition *Sigma* :: ['*a set*, '*a* => '*b set*] => ('*a* × '*b*) *set* **where**
Sigma-def: *Sigma A B* == *UN x*:*A*. *UN y*:*B x*. {*Pair x y*}

abbreviation
Times :: ['*a set*, '*b set*] => ('*a* * '*b*) *set*
(**infixr** <*> *80*) **where**
A <*> *B* == *Sigma A* (%-. *B*)

notation (*xsymbols*)
Times (**infixr** × *80*)

notation (*HTML* **output**)
Times (**infixr** × *80*)

syntax
@*Sigma* ::[*pttrn*, '*a set*, '*b set*] => ('*a* * '*b*) *set* ((*3SIGMA* -:-./ -) [*0*, *0*, *10*] *10*)

translations
SIGMA x:*A*. *B* == *Product-Type.Sigma A* (%*x*. *B*)

lemma *SigmaI* [*intro*!]: [| *a*:*A*; *b*:*B*(*a*) |] ==> (*a*,*b*) : *Sigma A B*
by (*unfold Sigma-def*) *blast*

lemma *SigmaE* [*elim*!]:
[| *c*: *Sigma A B*;

!!x y.[| x:A; y:$B(x)$; c=(x,y) |] ==> P
|] ==> P
— The general elimination rule.
by (*unfold Sigma-def*) *blast*

Elimination of $(a,\ b) \in A \times B$ – introduces no eigenvariables.

lemma *SigmaD1*: $(a,\ b)$: *Sigma A B* ==> a : A
by *blast*

lemma *SigmaD2*: $(a,\ b)$: *Sigma A B* ==> b : B a
by *blast*

lemma *SigmaE2*:
[| $(a,\ b)$: *Sigma A B*;
[| a:A; b:$B(a)$ |] ==> P
|] ==> P
by *blast*

lemma *Sigma-cong*:
⟦$A = B$; !!x. $x \in B \Longrightarrow C\ x = D\ x$⟧
$\Longrightarrow$ (*SIGMA* x: A. $C\ x$) = (*SIGMA* x: B. $D\ x$)
by *auto*

lemma *Sigma-mono*: [| A <= C; !!x. x:A ==> $B\ x$ <= $D\ x$ |] ==> *Sigma A B* <= *Sigma C D*
by *blast*

lemma *Sigma-empty1* [*simp*]: *Sigma* {} B = {}
by *blast*

lemma *Sigma-empty2* [*simp*]: A <*> {} = {}
by *blast*

lemma *UNIV-Times-UNIV* [*simp*]: *UNIV* <*> *UNIV* = *UNIV*
by *auto*

lemma *Compl-Times-UNIV1* [*simp*]: − (*UNIV* <*> A) = *UNIV* <*> (−A)
by *auto*

lemma *Compl-Times-UNIV2* [*simp*]: − (A <*> *UNIV*) = (−A) <*> *UNIV*
by *auto*

lemma *mem-Sigma-iff* [*iff*]: ((*a*,*b*): *Sigma A B*) = (*a*:*A* & *b*:$B(a)$)
by *blast*

lemma *Times-subset-cancel2*: x:C ==> (A <*> C <= B <*> C) = (A <= B)
by *blast*

lemma *Times-eq-cancel2*: x:C ==> (A <*> C = B <*> C) = (A = B)

by (*blast elim*: *equalityE*)

lemma *SetCompr-Sigma-eq*:
Collect (*split* (%*x y*. *P x* & *Q x y*)) = (*SIGMA x*:*Collect P*. *Collect* (*Q x*))
by *blast*

lemma *Collect-split* [*simp*]: {(*a*,*b*). *P a* & *Q b*} = *Collect P* <*> *Collect Q*
by *blast*

lemma *UN-Times-distrib*:
(*UN* (*a*,*b*):(*A* <*> *B*). *E a* <*> *F b*) = (*UNION A E*) <*> (*UNION B F*)
— Suggested by Pierre Chartier
by *blast*

lemma *split-paired-Ball-Sigma* [*simp*,*noatp*]:
(*ALL z*: *Sigma A B*. *P z*) = (*ALL x*:*A*. *ALL y*: *B x*. *P*(*x*,*y*))
by *blast*

lemma *split-paired-Bex-Sigma* [*simp*,*noatp*]:
(*EX z*: *Sigma A B*. *P z*) = (*EX x*:*A*. *EX y*: *B x*. *P*(*x*,*y*))
by *blast*

lemma *Sigma-Un-distrib1*: (*SIGMA i*:*I Un J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) *Un* (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Un-distrib2*: (*SIGMA i*:*I*. *A*(*i*) *Un B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) *Un* (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Int-distrib1*: (*SIGMA i*:*I Int J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) *Int* (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Int-distrib2*: (*SIGMA i*:*I*. *A*(*i*) *Int B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) *Int* (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Diff-distrib1*: (*SIGMA i*:*I* − *J*. *C*(*i*)) = (*SIGMA i*:*I*. *C*(*i*)) − (*SIGMA j*:*J*. *C*(*j*))
by *blast*

lemma *Sigma-Diff-distrib2*: (*SIGMA i*:*I*. *A*(*i*) − *B*(*i*)) = (*SIGMA i*:*I*. *A*(*i*)) − (*SIGMA i*:*I*. *B*(*i*))
by *blast*

lemma *Sigma-Union*: *Sigma* (*Union X*) *B* = (*UN A*:*X*. *Sigma A B*)
by *blast*

Non-dependent versions are needed to avoid the need for higher-order match-

ing, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: $(A\ Un\ B) <*> C = (A <*> C)\ Un\ (B <*> C)$
by *blast*

lemma *Times-Int-distrib1*: $(A\ Int\ B) <*> C = (A <*> C)\ Int\ (B <*> C)$
by *blast*

lemma *Times-Diff-distrib1*: $(A - B) <*> C = (A <*> C) - (B <*> C)$
by *blast*

lemma *insert-times-insert*[*simp*]:
$insert\ a\ A \times insert\ b\ B =$
$insert\ (a,b)\ (A \times insert\ b\ B \cup insert\ a\ A \times B)$
by *blast*

13.4.2 Code generator setup

instance $*$ :: $(eq,\ eq)\ eq$ **..**

lemma [*code*]:
eq-class.eq $(x1::'a::eq,\ y1::'b::eq)\ (x2,\ y2) \longleftrightarrow x1 = x2 \wedge y1 = y2$ **by** (*simp add*: *eq*)

lemma *split-case-cert*:
assumes $CASE \equiv split\ f$
shows $CASE\ (a,\ b) \equiv f\ a\ b$
using *assms* **by** *simp*

setup ⟪
Code.add-case @{*thm split-case-cert*}
⟫

code-type $*$
(*SML* **infix** *2* $*$)
(*OCaml* **infix** *2* $*$)
(*Haskell* !((-),/ (-)))

code-instance $*$ :: *eq*
(*Haskell* −)

code-const *eq-class.eq* :: $'a::eq \times 'b::eq \Rightarrow 'a \times 'b \Rightarrow bool$
(*Haskell* **infixl** *4* ==)

code-const *Pair*
(*SML* !((-),/ (-)))
(*OCaml* !((-),/ (-)))
(*Haskell* !((-),/ (-)))

code-const *fst* **and** *snd*

(*Haskell fst* **and** *snd*)

types-code
  *** ((- **/ -))
attach (*term-of*) ⟪
fun term-of-id-42 aF aT bF bT (*x*, *y*) = *HOLogic.pair-const aT bT* $ *aF x* $ *bF y*;
⟫
attach (*test*) ⟪
fun gen-id-42 aG aT bG bT i =
  let
    val (*x*, *t*) = *aG i*;
    val (*y*, *u*) = *bG i*
  in ((*x*, *y*), *fn* () => *HOLogic.pair-const aT bT* $ *t* () $ *u* ()) *end*;
⟫

consts-code
  Pair ((-,/ -))

setup ⟪

let

fun strip-abs-split 0 t = ([], *t*)
  | *strip-abs-split i* (*Abs* (*s*, *T*, *t*)) =
      let
        val s' = *Codegen.new-name t s*;
        val v = *Free* (*s'*, *T*)
      in apfst (*cons v*) (*strip-abs-split* (*i*−*1*) (*subst-bound* (*v*, *t*))) *end*
  | *strip-abs-split i* (*u as Const* (*split*, -) $ *t*) = (*case strip-abs-split* (*i*+*1*) *t of*
        (*v* :: *v'* :: *vs*, *u*) => (*HOLogic.mk-prod* (*v*, *v'*) :: *vs*, *u*)
      | - => ([], *u*))
  | *strip-abs-split i t* =
      strip-abs-split i (*Abs* (*x*, *hd* (*binder-types* (*fastype-of t*)), *t* $ *Bound 0*));

fun let-codegen thy defs dep thyname brack t gr = (*case strip-comb t of*
    (*t1 as Const* (*Let*, -), *t2* :: *t3* :: *ts*) =>
    let
      fun dest-let (*l as Const* (*Let*, -) $ *t* $ *u*) =
          (*case strip-abs-split 1 u of*
            ([*p*], *u'*) => *apfst* (*cons* (*p*, *t*)) (*dest-let u'*)
          | - => ([], *l*))
        | *dest-let t* = ([], *t*);
      fun mk-code (*l*, *r*) *gr* =
        let
          val (*pl*, *gr1*) = *Codegen.invoke-codegen thy defs dep thyname false l gr*;
          val (*pr*, *gr2*) = *Codegen.invoke-codegen thy defs dep thyname false r gr1*;
        in ((*pl*, *pr*), *gr2*) *end*
    in case dest-let (*t1* $ *t2* $ *t3*) *of*

```
      ([], -) => NONE
    | (ps, u) =>
        let
          val (qs, gr1) = fold-map mk-code ps gr;
         val (pu, gr2) = Codegen.invoke-codegen thy defs dep thyname false u gr1;
          val (pargs, gr3) = fold-map
            (Codegen.invoke-codegen thy defs dep thyname true) ts gr2
        in
          SOME (Codegen.mk-app brack
            (Pretty.blk (0, [Codegen.str let , Pretty.blk (0, List.concat
                (separate [Codegen.str ;, Pretty.brk 1] (map (fn (pl, pr) =>
                  [Pretty.block [Codegen.str val , pl, Codegen.str =,
                    Pretty.brk 1, pr]]) qs))),
              Pretty.brk 1, Codegen.str in , pu,
              Pretty.brk 1, Codegen.str end])) pargs, gr3)
        end
  end
| - => NONE);

fun split-codegen thy defs dep thyname brack t gr = (case strip-comb t of
    (t1 as Const (split, -), t2 :: ts) =>
      let
        val ([p], u) = strip-abs-split 1 (t1 $ t2);
        val (q, gr1) = Codegen.invoke-codegen thy defs dep thyname false p gr;
        val (pu, gr2) = Codegen.invoke-codegen thy defs dep thyname false u gr1;
        val (pargs, gr3) = fold-map
          (Codegen.invoke-codegen thy defs dep thyname true) ts gr2
      in
        SOME (Codegen.mk-app brack
          (Pretty.block [Codegen.str (fn , q, Codegen.str =>,
            Pretty.brk 1, pu, Codegen.str )]) pargs, gr2)
      end
  | - => NONE);

in

  Codegen.add-codegen let-codegen let-codegen
  #> Codegen.add-codegen split-codegen split-codegen

end
⟩⟩
```

13.5 Legacy bindings

ML ⟨⟨

```
val Collect-split = thm Collect-split;
val Compl-Times-UNIV1 = thm Compl-Times-UNIV1;
val Compl-Times-UNIV2 = thm Compl-Times-UNIV2;
val PairE = thm PairE;
```

val Pair-Rep-inject = thm Pair-Rep-inject;
val Pair-def = thm Pair-def;
val Pair-eq = @{*thm prod.inject*};
val Pair-fst-snd-eq = thm Pair-fst-snd-eq;
val ProdI = thm ProdI;
val SetCompr-Sigma-eq = thm SetCompr-Sigma-eq;
val SigmaD1 = thm SigmaD1;
val SigmaD2 = thm SigmaD2;
val SigmaE = thm SigmaE;
val SigmaE2 = thm SigmaE2;
val SigmaI = thm SigmaI;
val Sigma-Diff-distrib1 = thm Sigma-Diff-distrib1;
val Sigma-Diff-distrib2 = thm Sigma-Diff-distrib2;
val Sigma-Int-distrib1 = thm Sigma-Int-distrib1;
val Sigma-Int-distrib2 = thm Sigma-Int-distrib2;
val Sigma-Un-distrib1 = thm Sigma-Un-distrib1;
val Sigma-Un-distrib2 = thm Sigma-Un-distrib2;
val Sigma-Union = thm Sigma-Union;
val Sigma-def = thm Sigma-def;
val Sigma-empty1 = thm Sigma-empty1;
val Sigma-empty2 = thm Sigma-empty2;
val Sigma-mono = thm Sigma-mono;
val The-split = thm The-split;
val The-split-eq = thm The-split-eq;
val The-split-eq = thm The-split-eq;
val Times-Diff-distrib1 = thm Times-Diff-distrib1;
val Times-Int-distrib1 = thm Times-Int-distrib1;
val Times-Un-distrib1 = thm Times-Un-distrib1;
val Times-eq-cancel2 = thm Times-eq-cancel2;
val Times-subset-cancel2 = thm Times-subset-cancel2;
val UNIV-Times-UNIV = thm UNIV-Times-UNIV;
val UN-Times-distrib = thm UN-Times-distrib;
val Unity-def = thm Unity-def;
val cond-split-eta = thm cond-split-eta;
val fst-conv = thm fst-conv;
val fst-def = thm fst-def;
val fst-eqD = thm fst-eqD;
val inj-on-Abs-Prod = thm inj-on-Abs-Prod;
val mem-Sigma-iff = thm mem-Sigma-iff;
val mem-splitE = thm mem-splitE;
val mem-splitI = thm mem-splitI;
val mem-splitI2 = thm mem-splitI2;
val prod-eqI = thm prod-eqI;
val prod-fun = thm prod-fun;
val prod-fun-compose = thm prod-fun-compose;
val prod-fun-def = thm prod-fun-def;
val prod-fun-ident = thm prod-fun-ident;
val prod-fun-imageE = thm prod-fun-imageE;
val prod-fun-imageI = thm prod-fun-imageI;

val prod-induct = thm prod.induct;
val snd-conv = thm snd-conv;
val snd-def = thm snd-def;
val snd-eqD = thm snd-eqD;
val split = thm split;
val splitD = thm splitD;
val splitD' = thm splitD';
val splitE = thm splitE;
val splitE' = thm splitE';
val splitE2 = thm splitE2;
val splitI = thm splitI;
val splitI2 = thm splitI2;
val splitI2' = thm splitI2';
val split-beta = thm split-beta;
val split-conv = thm split-conv;
val split-def = thm split-def;
val split-eta = thm split-eta;
val split-eta-SetCompr = thm split-eta-SetCompr;
val split-eta-SetCompr2 = thm split-eta-SetCompr2;
val split-paired-All = thm split-paired-All;
val split-paired-Ball-Sigma = thm split-paired-Ball-Sigma;
val split-paired-Bex-Sigma = thm split-paired-Bex-Sigma;
val split-paired-Ex = thm split-paired-Ex;
val split-paired-The = thm split-paired-The;
val split-paired-all = thm split-paired-all;
val split-part = thm split-part;
val split-split = thm split-split;
val split-split-asm = thm split-split-asm;
val split-tupled-all = thms split-tupled-all;
val split-weak-cong = thm split-weak-cong;
val surj-pair = thm surj-pair;
val surjective-pairing = thm surjective-pairing;
val unit-abs-eta-conv = thm unit-abs-eta-conv;
val unit-all-eq1 = thm unit-all-eq1;
val unit-all-eq2 = thm unit-all-eq2;
val unit-eq = thm unit-eq;
⟩⟩

13.6 Further inductive packages

use *Tools/inductive-realizer.ML*
setup *InductiveRealizer.setup*

use *Tools/inductive-set-package.ML*
setup *InductiveSetPackage.setup*

use *Tools/datatype-realizer.ML*
setup *DatatypeRealizer.setup*

end

14 Datatype: Analogues of the Cartesian Product and Disjoint Sum for Datatypes

theory *Datatype*
imports *Nat Product-Type*
begin

typedef (*Node*)
(*'a*,*'b*) *node* = {*p*. *EX f x k*. *p* = (*f*::*nat*=>*'b*+*nat*, *x*::*'a*+*nat*) & *f k* = *Inr 0*}
— it is a subtype of (*nat*=>*'b*+*nat*) * (*'a*+*nat*)
by *auto*

Datatypes will be represented by sets of type *node*

types *'a item* = (*'a*, *unit*) *node set*
(*'a*, *'b*) *dtree* = (*'a*, *'b*) *node set*

consts
Push :: [(*'b* + *nat*), *nat* => (*'b* + *nat*)] => (*nat* => (*'b* + *nat*))

Push-Node :: [(*'b* + *nat*), (*'a*, *'b*) *node*] => (*'a*, *'b*) *node*
ndepth :: (*'a*, *'b*) *node* => *nat*

Atom :: (*'a* + *nat*) => (*'a*, *'b*) *dtree*
Leaf :: *'a* => (*'a*, *'b*) *dtree*
Numb :: *nat* => (*'a*, *'b*) *dtree*
Scons :: [(*'a*, *'b*) *dtree*, (*'a*, *'b*) *dtree*] => (*'a*, *'b*) *dtree*
In0 :: (*'a*, *'b*) *dtree* => (*'a*, *'b*) *dtree*
In1 :: (*'a*, *'b*) *dtree* => (*'a*, *'b*) *dtree*
Lim :: (*'b* => (*'a*, *'b*) *dtree*) => (*'a*, *'b*) *dtree*

ntrunc :: [*nat*, (*'a*, *'b*) *dtree*] => (*'a*, *'b*) *dtree*

uprod :: [(*'a*, *'b*) *dtree set*, (*'a*, *'b*) *dtree set*]=> (*'a*, *'b*) *dtree set*
usum :: [(*'a*, *'b*) *dtree set*, (*'a*, *'b*) *dtree set*]=> (*'a*, *'b*) *dtree set*

Split :: [[(*'a*, *'b*) *dtree*, (*'a*, *'b*) *dtree*]=>*'c*, (*'a*, *'b*) *dtree*] => *'c*
Case :: [[(*'a*, *'b*) *dtree*]=>*'c*, [(*'a*, *'b*) *dtree*]=>*'c*, (*'a*, *'b*) *dtree*] => *'c*

dprod :: [((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*, ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*]
=> ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*
dsum :: [((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*, ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*]
=> ((*'a*, *'b*) *dtree* * (*'a*, *'b*) *dtree*)*set*

defs

Push-Node-def: *Push-Node* == (%*n x*. *Abs-Node* (*apfst* (*Push n*) (*Rep-Node x*)))

Push-def: *Push* == (%*b h*. *nat-case b h*)

Atom-def: *Atom* == (%*x*. {*Abs-Node*((%*k*. *Inr 0*, *x*))})
Scons-def: *Scons M N* == (*Push-Node* (*Inr 1*) ‘ *M*) *Un* (*Push-Node* (*Inr* (*Suc 1*)) ‘ *N*)

Leaf-def: *Leaf* == *Atom o Inl*
Numb-def: *Numb* == *Atom o Inr*

In0-def: *In0*(*M*) == *Scons* (*Numb 0*) *M*
In1-def: *In1*(*M*) == *Scons* (*Numb 1*) *M*

Lim-def: *Lim f* == *Union* {*z*. *? x*. *z* = *Push-Node* (*Inl x*) ‘ (*f x*)}

ndepth-def: *ndepth*(*n*) == (%(*f*,*x*). *LEAST k*. *f k* = *Inr 0*) (*Rep-Node n*)
ntrunc-def: *ntrunc k N* == {*n*. *n*:*N* & *ndepth*(*n*)<*k*}

uprod-def: *uprod A B* == *UN x*:*A*. *UN y*:*B*. { *Scons x y* }
usum-def: *usum A B* == *In0*‘*A Un In1*‘*B*

Split-def: *Split c M* == *THE u*. *EX x y*. *M* = *Scons x y* & *u* = *c x y*

Case-def: *Case c d M* == *THE u*. (*EX x* . *M* = *In0*(*x*) & *u* = *c*(*x*))
| (*EX y* . *M* = *In1*(*y*) & *u* = *d*(*y*))

dprod-def: *dprod r s* == *UN* (*x*,*x*′):*r*. *UN* (*y*,*y*′):*s*. {(*Scons x y*, *Scons x*′ *y*′)}

dsum-def: *dsum r s* == (*UN* (*x*,*x*′):*r*. {(*In0*(*x*),*In0*(*x*′))}) *Un*
(*UN* (*y*,*y*′):*s*. {(*In1*(*y*),*In1*(*y*′))})

lemma *apfst-convE*:
[| *q* = *apfst f p*; !!*x y*. [| *p* = (*x*,*y*); *q* = (*f*(*x*),*y*) |] ==> *R* |] ==> *R*
by (*force simp add*: *apfst-def*)

lemma *Push-inject1*: *Push i f* = *Push j g* ==> *i*=*j*
apply (*simp add*: *Push-def expand-fun-eq*)
apply (*drule-tac x*=*0* **in** *spec*, *simp*)
done

lemma *Push-inject2*: *Push i f* = *Push j g* ==> *f*=*g*
apply (*auto simp add*: *Push-def expand-fun-eq*)
apply (*drule-tac x*=*Suc x* **in** *spec*, *simp*)
done

lemma *Push-inject*:
[| *Push i f* =*Push j g*; [| *i*=*j*; *f*=*g* |] ==> *P* |] ==> *P*
by (*blast dest*: *Push-inject1 Push-inject2*)

lemma *Push-neq-K0*: *Push* (*Inr* (*Suc k*)) *f* = (%*z*. *Inr 0*) ==> *P*
by (*auto simp add*: *Push-def expand-fun-eq split*: *nat.split-asm*)

lemmas *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*, *standard*]

lemma *Node-K0-I*: (%*k*. *Inr 0*, *a*) : *Node*
by (*simp add*: *Node-def*)

lemma *Node-Push-I*: *p*: *Node* ==> *apfst* (*Push i*) *p* : *Node*
apply (*simp add*: *Node-def Push-def*)
apply (*fast intro*!: *apfst-conv nat-case-Suc* [*THEN trans*])
done

14.1 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N* ≠ *Atom*(*a*)
apply (*simp add*: *Atom-def Scons-def Push-Node-def One-nat-def*)
apply (*blast intro*: *Node-K0-I Rep-Node* [*THEN Node-Push-I*]
dest!: *Abs-Node-inj*
elim!: *apfst-convE sym* [*THEN Push-neq-K0*])
done

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*, *standard*]

lemma *inj-Atom*: *inj*(*Atom*)
apply (*simp add*: *Atom-def*)
apply (*blast intro*!: *inj-onI Node-K0-I dest*!: *Abs-Node-inj*)
done
lemmas *Atom-inject* = *inj-Atom* [*THEN injD*, *standard*]

lemma *Atom-Atom-eq* [*iff*]: (*Atom*(*a*)=*Atom*(*b*)) = (*a*=*b*)
by (*blast dest*!: *Atom-inject*)

lemma *inj-Leaf*: *inj*(*Leaf*)
apply (*simp add*: *Leaf-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inl-inject*])
done

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*, *standard*]

lemma *inj-Numb*: *inj*(*Numb*)
apply (*simp add*: *Numb-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inr-inject*])
done

lemmas *Numb-inject* [*dest*!] = *inj-Numb* [*THEN injD*, *standard*]

lemma *Push-Node-inject*:
[| *Push-Node i m* =*Push-Node j n*; [| *i*=*j*; *m*=*n* |] ==> *P*
|] ==> *P*
apply (*simp add*: *Push-Node-def*)
apply (*erule Abs-Node-inj* [*THEN apfst-convE*])
apply (*rule Rep-Node* [*THEN Node-Push-I*])+
apply (*erule sym* [*THEN apfst-convE*])
apply (*blast intro*: *Rep-Node-inject* [*THEN iffD1*] *trans sym elim*!: *Push-inject*)
done

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M' N'* ==> *M*<=*M'*
apply (*simp add*: *Scons-def One-nat-def*)
apply (*blast dest*!: *Push-Node-inject*)
done

lemma *Scons-inject-lemma2*: *Scons M N <= Scons M' N' ==> N<=N'*
apply (*simp add*: *Scons-def One-nat-def*)
apply (*blast dest*!: *Push-Node-inject*)
done

lemma *Scons-inject1*: *Scons M N = Scons M' N' ==> M=M'*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma1*)
done

lemma *Scons-inject2*: *Scons M N = Scons M' N' ==> N=N'*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma2*)
done

lemma *Scons-inject*:
[| *Scons M N = Scons M' N'*; [| *M=M'*; *N=N'* |] ==> *P* |] ==> *P*
by (*iprover dest*: *Scons-inject1 Scons-inject2*)

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N = Scons M' N'*) = (*M=M' & N=N'*)
by (*blast elim*!: *Scons-inject*)

lemma *Scons-not-Leaf* [*iff*]: *Scons M N ≠ Leaf(a)*
by (*simp add*: *Leaf-def o-def Scons-not-Atom*)

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*, *standard*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N ≠ Numb(k)*
by (*simp add*: *Numb-def o-def Scons-not-Atom*)

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*, *standard*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf(a) ≠ Numb(k)*
by (*simp add*: *Leaf-def Numb-def*)

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*, *standard*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr* 0, *x*)) = 0
by (*simp add*: *ndepth-def* *Node-K0-I* [*THEN Abs-Node-inverse*] *Least-equality*)

lemma *ndepth-Push-Node-aux*:
    nat-case (*Inr* (*Suc i*)) *f k* = *Inr* 0 −−> *Suc*(*LEAST x*. *f x* = *Inr* 0) <= *k*
apply (*induct-tac k*, *auto*)
apply (*erule Least-le*)
done

lemma *ndepth-Push-Node*:
   ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
apply (*insert Rep-Node* [*of n*, *unfolded Node-def*])
apply (*auto simp add*: *ndepth-def Push-Node-def*
            Rep-Node [*THEN Node-Push-I*, *THEN Abs-Node-inverse*])
apply (*rule Least-equality*)
apply (*auto simp add*: *Push-def ndepth-Push-Node-aux*)
apply (*erule LeastI*)
done

lemma *ntrunc-0* [*simp*]: *ntrunc 0 M* = {}
by (*simp add*: *ntrunc-def*)

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
by (*auto simp add*: *Atom-def ntrunc-def ndepth-K0*)

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
by (*simp add*: *Leaf-def o-def ntrunc-Atom*)

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
by (*simp add*: *Numb-def o-def ntrunc-Atom*)

lemma *ntrunc-Scons* [*simp*]:
   ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)
by (*auto simp add*: *Scons-def ntrunc-def One-nat-def ndepth-Push-Node*)

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
apply (*simp add*: *In0-def*)
apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*)

M)
by (*simp add*: *In0-def*)

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
apply (*simp add*: *In1-def*)
apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)
by (*simp add*: *In1-def*)

14.2 Set Constructions

lemma *uprodI* [*intro*!]: [| *M*:*A*; *N*:*B* |] ==> *Scons M N* : *uprod A B*
by (*simp add*: *uprod-def*)

lemma *uprodE* [*elim*!]:
[| *c* : *uprod A B*;
!!*x y*. [| *x*:*A*; *y*:*B*; *c* = *Scons x y* |] ==> *P*
|] ==> *P*
by (*auto simp add*: *uprod-def*)

lemma *uprodE2*: [| *Scons M N* : *uprod A B*; [| *M*:*A*; *N*:*B* |] ==> *P* |] ==> *P*
by (*auto simp add*: *uprod-def*)

lemma *usum-In0I* [*intro*]: *M*:*A* ==> *In0*(*M*) : *usum A B*
by (*simp add*: *usum-def*)

lemma *usum-In1I* [*intro*]: *N*:*B* ==> *In1*(*N*) : *usum A B*
by (*simp add*: *usum-def*)

lemma *usumE* [*elim*!]:
[| *u* : *usum A B*;
!!*x*. [| *x*:*A*; *u*=*In0*(*x*) |] ==> *P*;
!!*y*. [| *y*:*B*; *u*=*In1*(*y*) |] ==> *P*
|] ==> *P*
by (*auto simp add*: *usum-def*)

lemma *In0-not-In1* [*iff*]: *In0*(*M*) ≠ *In1*(*N*)

by (*auto simp add*: *In0-def In1-def One-nat-def*)

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*, *standard*]

lemma *In0-inject*: *In0*(*M*) = *In0*(*N*) ==> *M*=*N*
by (*simp add*: *In0-def*)

lemma *In1-inject*: *In1*(*M*) = *In1*(*N*) ==> *M*=*N*
by (*simp add*: *In1-def*)

lemma *In0-eq* [*iff*]: (*In0 M* = *In0 N*) = (*M*=*N*)
by (*blast dest*!: *In0-inject*)

lemma *In1-eq* [*iff*]: (*In1 M* = *In1 N*) = (*M*=*N*)
by (*blast dest*!: *In1-inject*)

lemma *inj-In0*: *inj In0*
by (*blast intro*!: *inj-onI*)

lemma *inj-In1*: *inj In1*
by (*blast intro*!: *inj-onI*)

lemma *Lim-inject*: *Lim f* = *Lim g* ==> *f* = *g*
apply (*simp add*: *Lim-def*)
apply (*rule ext*)
apply (*blast elim*!: *Push-Node-inject*)
done

lemma *ntrunc-subsetI*: *ntrunc k M* <= *M*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-subsetD*: (!!*k*. *ntrunc k M* <= *N*) ==> *M*<=*N*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-equality*: (!!*k*. *ntrunc k M* = *ntrunc k N*) ==> *M*=*N*
apply (*rule equalityI*)
apply (*rule-tac* [!] *ntrunc-subsetD*)
apply (*rule-tac* [!] *ntrunc-subsetI* [*THEN* [*2*] *subset-trans*], *auto*)
done

lemma *ntrunc-o-equality*:
[| !!*k*. (*ntrunc*(*k*) *o h1*) = (*ntrunc*(*k*) *o h2*) |] ==> *h1*=*h2*

apply (*rule ntrunc-equality* [*THEN ext*])
apply (*simp add*: *expand-fun-eq*)
done

lemma *uprod-mono*: [| *A*<=*A'*; *B*<=*B'* |] ==> *uprod A B* <= *uprod A' B'*
by (*simp add*: *uprod-def*, *blast*)

lemma *usum-mono*: [| *A*<=*A'*; *B*<=*B'* |] ==> *usum A B* <= *usum A' B'*
by (*simp add*: *usum-def*, *blast*)

lemma *Scons-mono*: [| *M*<=*M'*; *N*<=*N'* |] ==> *Scons M N* <= *Scons M' N'*
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-mono*: *M*<=*N* ==> *In0*(*M*) <= *In0*(*N*)
by (*simp add*: *In0-def subset-refl Scons-mono*)

lemma *In1-mono*: *M*<=*N* ==> *In1*(*M*) <= *In1*(*N*)
by (*simp add*: *In1-def subset-refl Scons-mono*)

lemma *Split* [*simp*]: *Split c* (*Scons M N*) = *c M N*
by (*simp add*: *Split-def*)

lemma *Case-In0* [*simp*]: *Case c d* (*In0 M*) = *c*(*M*)
by (*simp add*: *Case-def*)

lemma *Case-In1* [*simp*]: *Case c d* (*In1 N*) = *d*(*N*)
by (*simp add*: *Case-def*)

lemma *ntrunc-UN1*: *ntrunc k* (*UN x*. *f*(*x*)) = (*UN x*. *ntrunc k* (*f x*))
by (*simp add*: *ntrunc-def*, *blast*)

lemma *Scons-UN1-x*: *Scons* (*UN x*. *f x*) *M* = (*UN x*. *Scons* (*f x*) *M*)
by (*simp add*: *Scons-def*, *blast*)

lemma *Scons-UN1-y*: *Scons M* (*UN x*. *f x*) = (*UN x*. *Scons M* (*f x*))
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-UN1*: *In0*(*UN x*. *f*(*x*)) = (*UN x*. *In0*(*f*(*x*)))
by (*simp add*: *In0-def Scons-UN1-y*)

lemma *In1-UN1*: $In1(UN\ x.\ f(x)) = (UN\ x.\ In1(f(x)))$
by (*simp add*: *In1-def Scons-UN1-y*)

lemma *dprodI* [*intro*!]:
$[|\ (M,M'):r;\ \ (N,N'):s\ |] ==> (Scons\ M\ N,\ Scons\ M'\ N') : dprod\ r\ s$
by (*auto simp add*: *dprod-def*)

lemma *dprodE* [*elim*!]:
$[|\ c : dprod\ r\ s;$
$\quad !!x\ y\ x'\ y'.\ [|\ (x,x') : r;\ \ (y,y') : s;$
$\qquad\qquad c = (Scons\ x\ y,\ Scons\ x'\ y')\ |] ==> P$
$\ |] ==> P$
by (*auto simp add*: *dprod-def*)

lemma *dsum-In0I* [*intro*]: $(M,M'):r ==> (In0(M),\ In0(M')) : dsum\ r\ s$
by (*auto simp add*: *dsum-def*)

lemma *dsum-In1I* [*intro*]: $(N,N'):s ==> (In1(N),\ In1(N')) : dsum\ r\ s$
by (*auto simp add*: *dsum-def*)

lemma *dsumE* [*elim*!]:
$[|\ w : dsum\ r\ s;$
$\quad !!x\ x'.\ [|\ (x,x') : r;\ \ w = (In0(x),\ In0(x'))\ |] ==> P;$
$\quad !!y\ y'.\ [|\ (y,y') : s;\ \ w = (In1(y),\ In1(y'))\ |] ==> P$
$\ |] ==> P$
by (*auto simp add*: *dsum-def*)

lemma *dprod-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dprod\ r\ s <= dprod\ r'\ s'$
by *blast*

lemma *dsum-mono*: $[|\ r<=r';\ \ s<=s'\ |] ==> dsum\ r\ s <= dsum\ r'\ s'$
by *blast*

lemma *dprod-Sigma*: $(dprod\ (A\ <*>\ B)\ (C\ <*>\ D)) <= (uprod\ A\ C)\ <*>\ (uprod\ B\ D)$

by *blast*

lemmas *dprod-subset-Sigma = subset-trans [OF dprod-mono dprod-Sigma, standard]*

lemma *dprod-subset-Sigma2*:
(*dprod* (*Sigma A B*) (*Sigma C D*)) <=
Sigma (*uprod A C*) (*Split* (%*x y*. *uprod* (*B x*) (*D y*)))
by *auto*

lemma *dsum-Sigma*: (*dsum* (*A* <*>* *B*) (*C* <*>* *D*)) <= (*usum A C*) <*>* (*usum B D*)
by *blast*

lemmas *dsum-subset-Sigma = subset-trans [OF dsum-mono dsum-Sigma, standard]*

hides popular names

hide (**open**) *type node item*
hide (**open**) *const Push Node Atom Leaf Numb Lim Split Case*

15 Datatypes

15.1 Representing sums

rep-datatype (*sum*) *Inl Inr*
proof −
fix *P*
fix *s* :: *'a* + *'b*
assume *x*: ⋀*x*::*'a*. *P* (*Inl x*) **and** *y*: ⋀*y*::*'b*. *P* (*Inr y*)
then show *P s* **by** (*auto intro*: *sumE* [*of s*])
qed *simp-all*

lemma *sum-case-KK*[*simp*]: *sum-case* (%*x*. *a*) (%*x*. *a*) = (%*x*. *a*)
by (*rule ext*) (*simp split*: *sum.split*)

lemma *surjective-sum*: *sum-case* (%*x*::*'a*. *f* (*Inl x*)) (%*y*::*'b*. *f* (*Inr y*)) *s* = *f*(*s*)
apply (*rule-tac s = s* **in** *sumE*)
apply (*erule ssubst*)
apply (*rule sum.cases*(*1*))
apply (*erule ssubst*)
apply (*rule sum.cases*(*2*))
done

lemma *sum-case-weak-cong*: *s = t ==> sum-case f g s = sum-case f g t*
— Prevents simplification of *f* and *g*: much faster.
by *simp*

lemma *sum-case-inject*:
sum-case f1 f2 = sum-case g1 g2 ==> (f1 = g1 ==> f2 = g2 ==> P) ==> P
proof −
assume *a*: *sum-case f1 f2 = sum-case g1 g2*
assume *r*: *f1 = g1 ==> f2 = g2 ==> P*
show *P*
apply (*rule r*)
apply (*rule ext*)
apply (*cut-tac x = Inl x* **in** *a* [*THEN fun-cong*], *simp*)
apply (*rule ext*)
apply (*cut-tac x = Inr x* **in** *a* [*THEN fun-cong*], *simp*)
done
qed

constdefs
Suml :: ('*a* => '*c*) => '*a* + '*b* => '*c*
Suml == (%*f*. *sum-case f undefined*)

Sumr :: ('*b* => '*c*) => '*a* + '*b* => '*c*
Sumr == *sum-case undefined*

lemma *Suml-inject*: *Suml f = Suml g ==> f = g*
by (*unfold Suml-def*) (*erule sum-case-inject*)

lemma *Sumr-inject*: *Sumr f = Sumr g ==> f = g*
by (*unfold Sumr-def*) (*erule sum-case-inject*)

primrec *Projl* :: '*a* + '*b* => '*a*
where *Projl-Inl*: *Projl* (*Inl x*) = *x*

primrec *Projr* :: '*a* + '*b* => '*b*
where *Projr-Inr*: *Projr* (*Inr x*) = *x*

hide (**open**) *const Suml Sumr Projl Projr*

end

16 Power: Exponentiation

theory *Power*
imports *Nat*
begin

class *power* =
fixes *power* :: '*a* ⇒ *nat* ⇒ '*a* (**infixr** ^ *80*)

16.1 Powers for Arbitrary Monoids

class *recpower* = *monoid-mult* + *power* +
  assumes *power-0* [*simp*]: *a ^ 0 = 1*
  assumes *power-Suc* [*simp*]: *a ^ Suc n = a * (a ^ n)*

lemma *power-0-Suc* [*simp*]: *(0::'a::{recpower,semiring-0}) ^ (Suc n) = 0*
  by *simp*

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*: *0^n = (if n=0 then 1 else (0::'a::{recpower,semiring-0}))*
  by (*induct n*) *simp-all*

lemma *power-one* [*simp*]: *1^n = (1::'a::recpower)*
  by (*induct n*) *simp-all*

lemma *power-one-right* [*simp*]: *(a::'a::recpower) ^ 1 = a*
  unfolding *One-nat-def* **by** *simp*

lemma *power-commutes*: *(a::'a::recpower) ^ n * a = a * a ^ n*
  by (*induct n*) (*simp-all add*: *mult-assoc*)

lemma *power-Suc2*: *(a::'a::recpower) ^ Suc n = a ^ n * a*
  by (*simp add*: *power-commutes*)

lemma *power-add*: *(a::'a::recpower) ^ (m+n) = (a^m) * (a^n)*
  by (*induct m*) (*simp-all add*: *mult-ac*)

lemma *power-mult*: *(a::'a::recpower) ^ (m*n) = (a^m) ^ n*
  by (*induct n*) (*simp-all add*: *power-add*)

lemma *power-mult-distrib*: *((a::'a::{recpower,comm-monoid-mult}) * b) ^ n = (a^n) * (b^n)*
  by (*induct n*) (*simp-all add*: *mult-ac*)

lemma *zero-less-power*[*simp*]:
    0 < (a::'a::{ordered-semidom,recpower}) ==> 0 < a^n
by (*induct n*) (*simp-all add*: *mult-pos-pos*)

lemma *zero-le-power*[*simp*]:
    0 ≤ (a::'a::{ordered-semidom,recpower}) ==> 0 ≤ a^n
by (*induct n*) (*simp-all add*: *mult-nonneg-nonneg*)

lemma *one-le-power*[*simp*]:
    1 ≤ (a::'a::{ordered-semidom,recpower}) ==> 1 ≤ a^n
apply (*induct n*)
apply *simp-all*
apply (*rule order-trans* [*OF - mult-mono* [*of 1 - 1*]])
apply (*simp-all add*: *order-trans* [*OF zero-le-one*])
done

lemma *gt1-imp-ge0*: *1 < a ==> 0 ≤* (*a*::*'a*::*ordered-semidom*)
by (*simp add*: *order-trans* [*OF zero-le-one order-less-imp-le*])

lemma *power-gt1-lemma*:
assumes *gt1*: *1 <* (*a*::*'a*::{*ordered-semidom*,*recpower*})
shows *1 < a * a^n*
proof −
have *1*1 < a*1* **using** *gt1* **by** *simp*
also have ... ≤ *a * a^n* **using** *gt1*
by (*simp only*: *mult-mono gt1-imp-ge0 one-le-power order-less-imp-le*
zero-le-one order-refl)
finally show *?thesis* **by** *simp*
qed

lemma *one-less-power*[*simp*]:
⟦*1 <* (*a*::*'a*::{*ordered-semidom*,*recpower*}); *0 < n*⟧ ⟹ *1 < a ^ n*
by (*cases n*, *simp-all add*: *power-gt1-lemma*)

lemma *power-gt1*:
1 < (*a*::*'a*::{*ordered-semidom*,*recpower*}) ==> *1 < a ^* (*Suc n*)
by (*simp add*: *power-gt1-lemma*)

lemma *power-le-imp-le-exp*:
assumes *gt1*: (*1*::*'a*::{*recpower*,*ordered-semidom*}) *< a*
shows !!*n*. *a^m ≤ a^n ==> m ≤ n*
proof (*induct m*)
case *0*
show *?case* **by** *simp*
next
case (*Suc m*)
show *?case*
proof (*cases n*)
case *0*
from *prems* **have** *a * a^m ≤ 1* **by** *simp*
with *gt1* **show** *?thesis*
by (*force simp only*: *power-gt1-lemma*
linorder-not-less [*symmetric*])
next
case (*Suc n*)
from *prems* **show** *?thesis*
by (*force dest*: *mult-left-le-imp-le*
simp add: *order-less-trans* [*OF zero-less-one gt1*])
qed
qed

Surely we can strengthen this? It holds for *0<a<1* too.

lemma *power-inject-exp* [*simp*]:
1 < (*a*::*'a*::{*ordered-semidom*,*recpower*}) ==> (*a^m = a^n*) = (*m=n*)

by (*force simp add*: *order-antisym power-le-imp-le-exp*)

Can relax the first premise to $(0::'a) < a$ in the case of the natural numbers.

lemma *power-less-imp-less-exp*:
$[|\ (1::'a::\{recpower,ordered\text{-}semidom\}) < a;\ a\hat{}m < a\hat{}n\ |] ==> m < n$
by (*simp add*: *order-less-le* [*of m n*] *order-less-le* [*of a^m a^n*]
power-le-imp-le-exp)

lemma *power-mono*:
$[|a \leq b;\ (0::'a::\{recpower,ordered\text{-}semidom\}) \leq a|] ==> a\hat{}n \leq b\hat{}n$
apply (*induct n*)
apply *simp-all*
apply (*auto intro*: *mult-mono order-trans* [*of 0 a b*])
done

lemma *power-strict-mono* [*rule-format*]:
$[|a < b;\ (0::'a::\{recpower,ordered\text{-}semidom\}) \leq a|]$
$==> 0 < n \longrightarrow a\hat{}n < b\hat{}n$
apply (*induct n*)
apply (*auto simp add*: *mult-strict-mono order-le-less-trans* [*of 0 a b*])
done

lemma *power-eq-0-iff* [*simp*]:
$(a\hat{}n = 0) \longleftrightarrow$
$(a = (0::'a::\{mult\text{-}zero,zero\text{-}neq\text{-}one,no\text{-}zero\text{-}divisors,recpower\})\ \&\ n\neq 0)$
apply (*induct n*)
apply (*auto simp add*: *no-zero-divisors*)
done

lemma *field-power-not-zero*:
$a \neq (0::'a::\{ring\text{-}1\text{-}no\text{-}zero\text{-}divisors,recpower\}) ==> a\hat{}n \neq 0$
by *force*

lemma *nonzero-power-inverse*:
fixes $a :: 'a::\{division\text{-}ring,recpower\}$
shows $a \neq 0 ==> inverse\ (a\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
apply (*induct n*)
apply (*auto simp add*: *nonzero-inverse-mult-distrib power-commutes*)
done

Perhaps these should be simprules.

lemma *power-inverse*:
fixes $a :: 'a::\{division\text{-}ring,division\text{-}by\text{-}zero,recpower\}$
shows $inverse\ (a\ \hat{}\ n) = (inverse\ a)\ \hat{}\ n$
apply (*cases a = 0*)
apply (*simp add*: *power-0-left*)
apply (*simp add*: *nonzero-power-inverse*)

done

lemma *power-one-over*: *1 / (a::'a::{field,division-by-zero,recpower}) ^n =*
(1 / a) ^n
apply (*simp add*: *divide-inverse*)
apply (*rule power-inverse*)
done

lemma *nonzero-power-divide*:
b ≠ 0 ==> (a/b) ^ n = ((a::'a::{field,recpower}) ^ n) / (b ^ n)
by (*simp add*: *divide-inverse power-mult-distrib nonzero-power-inverse*)

lemma *power-divide*:
(a/b) ^ n = ((a::'a::{field,division-by-zero,recpower}) ^ n / b ^ n)
apply (*case-tac b=0*, *simp add*: *power-0-left*)
apply (*rule nonzero-power-divide*)
apply *assumption*
done

lemma *power-abs*: *abs(a ^ n) = abs(a::'a::{ordered-idom,recpower}) ^ n*
apply (*induct n*)
apply (*auto simp add*: *abs-mult*)
done

lemma *abs-power-minus* [*simp*]:
fixes *a*:: *'a::{ordered-idom,recpower}* **shows** *abs((−a) ^ n) = abs(a ^ n)*
by (*simp add*: *abs-minus-cancel power-abs*)

lemma *zero-less-power-abs-iff* [*simp*,*noatp*]:
(0 < (abs a)^n) = (a ≠ (0::'a::{ordered-idom,recpower}) | n=0)
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
show *?case* **by** (*auto simp add*: *prems zero-less-mult-iff*)
qed

lemma *zero-le-power-abs* [*simp*]:
(0::'a::{ordered-idom,recpower}) ≤ (abs a)^n
by (*rule zero-le-power* [*OF abs-ge-zero*])

lemma *power-minus*: *(−a) ^ n = (− 1)^n * (a::'a::{ring-1,recpower}) ^ n*
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **then show** *?case*
by (*simp del*: *power-Suc add*: *power-Suc2 mult-assoc*)
qed

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*:
[|(0::'a::{ordered-semidom,recpower}) < a; a < 1|]
==> a * a^n < a^n
apply (*induct n*)
apply (*auto simp add*: *mult-strict-left-mono*)
done

lemma *power-strict-decreasing*:
[|n < N; 0 < a; a < (1::'a::{ordered-semidom,recpower})|]
==> a^N < a^n
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-Suc-less less-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac a * a^m < 1 * a^n, simp*)
apply (*rule mult-strict-mono*)
apply (*auto simp add*: *order-less-imp-le*)
done

Proof resembles that of *power-strict-decreasing*

lemma *power-decreasing*:
[|n ≤ N; 0 ≤ a; a ≤ (1::'a::{ordered-semidom,recpower})|]
==> a^N ≤ a^n
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *le-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac a * a^m ≤ 1 * a^n, simp*)
apply (*rule mult-mono*)
apply *auto*
done

lemma *power-Suc-less-one*:
[| 0 < a; a < (1::'a::{ordered-semidom,recpower}) |] ==> a ^ Suc n < 1
apply (*insert power-strict-decreasing* [*of 0 Suc n a*], *simp*)
done

Proof again resembles that of *power-strict-decreasing*

lemma *power-increasing*:
[|n ≤ N; (1::'a::{ordered-semidom,recpower}) ≤ a|] ==> a^n ≤ a^N
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *le-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac 1 * a^n ≤ a * a^m, simp*)
apply (*rule mult-mono*)
apply (*auto simp add*: *order-trans* [*OF zero-le-one*])
done

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:
(*1*::*'a*::{*ordered-semidom*,*recpower*}) < *a* ==> *a* ^ *n* < *a* * *a* ^ *n*
apply (*induct n*)
apply (*auto simp add*: *mult-strict-left-mono order-less-trans* [*OF zero-less-one*])
done

lemma *power-strict-increasing*:
[|*n* < *N*; (*1*::*'a*::{*ordered-semidom*,*recpower*}) < *a*|] ==> *a* ^ *n* < *a* ^ *N*
apply (*erule rev-mp*)
apply (*induct N*)
apply (*auto simp add*: *power-less-power-Suc less-Suc-eq*)
apply (*rename-tac m*)
apply (*subgoal-tac 1* * *a* ^ *n* < *a* * *a* ^ *m*, *simp*)
apply (*rule mult-strict-mono*)
apply (*auto simp add*: *order-less-trans* [*OF zero-less-one*] *order-less-imp-le*)
done

lemma *power-increasing-iff* [*simp*]:
1 < (*b*::*'a*::{*ordered-semidom*,*recpower*}) ==> (*b* ^ *x* ≤ *b* ^ *y*) = (*x* ≤ *y*)
by (*blast intro*: *power-le-imp-le-exp power-increasing order-less-imp-le*)

lemma *power-strict-increasing-iff* [*simp*]:
1 < (*b*::*'a*::{*ordered-semidom*,*recpower*}) ==> (*b* ^ *x* < *b* ^ *y*) = (*x* < *y*)
by (*blast intro*: *power-less-imp-less-exp power-strict-increasing*)

lemma *power-le-imp-le-base*:
assumes *le*: *a* ^ *Suc n* ≤ *b* ^ *Suc n*
and *ynonneg*: (*0*::*'a*::{*ordered-semidom*,*recpower*}) ≤ *b*
shows *a* ≤ *b*
proof (*rule ccontr*)
assume ~ *a* ≤ *b*
then have *b* < *a* **by** (*simp only*: *linorder-not-le*)
then have *b* ^ *Suc n* < *a* ^ *Suc n*
by (*simp only*: *prems power-strict-mono*)
from *le* **and** *this* **show** *False*
by (*simp add*: *linorder-not-less* [*symmetric*])
qed

lemma *power-less-imp-less-base*:
fixes *a b* :: *'a*::{*ordered-semidom*,*recpower*}
assumes *less*: *a* ^ *n* < *b* ^ *n*
assumes *nonneg*: *0* ≤ *b*
shows *a* < *b*
proof (*rule contrapos-pp* [*OF less*])
assume ~ *a* < *b*
hence *b* ≤ *a* **by** (*simp only*: *linorder-not-less*)
hence *b* ^ *n* ≤ *a* ^ *n* **using** *nonneg* **by** (*rule power-mono*)
thus ~ *a* ^ *n* < *b* ^ *n* **by** (*simp only*: *linorder-not-less*)

qed

lemma *power-inject-base*:
[| *a ^ Suc n = b ^ Suc n*; *0 ≤ a*; *0 ≤ b* |]
==> *a = (b::'a::{ordered-semidom,recpower})*
by (*blast intro*: *power-le-imp-le-base order-antisym order-eq-refl sym*)

lemma *power-eq-imp-eq-base*:
fixes *a b :: 'a::{ordered-semidom,recpower}*
shows ⟦*a ^ n = b ^ n*; *0 ≤ a*; *0 ≤ b*; *0 < n*⟧ ⟹ *a = b*
by (*cases n*, *simp-all del*: *power-Suc*, *rule power-inject-base*)

The divides relation

lemma *le-imp-power-dvd*:
fixes *a :: 'a::{comm-semiring-1,recpower}*
assumes *m ≤ n* **shows** *a^m dvd a^n*
proof
have *a^n = a^(m + (n − m))*
using ‹*m ≤ n*› **by** *simp*
also have *... = a^m * a^(n − m)*
by (*rule power-add*)
finally show *a^n = a^m * a^(n − m)* **.**
qed

lemma *power-le-dvd*:
fixes *a b :: 'a::{comm-semiring-1,recpower}*
shows *a^n dvd b* ⟹ *m ≤ n* ⟹ *a^m dvd b*
by (*rule dvd-trans* [*OF le-imp-power-dvd*])

lemma *dvd-power-same*:
(*x::'a::{comm-semiring-1,recpower}*) *dvd y* ⟹ *x^n dvd y^n*
by (*induct n*) (*auto simp add*: *mult-dvd-mono*)

lemma *dvd-power-le*:
(*x::'a::{comm-semiring-1,recpower}*) *dvd y* ⟹ *m >= n* ⟹ *x^n dvd y^m*
by(*rule power-le-dvd*[*OF dvd-power-same*])

lemma *dvd-power* [*simp*]:
n > 0 | (*x::'a::{comm-semiring-1,recpower}*) *= 1* ⟹ *x dvd x^n*
apply (*erule disjE*)
apply (*subgoal-tac x ^ n = x^(Suc (n − 1))*)
apply (*erule ssubst*)
apply (*subst power-Suc*)
apply *auto*
done

16.2 Exponentiation for the Natural Numbers

instantiation *nat* :: *recpower*
begin

primrec *power-nat* **where**
  p ^ 0 = (1::nat)
  | *p ^ (Suc n) = (p::nat) * (p ^ n)*

instance proof
  fix *z n* :: *nat*
  show *z^0 = 1* **by** *simp*
  show *z^(Suc n) = z * (z^n)* **by** *simp*
qed

declare *power-nat.simps* [*simp del*]

end

lemma *of-nat-power*:
  of-nat (m ^ n) = (of-nat m::'a::{semiring-1,recpower}) ^ n
by (*induct n*, *simp-all add*: *of-nat-mult*)

lemma *nat-one-le-power* [*simp*]: *Suc 0 ≤ i ==> Suc 0 ≤ i^n*
by (*rule one-le-power* [*of i n*, *unfolded One-nat-def*])

lemma *nat-zero-less-power-iff* [*simp*]: *(x^n > 0) = (x > (0::nat) | n=0)*
by (*induct n*, *auto*)

lemma *nat-power-eq-Suc-0-iff* [*simp*]:
  ((x::nat)^m = Suc 0) = (m = 0 | x = Suc 0)
by (*induct-tac m*, *auto*)

lemma *power-Suc-0*[*simp*]: *(Suc 0)^n = Suc 0*
by *simp*

Valid for the naturals, but what if *0<i<1*? Premises cannot be weakened: consider the case where *i = (0::'a)*, *m = (1::'a)* and *n = (0::'a)*.

lemma *nat-power-less-imp-less*:
  assumes *nonneg*: *0 < (i::nat)*
  assumes *less*: *i^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*: *nonzero-mult-divide-cancel-left nz*)

end

17 Finite-Set: Finite sets

theory *Finite-Set*
imports *Nat Product-Type Power*
begin

17.1 Definition and basic properties

inductive *finite* :: '*a set* => *bool*
where
emptyI [*simp*, *intro*!]: *finite* {}
| *insertI* [*simp*, *intro*!]: *finite A* ==> *finite* (*insert a A*)

lemma *ex-new-if-finite*: — does not depend on def of finite at all
assumes ¬ *finite* (*UNIV* :: '*a set*) **and** *finite A*
shows ∃ *a*::'*a*. *a* ∉ *A*
proof −
from *assms* **have** *A* ≠ *UNIV* **by** *blast*
thus *?thesis* **by** *blast*
qed

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
finite F ==>
P {} ==> (!!*x F*. *finite F* ==> *x* ∉ *F* ==> *P F* ==> *P* (*insert x F*)) ==>
P F
— Discharging *x* ∉ *F* entails extra work.
proof −
assume *P* {} **and**
insert: !!*x F*. *finite F* ==> *x* ∉ *F* ==> *P F* ==> *P* (*insert x F*)
assume *finite F*
thus *P F*
proof *induct*
show *P* {} **by** *fact*
fix *x F* **assume** *F*: *finite F* **and** *P*: *P F*
show *P* (*insert x F*)
proof *cases*
assume *x* ∈ *F*
hence *insert x F* = *F* **by** (*rule insert-absorb*)
with *P* **show** *?thesis* **by** (*simp only*:)
next
assume *x* ∉ *F*

```
    from F this P show ?thesis by (rule insert)
   qed
  qed
qed

lemma finite-ne-induct[case-names singleton insert, consumes 2]:
assumes fin: finite F shows F ≠ {} ⟹
⟦ ⋀x. P{x};
  ⋀x F. ⟦ finite F; F ≠ {}; x ∉ F; P F ⟧ ⟹ P (insert x F) ⟧
⟹ P F
using fin
proof induct
  case empty thus ?case by simp
next
  case (insert x F)
  show ?case
  proof cases
    assume F = {}
    thus ?thesis using ‹P {x}› by simp
  next
    assume F ≠ {}
    thus ?thesis using insert by blast
  qed
qed

lemma finite-subset-induct [consumes 2, case-names empty insert]:
  assumes finite F and F ⊆ A
    and empty: P {}
    and insert: !!a F. finite F ==> a ∈ A ==> a ∉ F ==> P F ==> P (insert
a F)
  shows P F
proof −
  from ‹finite F› and ‹F ⊆ A›
  show ?thesis
  proof induct
    show P {} by fact
  next
    fix x F
    assume finite F and x ∉ F and
      P: F ⊆ A ==> P F and i: insert x F ⊆ A
    show P (insert x F)
    proof (rule insert)
      from i show x ∈ A by blast
      from i have F ⊆ A by blast
      with P show P F .
      show finite F by fact
      show x ∉ F by fact
    qed
  qed
```

qed

A finite choice principle. Does not need the SOME choice operator.

lemma *finite-set-choice*:
finite A $\Longrightarrow$ *ALL x:A. (EX y. P x y)* $\Longrightarrow$ *EX f. ALL x:A. P x (f x)*
proof (*induct set*: *finite*)
case *empty* **thus** *?case* **by** *simp*
next
case (*insert a A*)
then obtain *f b* **where** *f*: *ALL x:A. P x (f x)* **and** *ab*: *P a b* **by** *auto*
show *?case* (**is** *EX f. ?P f*)
proof
show *?P*(%*x. if x = a then b else f x*) **using** *f ab* **by** *auto*
qed
qed

Finite sets are the images of initial segments of natural numbers:

lemma *finite-imp-nat-seg-image-inj-on*:
assumes *fin*: *finite A*
shows $\exists$ (*n*::*nat*) *f. A = f ' {i. i<n} & inj-on f {i. i<n}*
using *fin*
proof *induct*
case *empty*
show *?case*
proof show $\exists f.$ *{} = f ' {i::nat. i < 0} & inj-on f {i. i<0}* **by** *simp*
qed
next
case (*insert a A*)
have *notinA*: *a* $\notin$ *A* **by** *fact*
from *insert.hyps* **obtain** *n f*
where *A = f ' {i::nat. i < n} inj-on f {i. i < n}* **by** *blast*
hence *insert a A = f*(*n*:=*a*) *' {i. i < Suc n}*
inj-on (*f*(*n*:=*a*)) *{i. i < Suc n}* **using** *notinA*
by (*auto simp add*: *image-def Ball-def inj-on-def less-Suc-eq*)
thus *?case* **by** *blast*
qed

lemma *nat-seg-image-imp-finite*:
!!*f A. A = f ' {i::nat. i<n}* $\Longrightarrow$ *finite A*
proof (*induct n*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc n*)
let *?B = f ' {i. i < n}*
have *finB*: *finite ?B* **by**(*rule Suc.hyps*[*OF refl*])
show *?case*
proof *cases*
assume $\exists$ *k<n. f n = f k*
hence *A = ?B* **using** *Suc.prems* **by**(*auto simp*:*less-Suc-eq*)

thus *?thesis* **using** *finB* **by** *simp*
next
assume ¬(∃ *k*<*n*. *f n* = *f k*)
hence *A* = *insert* (*f n*) *?B* **using** *Suc.prems* **by**(*auto simp*:*less-Suc-eq*)
thus *?thesis* **using** *finB* **by** *simp*
qed
qed

lemma *finite-conv-nat-seg-image*:
finite A = (∃ (*n*::*nat*) *f*. *A* = *f* ‘ {*i*::*nat*. *i*<*n*})
by(*blast intro*: *nat-seg-image-imp-finite dest*: *finite-imp-nat-seg-image-inj-on*)

lemma *finite-Collect-less-nat*[*iff*]: *finite*{*n*::*nat*. *n*<*k*}
by(*fastsimp simp*: *finite-conv-nat-seg-image*)

17.1.1 Finiteness and set theoretic constructions

lemma *finite-UnI*: *finite F* ==> *finite G* ==> *finite* (*F Un G*)
by (*induct set*: *finite*) *simp-all*

lemma *finite-subset*: *A* ⊆ *B* ==> *finite B* ==> *finite A*
— Every subset of a finite set is finite.
proof −
assume *finite B*
thus !!*A*. *A* ⊆ *B* ==> *finite A*
proof *induct*
case *empty*
thus *?case* **by** *simp*
next
case (*insert x F A*)
have *A*: *A* ⊆ *insert x F* **and** *r*: *A* − {*x*} ⊆ *F* ==> *finite* (*A* − {*x*}) **by** *fact*+
show *finite A*
proof *cases*
assume *x*: *x* ∈ *A*
with *A* **have** *A* − {*x*} ⊆ *F* **by** (*simp add*: *subset-insert-iff*)
with *r* **have** *finite* (*A* − {*x*}) **.**
hence *finite* (*insert x* (*A* − {*x*})) **..**
also have *insert x* (*A* − {*x*}) = *A* **using** *x* **by** (*rule insert-Diff*)
finally show *?thesis* **.**
next
show *A* ⊆ *F* ==> *?thesis* **by** *fact*
assume *x* ∉ *A*
with *A* **show** *A* ⊆ *F* **by** (*simp add*: *subset-insert-iff*)
qed
qed
qed

lemma *finite-Un* [*iff*]: *finite* (*F Un G*) = (*finite F* & *finite G*)
by (*blast intro*: *finite-subset* [*of* - *X Un Y*, *standard*] *finite-UnI*)

lemma *finite-Collect-disjI*[*simp*]:
finite{*x*. *P x* | *Q x*} = (*finite*{*x*. *P x*} & *finite*{*x*. *Q x*})
by(*simp add*:*Collect-disj-eq*)

lemma *finite-Int* [*simp*, *intro*]: *finite F* | *finite G* ==> *finite* (*F Int G*)
— The converse obviously fails.
by (*blast intro*: *finite-subset*)

lemma *finite-Collect-conjI* [*simp*, *intro*]:
finite{*x*. *P x*} | *finite*{*x*. *Q x*} ==> *finite*{*x*. *P x* & *Q x*}
— The converse obviously fails.
by(*simp add*:*Collect-conj-eq*)

lemma *finite-Collect-le-nat*[*iff*]: *finite*{*n*::*nat*. *n*<=*k*}
by(*simp add*: *le-eq-less-or-eq*)

lemma *finite-insert* [*simp*]: *finite* (*insert a A*) = *finite A*
apply (*subst insert-is-Un*)
apply (*simp only*: *finite-Un*, *blast*)
done

lemma *finite-Union*[*simp*, *intro*]:
⟦ *finite A*; !!*M*. *M* ∈ *A* ⟹ *finite M* ⟧ ⟹ *finite*(⋃ *A*)
by (*induct rule*:*finite-induct*) *simp-all*

lemma *finite-empty-induct*:
assumes *finite A*
and *P A*
and !!*a A*. *finite A* ==> *a*:*A* ==> *P A* ==> *P* (*A* − {*a*})
shows *P* {}
proof −
have *P* (*A* − *A*)
proof −
{
fix *c b* :: *'a set*
assume *c*: *finite c* **and** *b*: *finite b*
and *P1*: *P b* **and** *P2*: !!*x y*. *finite y* ==> *x* ∈ *y* ==> *P y* ==> *P* (*y* − {*x*})
have *c* ⊆ *b* ==> *P* (*b* − *c*)
using *c*
proof *induct*
case *empty*
from *P1* **show** *?case* **by** *simp*
next
case (*insert x F*)
have *P* (*b* − *F* − {*x*})
proof (*rule P2*)
from - *b* **show** *finite* (*b* − *F*) **by** (*rule finite-subset*) *blast*

```
        from insert show x ∈ b − F by simp
        from insert show P (b − F) by simp
      qed
      also have b − F − {x} = b − insert x F by (rule Diff-insert [symmetric])
      finally show ?case .
    qed
  }
  then show ?thesis by this (simp-all add: assms)
 qed
 then show ?thesis by simp
qed

lemma finite-Diff [simp]: finite A ==> finite (A − B)
by (rule Diff-subset [THEN finite-subset])

lemma finite-Diff2 [simp]:
 assumes finite B shows finite (A − B) = finite A
proof −
 have finite A ⟷ finite((A−B) Un (A Int B)) by(simp add: Un-Diff-Int)
 also have ... ⟷ finite(A−B) using ‹finite B› by(simp)
 finally show ?thesis ..
qed

lemma finite-compl[simp]:
 finite(A::'a set) ⟹ finite(−A) = finite(UNIV::'a set)
by(simp add:Compl-eq-Diff-UNIV)

lemma finite-Collect-not[simp]:
 finite{x::'a. P x} ⟹ finite{x. ~P x} = finite(UNIV::'a set)
by(simp add:Collect-neg-eq)

lemma finite-Diff-insert [iff]: finite (A − insert a B) = finite (A − B)
 apply (subst Diff-insert)
 apply (case-tac a : A − B)
  apply (rule finite-insert [symmetric, THEN trans])
  apply (subst insert-Diff, simp-all)
 done
```

Image and Inverse Image over Finite Sets

```
lemma finite-imageI[simp]: finite F ==> finite (h ‘ F)
 — The image of a finite set is finite.
 by (induct set: finite) simp-all

lemma finite-surj: finite A ==> B <= f ‘ A ==> finite B
 apply (frule finite-imageI)
 apply (erule finite-subset, assumption)
 done

lemma finite-range-imageI:
```

finite (*range g*) ==> *finite* (*range* (%*x*. *f* (*g x*)))
apply (*drule finite-imageI*, *simp add*: *range-composition*)
done

lemma *finite-imageD*: *finite* (*f*‘*A*) ==> *inj-on f A* ==> *finite A*
proof −
have *aux*: !!*A*. *finite* (*A* − {}) = *finite A* **by** *simp*
fix *B* :: *'a set*
assume *finite B*
thus !!*A*. *f*‘*A* = *B* ==> *inj-on f A* ==> *finite A*
apply *induct*
apply *simp*
apply (*subgoal-tac EX y*:*A*. *f y* = *x* & *F* = *f* ‘ (*A* − {*y*}))
apply *clarify*
apply (*simp* (*no-asm-use*) *add*: *inj-on-def*)
apply (*blast dest*!: *aux* [*THEN iffD1*], *atomize*)
apply (*erule-tac V* = *ALL A*. *?PP* (*A*) **in** *thin-rl*)
apply (*frule subsetD* [*OF equalityD2 insertI1*], *clarify*)
apply (*rule-tac x* = *xa* **in** *bexI*)
apply (*simp-all add*: *inj-on-image-set-diff*)
done
qed (*rule refl*)

lemma *inj-vimage-singleton*: *inj f* ==> *f*−‘{*a*} ⊆ {*THE x*. *f x* = *a*}
— The inverse image of a singleton under an injective function is included in a singleton.
apply (*auto simp add*: *inj-on-def*)
apply (*blast intro*: *the-equality* [*symmetric*])
done

lemma *finite-vimageI*: [|*finite F*; *inj h*|] ==> *finite* (*h* −‘ *F*)
— The inverse image of a finite set under an injective function is finite.
apply (*induct set*: *finite*)
apply *simp-all*
apply (*subst vimage-insert*)
apply (*simp add*: *finite-Un finite-subset* [*OF inj-vimage-singleton*])
done

The finite UNION of finite sets

lemma *finite-UN-I*: *finite A* ==> (!!*a*. *a*:*A* ==> *finite* (*B a*)) ==> *finite* (*UN a*:*A*. *B a*)
by (*induct set*: *finite*) *simp-all*

Strengthen RHS to (∀ *x*∈*A*. *finite* (*B x*)) ∧ *finite* {*x* ∈ *A*. *B x* ≠ {}}?
We'd need to prove *finite C* ⟹ ∀ *A B*. *UNION A B* ⊆ *C* ⟶ *finite* {*x* ∈ *A*. *B x* ≠ {}} by induction.

lemma *finite-UN* [*simp*]:
finite A ==> *finite* (*UNION A B*) = (*ALL x*:*A*. *finite* (*B x*))

by (*blast intro*: *finite-UN-I finite-subset*)

lemma *finite-Collect-bex*[*simp*]: *finite A* ⟹
finite{*x*. *EX y*:*A*. *Q x y*} = (*ALL y*:*A*. *finite*{*x*. *Q x y*})
apply(*subgoal-tac* {*x*. *EX y*:*A*. *Q x y*} = *UNION A* (%*y*. {*x*. *Q x y*}))
apply *auto*
done

lemma *finite-Collect-bounded-ex*[*simp*]: *finite*{*y*. *P y*} ⟹
finite{*x*. *EX y*. *P y* & *Q x y*} = (*ALL y*. *P y* ⟶ *finite*{*x*. *Q x y*})
apply(*subgoal-tac* {*x*. *EX y*. *P y* & *Q x y*} = *UNION* {*y*. *P y*} (%*y*. {*x*. *Q x y*}))
apply *auto*
done

lemma *finite-Plus*: [| *finite A*; *finite B* |] ==> *finite* (*A* <+> *B*)
by (*simp add*: *Plus-def*)

Sigma of finite sets

lemma *finite-SigmaI* [*simp*]:
finite A ==> (!!*a*. *a*:*A* ==> *finite* (*B a*)) ==> *finite* (*SIGMA a*:*A*. *B a*)
by (*unfold Sigma-def*) (*blast intro*!: *finite-UN-I*)

lemma *finite-cartesian-product*: [| *finite A*; *finite B* |] ==>
finite (*A* <*> *B*)
by (*rule finite-SigmaI*)

lemma *finite-Prod-UNIV*:
finite (*UNIV*::'*a set*) ==> *finite* (*UNIV*::'*b set*) ==> *finite* (*UNIV*::('*a* * '*b*) *set*)
apply (*subgoal-tac* (*UNIV*:: ('*a* * '*b*) *set*) = *Sigma UNIV* (%*x*. *UNIV*))
apply (*erule ssubst*)
apply (*erule finite-SigmaI*, *auto*)
done

lemma *finite-cartesian-productD1*:
[| *finite* (*A* <*> *B*); *B* ≠ {} |] ==> *finite A*
apply (*auto simp add*: *finite-conv-nat-seg-image*)
apply (*drule-tac x*=*n* **in** *spec*)
apply (*drule-tac x*=*fst o f* **in** *spec*)
apply (*auto simp add*: *o-def*)
prefer *2* **apply** (*force dest*!: *equalityD2*)
apply (*drule equalityD1*)
apply (*rename-tac y x*)
apply (*subgoal-tac* ∃*k*. *k*<*n* & *f k* = (*x*,*y*))
prefer *2* **apply** *force*
apply *clarify*
apply (*rule-tac x*=*k* **in** *image-eqI*, *auto*)
done

lemma *finite-cartesian-productD2*:
[| *finite* (*A* <*> *B*); *A* ≠ {} |] ==> *finite B*
apply (*auto simp add*: *finite-conv-nat-seg-image*)
apply (*drule-tac x*=*n* **in** *spec*)
apply (*drule-tac x*=*snd o f* **in** *spec*)
apply (*auto simp add*: *o-def*)
prefer *2* **apply** (*force dest*!: *equalityD2*)
apply (*drule equalityD1*)
apply (*rename-tac x y*)
apply (*subgoal-tac* ∃*k*. *k*<*n* & *f k* = (*x*,*y*))
prefer *2* **apply** *force*
apply *clarify*
apply (*rule-tac x*=*k* **in** *image-eqI*, *auto*)
done

The powerset of a finite set

lemma *finite-Pow-iff* [*iff*]: *finite* (*Pow A*) = *finite A*
proof
assume *finite* (*Pow A*)
with - **have** *finite* ((%*x*. {*x*}) ' *A*) **by** (*rule finite-subset*) *blast*
thus *finite A* **by** (*rule finite-imageD* [*unfolded inj-on-def*]) *simp*
next
assume *finite A*
thus *finite* (*Pow A*)
by *induct* (*simp-all add*: *finite-UnI finite-imageI Pow-insert*)
qed

lemma *finite-Collect-subsets*[*simp*,*intro*]: *finite A* ⟹ *finite*{*B*. *B* ⊆ *A*}
by(*simp add*: *Pow-def*[*symmetric*])

lemma *finite-UnionD*: *finite*(⋃ *A*) ⟹ *finite A*
by(*blast intro*: *finite-subset*[*OF subset-Pow-Union*])

17.2 Class *finite*

setup ⟪ *Sign.add-path finite* ⟫ — FIXME: name tweaking
class *finite* =
assumes *finite-UNIV*: *finite* (*UNIV* :: *'a set*)
setup ⟪ *Sign.parent-path* ⟫
hide *const finite*

context *finite*
begin

lemma *finite* [*simp*]: *finite* (*A* :: *'a set*)
by (*rule subset-UNIV finite-UNIV finite-subset*)+

end

lemma *UNIV-unit* [*noatp*]:
UNIV = {()} **by** *auto*

instance *unit* :: *finite*
by *default* (*simp add*: *UNIV-unit*)

lemma *UNIV-bool* [*noatp*]:
UNIV = {*False*, *True*} **by** *auto*

instance *bool* :: *finite*
by *default* (*simp add*: *UNIV-bool*)

instance ∗ :: (*finite*, *finite*) *finite*
by *default* (*simp only*: *UNIV-Times-UNIV* [*symmetric*] *finite-cartesian-product finite*)

lemma *inj-graph*: *inj* (%*f*. {(*x*, *y*). *y* = *f x*})
by (*rule inj-onI*, *auto simp add*: *expand-set-eq expand-fun-eq*)

instance *fun* :: (*finite*, *finite*) *finite*
proof
show *finite* (*UNIV* :: ('*a* => '*b*) *set*)
proof (*rule finite-imageD*)
let *?graph* = %*f*::'*a* => '*b*. {(*x*, *y*). *y* = *f x*}
have *range ?graph* ⊆ *Pow UNIV* **by** *simp*
moreover have *finite* (*Pow* (*UNIV* :: ('*a* ∗ '*b*) *set*))
by (*simp only*: *finite-Pow-iff finite*)
ultimately show *finite* (*range ?graph*)
by (*rule finite-subset*)
show *inj ?graph* **by** (*rule inj-graph*)
qed
qed

instance + :: (*finite*, *finite*) *finite*
by *default* (*simp only*: *UNIV-Plus-UNIV* [*symmetric*] *finite-Plus finite*)

17.3 A fold functional for finite sets

The intended behaviour is *fold f z* {x_1, ..., x_n} = *f* x_1 (... (*f* x_n *z*)...) if *f* is "left-commutative":

locale *fun-left-comm* =
fixes *f* :: '*a* ⇒ '*b* ⇒ '*b*
assumes *fun-left-comm*: *f x* (*f y z*) = *f y* (*f x z*)
begin

On a functional level it looks much nicer:

lemma *fun-comp-comm*: *f x* ∘ *f y* = *f y* ∘ *f x*

by (*simp add*: *fun-left-comm expand-fun-eq*)

end

inductive *fold-graph* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool$
for $f :: 'a \Rightarrow 'b \Rightarrow 'b$ **and** $z :: 'b$ **where**
emptyI [*intro*]: *fold-graph f z* {} *z* |
insertI [*intro*]: $x \notin A \Longrightarrow$ *fold-graph f z A y*
$\Longrightarrow$ *fold-graph f z* (*insert x A*) (*f x y*)

inductive-cases *empty-fold-graphE* [*elim*!]: *fold-graph f z* {} *x*

definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b$ **where**
[*code del*]: *fold f z A* = (*THE y*. *fold-graph f z A y*)

A tempting alternative for the definiens is *if finite A then THE y*. *fold-graph f z A y else e*. It allows the removal of finiteness assumptions from the theorems *fold-comm*, *fold-reindex* and *fold-distrib*. The proofs become ugly. It is not worth the effort. (???)

lemma *Diff1-fold-graph*:
fold-graph f z (*A* − {*x*}) *y* $\Longrightarrow$ $x \in A \Longrightarrow$ *fold-graph f z A* (*f x y*)
by (*erule insert-Diff* [*THEN subst*], *rule fold-graph.intros*, *auto*)

lemma *fold-graph-imp-finite*: *fold-graph f z A x* $\Longrightarrow$ *finite A*
by (*induct set*: *fold-graph*) *auto*

lemma *finite-imp-fold-graph*: *finite A* $\Longrightarrow \exists x$. *fold-graph f z A x*
by (*induct set*: *finite*) *auto*

17.3.1 From *fold-graph* to *fold*

lemma *image-less-Suc*: *h* ‘ {*i*. *i* < *Suc m*} = *insert* (*h m*) (*h* ‘ {*i*. *i* < *m*})
by (*auto simp add*: *less-Suc-eq*)

lemma *insert-image-inj-on-eq*:
[|*insert* (*h m*) *A* = *h* ‘ {*i*. *i* < *Suc m*}; *h m* ∉ *A*;
inj-on h {*i*. *i* < *Suc m*}|]
==> *A* = *h* ‘ {*i*. *i* < *m*}
apply (*auto simp add*: *image-less-Suc inj-on-def*)
apply (*blast intro*: *less-trans*)
done

lemma *insert-inj-onE*:
assumes *aA*: *insert a A* = *h*‘{*i*::*nat*. *i*<*n*} **and** *anot*: *a* ∉ *A*
and *inj-on*: *inj-on h* {*i*::*nat*. *i*<*n*}
shows ∃*hm m*. *inj-on hm* {*i*::*nat*. *i*<*m*} & *A* = *hm* ‘ {*i*. *i*<*m*} & *m* < *n*
proof (*cases n*)
case *0* **thus** *?thesis* **using** *aA* **by** *auto*
next

```
  case (Suc m)
  have nSuc: n = Suc m by fact
  have mlessn: m<n by (simp add: nSuc)
   from aA obtain k where hkeq: h k = a and klessn: k<n by (blast elim!:
equalityE)
  let ?hm = Fun.swap k m h
  have inj-hm: inj-on ?hm {i. i < n} using klessn mlessn
    by (simp add: inj-on-swap-iff inj-on)
  show ?thesis
  proof (intro exI conjI)
    show inj-on ?hm {i. i < m} using inj-hm
      by (auto simp add: nSuc less-Suc-eq intro: subset-inj-on)
    show m<n by (rule mlessn)
    show A = ?hm ' {i. i < m}
    proof (rule insert-image-inj-on-eq)
      show inj-on (Fun.swap k m h) {i. i < Suc m} using inj-hm nSuc by simp
      show ?hm m ∉ A by (simp add: swap-def hkeq anot)
      show insert (?hm m) A = ?hm ' {i. i < Suc m}
        using aA hkeq nSuc klessn
        by (auto simp add: swap-def image-less-Suc fun-upd-image
                  less-Suc-eq inj-on-image-set-diff [OF inj-on])
    qed
  qed
qed

context fun-left-comm
begin

lemma fold-graph-determ-aux:
  A = h'{i::nat. i<n} ⟹ inj-on h {i. i<n}
   ⟹ fold-graph f z A x ⟹ fold-graph f z A x'
   ⟹ x' = x
proof (induct n arbitrary: A x x' h rule: less-induct)
  case (less n)
  have IH: ⋀m h A x x'. m < n ⟹ A = h ' {i. i<m}
     ⟹ inj-on h {i. i<m} ⟹ fold-graph f z A x
     ⟹ fold-graph f z A x' ⟹ x' = x by fact
  have Afoldx: fold-graph f z A x and Afoldx': fold-graph f z A x'
    and A: A = h'{i. i<n} and injh: inj-on h {i. i<n} by fact+
  show ?case
  proof (rule fold-graph.cases [OF Afoldx])
    assume A = {} and x = z
    with Afoldx' show x' = x by auto
  next
    fix B b u
    assume AbB: A = insert b B and x: x = f b u
      and notinB: b ∉ B and Bu: fold-graph f z B u
    show x'=x
    proof (rule fold-graph.cases [OF Afoldx'])
```

```
    assume A = {} and x' = z
    with AbB show x' = x by blast
  next
    fix C c v
    assume AcC: A = insert c C and x': x' = f c v
      and notinC: c ∉ C and Cv: fold-graph f z C v
    from A AbB have Beq: insert b B = h`{i. i<n} by simp
    from insert-inj-onE [OF Beq notinB injh]
    obtain hB mB where inj-onB: inj-on hB {i. i < mB}
      and Beq: B = hB ` {i. i < mB} and lessB: mB < n by auto
    from A AcC have Ceq: insert c C = h`{i. i<n} by simp
    from insert-inj-onE [OF Ceq notinC injh]
    obtain hC mC where inj-onC: inj-on hC {i. i < mC}
      and Ceq: C = hC ` {i. i < mC} and lessC: mC < n by auto
    show x'=x
    proof cases
      assume b=c
      then moreover have B = C using AbB AcC notinB notinC by auto
      ultimately show ?thesis  using Bu Cv x x' IH [OF lessC Ceq inj-onC]
        by auto
    next
      assume diff: b ≠ c
      let ?D = B - {c}
      have B: B = insert c ?D and C: C = insert b ?D
        using AbB AcC notinB notinC diff by(blast elim!:equalityE)+
      have finite A by(rule fold-graph-imp-finite [OF Afoldx])
      with AbB have finite ?D by simp
      then obtain d where Dfoldd: fold-graph f z ?D d
        using finite-imp-fold-graph by iprover
      moreover have cinB: c ∈ B using B by auto
      ultimately have fold-graph f z B (f c d) by(rule Diff1-fold-graph)
      hence f c d = u by (rule IH [OF lessB Beq inj-onB Bu])
      moreover have f b d = v
      proof (rule IH[OF lessC Ceq inj-onC Cv])
        show fold-graph f z C (f b d) using C notinB Dfoldd by fastsimp
      qed
      ultimately show ?thesis
        using fun-left-comm [of c b] x x' by (auto simp add: o-def)
    qed
  qed
qed

lemma fold-graph-determ:
  fold-graph f z A x ⟹ fold-graph f z A y ⟹ y = x
apply (frule fold-graph-imp-finite [THEN finite-imp-nat-seg-image-inj-on])
apply (blast intro: fold-graph-determ-aux [rule-format])
done
```

lemma *fold-equality*:
  fold-graph f z A y $\Longrightarrow$ *fold f z A* = *y*
by (*unfold fold-def*) (*blast intro*: *fold-graph-determ*)

The base case for *fold*:

lemma (**in** −) *fold-empty* [*simp*]: *fold f z* {} = *z*
by (*unfold fold-def*) *blast*

The various recursion equations for *fold*:

lemma *fold-insert-aux*: *x* ∉ *A*
  $\Longrightarrow$ *fold-graph f z* (*insert x A*) *v* $\longleftrightarrow$
    (∃ *y*. *fold-graph f z A y* ∧ *v* = *f x y*)
apply *auto*
apply (*rule-tac A1* = *A* **and** *f1* = *f* **in** *finite-imp-fold-graph* [*THEN exE*])
 apply (*fastsimp dest*: *fold-graph-imp-finite*)
apply (*blast intro*: *fold-graph-determ*)
done

lemma *fold-insert* [*simp*]:
  finite A ==> *x* ∉ *A* ==> *fold f z* (*insert x A*) = *f x* (*fold f z A*)
apply (*simp add*: *fold-def fold-insert-aux*)
apply (*rule the-equality*)
 apply (*auto intro*: *finite-imp-fold-graph*
      cong add: *conj-cong simp add*: *fold-def*[*symmetric*] *fold-equality*)
done

lemma *fold-fun-comm*:
  finite A $\Longrightarrow$ *f x* (*fold f z A*) = *fold f* (*f x z*) *A*
proof (*induct rule*: *finite-induct*)
  case *empty* **then show** *?case* **by** *simp*
next
  case (*insert y A*) **then show** *?case*
    by (*simp add*: *fun-left-comm*[*of x*])
qed

lemma *fold-insert2*:
  finite A $\Longrightarrow$ *x* ∉ *A* $\Longrightarrow$ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
by (*simp add*: *fold-insert fold-fun-comm*)

lemma *fold-rec*:
assumes *finite A* **and** *x* ∈ *A*
shows *fold f z A* = *f x* (*fold f z* (*A* − {*x*}))
proof −
  have *A*: *A* = *insert x* (*A* − {*x*}) **using** ‹*x* ∈ *A*› **by** *blast*
  then have *fold f z A* = *fold f z* (*insert x* (*A* − {*x*})) **by** *simp*
  also have ... = *f x* (*fold f z* (*A* − {*x*}))
    by (*rule fold-insert*) (*simp add*: ‹*finite A*›)+
  finally show *?thesis* **.**
qed

lemma *fold-insert-remove*:
assumes *finite A*
shows *fold f z* (*insert x A*) = *f x* (*fold f z* (*A* − {*x*}))
proof −
from ‹*finite A*› **have** *finite* (*insert x A*) **by** *auto*
moreover have *x* ∈ *insert x A* **by** *auto*
ultimately have *fold f z* (*insert x A*) = *f x* (*fold f z* (*insert x A* − {*x*}))
by (*rule fold-rec*)
then show *?thesis* **by** *simp*
qed

end

A simplified version for idempotent functions:

locale *fun-left-comm-idem* = *fun-left-comm* +
assumes *fun-left-idem*: *f x* (*f x z*) = *f x z*
begin

The nice version:

lemma *fun-comp-idem* : *f x o f x* = *f x*
by (*simp add*: *fun-left-idem expand-fun-eq*)

lemma *fold-insert-idem*:
assumes *fin*: *finite A*
shows *fold f z* (*insert x A*) = *f x* (*fold f z A*)
proof *cases*
assume *x* ∈ *A*
then obtain *B* **where** *A* = *insert x B* **and** *x* ∉ *B* **by** (*rule set-insert*)
then show *?thesis* **using** *assms* **by** (*simp add*:*fun-left-idem*)
next
assume *x* ∉ *A* **then show** *?thesis* **using** *assms* **by** *simp*
qed

declare *fold-insert*[*simp del*] *fold-insert-idem*[*simp*]

lemma *fold-insert-idem2*:
finite A ⟹ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
by(*simp add*:*fold-fun-comm*)

end

17.3.2 The derived combinator *fold-image*

definition *fold-image* :: ('*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*) ⇒ '*b* ⇒ '*a set* ⇒ '*b*
where *fold-image f g* = *fold* (%*x y*. *f* (*g x*) *y*)

lemma *fold-image-empty*[*simp*]: *fold-image f g z* {} = *z*
by(*simp add*:*fold-image-def*)

context *ab-semigroup-mult*
begin

lemma *fold-image-insert*[*simp*]:
assumes *finite A* **and** $a \notin A$
shows *fold-image times g z* (*insert a A*) = *g a* ∗ (*fold-image times g z A*)
proof −
interpret *I*: *fun-left-comm* %*x y*. (*g x*) ∗ *y*
by *unfold-locales* (*simp add*: *mult-ac*)
show *?thesis* **using** *assms* **by**(*simp add*:*fold-image-def I*.*fold-insert*)
qed

lemma *fold-image-reindex*:
assumes *fin*: *finite A*
shows *inj-on h A* ⟹ *fold-image times g z* (*h*‘*A*) = *fold-image times* (*g*∘*h*) *z A*
using *fin* **apply** *induct*
apply *simp*
apply *simp*
done

lemma *fold-image-cong*:
finite A ⟹
(!!*x*. *x*:*A* ==> *g x* = *h x*) ==> *fold-image times g z A* = *fold-image times h z A*
apply (*subgoal-tac ALL C*. *C* <= *A* −−> (*ALL x*:*C*. *g x* = *h x*) −−> *fold-image times g z C* = *fold-image times h z C*)
apply *simp*
apply (*erule finite-induct*, *simp*)
apply (*simp add*: *subset-insert-iff*, *clarify*)
apply (*subgoal-tac finite C*)
prefer *2* **apply** (*blast dest*: *finite-subset* [*COMP swap-prems-rl*])
apply (*subgoal-tac C* = *insert x* (*C* − {*x*}))
prefer *2* **apply** *blast*
apply (*erule ssubst*)
apply (*drule spec*)
apply (*erule* (*1*) *notE impE*)
apply (*simp add*: *Ball-def del*: *insert-Diff-single*)
done

end

context *comm-monoid-mult*
begin

lemma *fold-image-Un-Int*:

finite A ==> finite B ==>
*fold-image times g 1 A * fold-image times g 1 B =*
*fold-image times g 1 (A Un B) * fold-image times g 1 (A Int B)*
by (*induct set*: *finite*)
(*auto simp add*: *mult-ac insert-absorb Int-insert-left*)

corollary *fold-Un-disjoint*:
finite A ==> finite B ==> A Int B = {} ==>
fold-image times g 1 (A Un B) =
*fold-image times g 1 A * fold-image times g 1 B*
by (*simp add*: *fold-image-Un-Int*)

lemma *fold-image-UN-disjoint*:
⟦ *finite I*; *ALL i:I. finite (A i)*;
ALL i:I. ALL j:I. i ≠ j --> A i Int A j = {} ⟧
⟹ *fold-image times g 1 (UNION I A) =*
fold-image times (%i. fold-image times g 1 (A i)) 1 I
apply (*induct set*: *finite*, *simp*, *atomize*)
apply (*subgoal-tac ALL i:F. x ≠ i*)
prefer *2* **apply** *blast*
apply (*subgoal-tac A x Int UNION F A = {}*)
prefer *2* **apply** *blast*
apply (*simp add*: *fold-Un-disjoint*)
done

lemma *fold-image-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
fold-image times (%x. fold-image times (g x) 1 (B x)) 1 A =
fold-image times (split g) 1 (SIGMA x:A. B x)
apply (*subst Sigma-def*)
apply (*subst fold-image-UN-disjoint*, *assumption*, *simp*)
apply *blast*
apply (*erule fold-image-cong*)
apply (*subst fold-image-UN-disjoint*, *simp*, *simp*)
apply *blast*
apply *simp*
done

lemma *fold-image-distrib*: *finite A* ⟹
*fold-image times (%x. g x * h x) 1 A =*
*fold-image times g 1 A * fold-image times h 1 A*
by (*erule finite-induct*) (*simp-all add*: *mult-ac*)

lemma *fold-image-related*:
assumes *Re*: *R e e*
and *Rop*: ∀ *x1 y1 x2 y2*. *R x1 x2* ∧ *R y1 y2* ⟶ *R* (*x1 * y1*) (*x2 * y2*)
and *fS*: *finite S* **and** *Rfg*: ∀ *x*∈*S*. *R* (*h x*) (*g x*)
shows *R* (*fold-image* (*op **) *h e S*) (*fold-image* (*op **) *g e S*)
using *fS* **by** (*rule finite-subset-induct*) (*insert assms*, *auto*)

lemma *fold-image-eq-general*:
assumes *fS*: *finite S*
and *h*: $\forall\, y{\in}S'.\ \exists!x.\ x{\in}\ S \wedge h(x) = y$
and *f12*: $\forall\, x{\in}S.\ h\ x \in S' \wedge f2(h\ x) = f1\ x$
shows *fold-image* (*op* $*$) *f1 e S* = *fold-image* (*op* $*$) *f2 e S'*
proof−
from *h f12* **have** *hS*: *h* ' *S* = *S'* **by** *auto*
{fix *x y* **assume** *H*: *x* ∈ *S y* ∈ *S h x* = *h y*
from *f12 h H* **have** *x* = *y* **by** *auto* **}**
hence *hinj*: *inj-on h S* **unfolding** *inj-on-def Ex1-def* **by** *blast*
from *f12* **have** *th*: $\bigwedge x.\ x \in S \Longrightarrow (f2 \circ h)\ x = f1\ x$ **by** *auto*
from *hS* **have** *fold-image* (*op* $*$) *f2 e S'* = *fold-image* (*op* $*$) *f2 e* (*h* ' *S*) **by** *simp*
also have ... = *fold-image* (*op* $*$) (*f2 o h*) *e S*
using *fold-image-reindex*[*OF fS hinj*, *of f2 e*] **.**
also have ... = *fold-image* (*op* $*$) *f1 e S* **using** *th fold-image-cong*[*OF fS*, *of f2 o h f1 e*]
by *blast*
finally show *?thesis* **..**
qed

lemma *fold-image-eq-general-inverses*:
assumes *fS*: *finite S*
and *kh*: $\bigwedge y.\ y \in T \Longrightarrow k\ y \in S \wedge h\ (k\ y) = y$
and *hk*: $\bigwedge x.\ x \in S \Longrightarrow h\ x \in T \wedge k\ (h\ x) = x\ \wedge g\ (h\ x) = f\ x$
shows *fold-image* (*op* $*$) *f e S* = *fold-image* (*op* $*$) *g e T*

apply (*rule fold-image-eq-general*[*OF fS*, *of T h g f e*])
apply (*rule ballI*)
apply (*frule kh*)
apply (*rule ex1I*[])
apply *blast*
apply *clarsimp*
apply (*drule hk*) **apply** *simp*
apply (*rule sym*)
apply (*erule conjunct1*[*OF conjunct2*[*OF hk*]])
apply (*rule ballI*)
apply (*drule hk*)
apply *blast*
done

end

17.4 Generalized summation over a set

interpretation *comm-monoid-add*: *comm-monoid-mult 0*::'*a*::*comm-monoid-add op* +
proof qed (*auto intro*: *add-assoc add-commute*)

definition *setsum* :: (*'a => 'b*) => *'a set* => *'b*::*comm-monoid-add*
where *setsum f A == if finite A then fold-image (op +) f 0 A else 0*

abbreviation
Setsum (∑ - [*1000*] *999*) **where**
∑ *A == setsum* (%*x*. *x*) *A*

Now: lot's of fancy syntax. First, *setsum* (λ*x*. *e*) *A* is written ∑ *x*∈*A*. *e*.

syntax
-setsum :: *pttrn => 'a set => 'b => 'b*::*comm-monoid-add* ((*3SUM* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-setsum :: *pttrn => 'a set => 'b => 'b*::*comm-monoid-add* ((*3*∑ -∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setsum :: *pttrn => 'a set => 'b => 'b*::*comm-monoid-add* ((*3*∑ -∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
SUM i:*A*. *b* == *CONST setsum* (%*i*. *b*) *A*
∑ *i*∈*A*. *b* == *CONST setsum* (%*i*. *b*) *A*

Instead of ∑ *x*∈{*x*. *P*}. *e* we introduce the shorter ∑ *x*|*P*. *e*.

syntax
-qsetsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3SUM* - |/ -./ -) [*0*,*0*,*10*] *10*)
syntax (*xsymbols*)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*∑ - | (-)./ -) [*0*,*0*,*10*] *10*)
syntax (*HTML* **output**)
-qsetsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* ((*3*∑ - | (-)./ -) [*0*,*0*,*10*] *10*)

translations
SUM x|*P*. *t* => *CONST setsum* (%*x*. *t*) {*x*. *P*}
∑ *x*|*P*. *t* => *CONST setsum* (%*x*. *t*) {*x*. *P*}

print-translation ⟪
let
fun setsum-tr' [*Abs*(*x*,*Tx*,*t*), *Const* (*Collect*,-) $ *Abs*(*y*,*Ty*,*P*)] =
if x<>*y then raise Match*
else let val x' = *Syntax.mark-bound x*
val t' = *subst-bound*(*x'*,*t*)
val P' = *subst-bound*(*x'*,*P*)
in Syntax.const -qsetsum $ *Syntax.mark-bound x* $ *P'* $ *t' end*
in [(*setsum*, *setsum-tr'*)] *end*
⟫

lemma *setsum-empty* [*simp*]: *setsum f* {} = *0*
by (*simp add*: *setsum-def*)

lemma *setsum-insert* [*simp*]:
  finite F ==> a ∉ F ==> setsum f (insert a F) = f a + setsum f F
by (*simp add*: *setsum-def*)

lemma *setsum-infinite* [*simp*]: ~ *finite A ==> setsum f A = 0*
by (*simp add*: *setsum-def*)

lemma *setsum-reindex*:
    inj-on f B ==> setsum h (f ‘ B) = setsum (h ∘ f) B
by(*auto simp add*: *setsum-def comm-monoid-add.fold-image-reindex dest*!:*finite-imageD*)

lemma *setsum-reindex-id*:
    inj-on f B ==> setsum f B = setsum id (f ‘ B)
by (*auto simp add*: *setsum-reindex*)

lemma *setsum-reindex-nonzero*:
  assumes *fS*: *finite S*
  and *nz*: ⋀ *x y*. *x ∈ S ⟹ y ∈ S ⟹ x ≠ y ⟹ f x = f y ⟹ h (f x) = 0*
  shows *setsum h (f ‘ S) = setsum (h o f) S*
using *nz*
proof(*induct rule*: *finite-induct*[*OF fS*])
  case *1* **thus** *?case* **by** *simp*
next
  case (*2 x F*)
  {assume *fxF*: *f x ∈ f ‘ F* **hence** ∃ *y ∈ F . f y = f x* **by** *auto*
    then obtain *y* **where** *y*: *y ∈ F f x = f y* **by** *auto*
    from *2.hyps y* **have** *xy*: *x ≠ y* **by** *auto*

    from *2.prems*[*of x y*] *2.hyps xy y* **have** *h0*: *h (f x) = 0* **by** *simp*
    have *setsum h (f ‘ insert x F) = setsum h (f ‘ F)* **using** *fxF* **by** *auto*
    also have *... = setsum (h o f) (insert x F)*
      unfolding *setsum-insert*[*OF ‹finite F› ‹x∉F›*]
      using *h0*
      apply *simp*
      apply (*rule 2.hyps*(*3*))
      apply (*rule-tac y=y* **in** *2.prems*)
      apply *simp-all*
      done
    finally have *?case* **.}**
  moreover
  {assume *fxF*: *f x ∉ f ‘ F*
    have *setsum h (f ‘ insert x F) = h (f x) + setsum h (f ‘ F)*
      using *fxF 2.hyps* **by** *simp*
    also have *... = setsum (h o f) (insert x F)*
      unfolding *setsum-insert*[*OF ‹finite F› ‹x∉F›*]
      apply *simp*
      apply (*rule cong*[*OF refl*[*of op + (h (f x))*]])
      apply (*rule 2.hyps*(*3*))
      apply (*rule-tac y=y* **in** *2.prems*)

apply *simp-all*
done
finally have *?case* **.}**
ultimately show *?case* **by** *blast*
qed

lemma *setsum-cong*:
A = B ==> (!!x. x:B ==> f x = g x) ==> setsum f A = setsum g B
by(*fastsimp simp*: *setsum-def intro*: *comm-monoid-add.fold-image-cong*)

lemma *strong-setsum-cong*[*cong*]:
A = B ==> (!!x. x:B =simp=> f x = g x)
==> setsum (%x. f x) A = setsum (%x. g x) B
by(*fastsimp simp*: *simp-implies-def setsum-def intro*: *comm-monoid-add.fold-image-cong*)

lemma *setsum-cong2*: ⟦⋀*x*. *x* ∈ *A* ⟹ *f x* = *g x*⟧ ⟹ *setsum f A* = *setsum g A*
by (*rule setsum-cong*[*OF refl*], *auto*)

lemma *setsum-reindex-cong*:
[|*inj-on f A*; *B* = *f* ' *A*; !!*a*. *a*:*A* ⟹ *g a* = *h* (*f a*)|]
==> setsum h B = setsum g A
by (*simp add*: *setsum-reindex cong*: *setsum-cong*)

lemma *setsum-0*[*simp*]: *setsum* (%*i*. *0*) *A* = *0*
apply (*clarsimp simp*: *setsum-def*)
apply (*erule finite-induct*, *auto*)
done

lemma *setsum-0'*: *ALL a*:*A*. *f a* = *0 ==> setsum f A* = *0*
by(*simp add*:*setsum-cong*)

lemma *setsum-Un-Int*: *finite A ==> finite B ==>*
setsum g (*A Un B*) + *setsum g* (*A Int B*) = *setsum g A* + *setsum g B*
— The reversed orientation looks more natural, but LOOPS as a simprule!
by(*simp add*: *setsum-def comm-monoid-add.fold-image-Un-Int* [*symmetric*])

lemma *setsum-Un-disjoint*: *finite A ==> finite B*
==> A Int B = {} *==> setsum g* (*A Un B*) = *setsum g A* + *setsum g B*
by (*subst setsum-Un-Int* [*symmetric*], *auto*)

lemma *setsum-mono-zero-left*:
assumes *fT*: *finite T* **and** *ST*: *S* ⊆ *T*
and *z*: ∀ *i* ∈ *T* − *S*. *f i* = *0*
shows *setsum f S* = *setsum f T*
proof−
have *eq*: *T* = *S* ∪ (*T* − *S*) **using** *ST* **by** *blast*
have *d*: *S* ∩ (*T* − *S*) = {} **using** *ST* **by** *blast*
from *fT ST* **have** *f*: *finite S finite* (*T* − *S*) **by** (*auto intro*: *finite-subset*)

show *?thesis*
by (*simp add*: *setsum-Un-disjoint*[*OF f d*, *unfolded eq*[*symmetric*]] *setsum-0'*[*OF z*])
qed

lemma *setsum-mono-zero-right*:
finite T $\Longrightarrow$ *S* $\subseteq$ *T* $\Longrightarrow$ $\forall$ *i* $\in$ *T* $-$ *S*. *f i* = *0* $\Longrightarrow$ *setsum f T* = *setsum f S*
by(*blast intro*!: *setsum-mono-zero-left*[*symmetric*])

lemma *setsum-mono-zero-cong-left*:
assumes *fT*: *finite T* **and** *ST*: *S* $\subseteq$ *T*
and *z*: $\forall$ *i* $\in$ *T* $-$ *S*. *g i* = *0*
and *fg*: $\bigwedge$*x*. *x* $\in$ *S* $\Longrightarrow$ *f x* = *g x*
shows *setsum f S* = *setsum g T*
proof−
have *eq*: *T* = *S* $\cup$ (*T* $-$ *S*) **using** *ST* **by** *blast*
have *d*: *S* $\cap$ (*T* $-$ *S*) = {} **using** *ST* **by** *blast*
from *fT ST* **have** *f*: *finite S finite* (*T* $-$ *S*) **by** (*auto intro*: *finite-subset*)
show *?thesis*
using *fg* **by** (*simp add*: *setsum-Un-disjoint*[*OF f d*, *unfolded eq*[*symmetric*]] *setsum-0'*[*OF z*])
qed

lemma *setsum-mono-zero-cong-right*:
assumes *fT*: *finite T* **and** *ST*: *S* $\subseteq$ *T*
and *z*: $\forall$ *i* $\in$ *T* $-$ *S*. *f i* = *0*
and *fg*: $\bigwedge$*x*. *x* $\in$ *S* $\Longrightarrow$ *f x* = *g x*
shows *setsum f T* = *setsum g S*
using *setsum-mono-zero-cong-left*[*OF fT ST z*] *fg*[*symmetric*] **by** *auto*

lemma *setsum-delta*:
assumes *fS*: *finite S*
shows *setsum* (*λk*. *if k*=*a then b k else 0*) *S* = (*if a* $\in$ *S then b a else 0*)
proof−
let *?f* = (*λk*. *if k*=*a then b k else 0*)
{**assume** *a*: *a* $\notin$ *S*
hence $\forall$ *k*$\in$ *S*. *?f k* = *0* **by** *simp*
hence *?thesis* **using** *a* **by** *simp*}
moreover
{**assume** *a*: *a* $\in$ *S*
let *?A* = *S* $-$ {*a*}
let *?B* = {*a*}
have *eq*: *S* = *?A* $\cup$ *?B* **using** *a* **by** *blast*
have *dj*: *?A* $\cap$ *?B* = {} **by** *simp*
from *fS* **have** *fAB*: *finite ?A finite ?B* **by** *auto*
have *setsum ?f S* = *setsum ?f ?A* + *setsum ?f ?B*
using *setsum-Un-disjoint*[*OF fAB dj*, *of ?f*, *unfolded eq*[*symmetric*]]
by *simp*
then have *?thesis* **using** *a* **by** *simp*}

ultimately show *?thesis* **by** *blast*
qed
lemma *setsum-delta'*:
assumes *fS*: *finite S* **shows**
setsum (λ*k*. *if a = k then b k else 0*) *S* =
(*if a∈ S then b a else 0*)
using *setsum-delta*[*OF fS*, *of a b*, *symmetric*]
by (*auto intro*: *setsum-cong*)

lemma *setsum-restrict-set*:
assumes *fA*: *finite A*
shows *setsum f* (*A* ∩ *B*) = *setsum* (λ*x*. *if x* ∈ *B then f x else 0*) *A*
proof−
from *fA* **have** *fab*: *finite* (*A* ∩ *B*) **by** *auto*
have *aba*: *A* ∩ *B* ⊆ *A* **by** *blast*
let *?g* = λ*x*. *if x* ∈ *A*∩*B then f x else 0*
from *setsum-mono-zero-left*[*OF fA aba*, *of ?g*]
show *?thesis* **by** *simp*
qed

lemma *setsum-cases*:
assumes *fA*: *finite A*
shows *setsum* (λ*x*. *if x* ∈ *B then f x else g x*) *A* =
setsum f (*A* ∩ *B*) + *setsum g* (*A* ∩ − *B*)
proof−
have *a*: *A* = *A* ∩ *B* ∪ *A* ∩ −*B* (*A* ∩ *B*) ∩ (*A* ∩ −*B*) = {}
by *blast*+
from *fA*
have *f*: *finite* (*A* ∩ *B*) *finite* (*A* ∩ −*B*) **by** *auto*
let *?g* = λ*x*. *if x* ∈ *B then f x else g x*
from *setsum-Un-disjoint*[*OF f a*(*2*), *of ?g*] *a*(*1*)
show *?thesis* **by** *simp*
qed

lemma *setsum-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. *i* ≠ *j* −−> *A i Int A j* = {}) ==>
setsum f (*UNION I A*) = (∑ *i*∈*I*. *setsum f* (*A i*))
by(*simp add*: *setsum-def comm-monoid-add.fold-image-UN-disjoint cong*: *setsum-cong*)

No need to assume that C is finite. If infinite, the rhs is directly 0, and $\bigcup C$ is also infinite, hence the lhs is also 0.

lemma *setsum-Union-disjoint*:
[| (*ALL A*:*C*. *finite A*);
(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* −−> *A Int B* = {}) |]
==> *setsum f* (*Union C*) = *setsum* (*setsum f*) *C*
apply (*cases finite C*)

prefer *2* **apply** (*force dest*: *finite-UnionD simp add*: *setsum-def*)
apply (*frule setsum-UN-disjoint* [*of C id f*])
apply (*unfold Union-def id-def*, *assumption+*)
done

lemma *setsum-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
$(\sum x{\in}A.\ (\sum y{\in}B\ x.\ f\ x\ y)) = (\sum (x,y){\in}(SIGMA\ x{:}A.\ B\ x).\ f\ x\ y)$
by(*simp add*:*setsum-def comm-monoid-add.fold-image-Sigma split-def cong*:*setsum-cong*)

Here we can eliminate the finiteness assumptions, by cases.

lemma *setsum-cartesian-product*:
$(\sum x{\in}A.\ (\sum y{\in}B.\ f\ x\ y)) = (\sum (x,y) \in A <*> B.\ f\ x\ y)$
apply (*cases finite A*)
apply (*cases finite B*)
apply (*simp add*: *setsum-Sigma*)
apply (*cases A={}*, *simp*)
apply (*simp*)
apply (*auto simp add*: *setsum-def*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *setsum-addf*: *setsum* (%*x*. *f x* + *g x*) *A* = (*setsum f A* + *setsum g A*)
by(*simp add*:*setsum-def comm-monoid-add.fold-image-distrib*)

17.4.1 Properties in more restricted classes of structures

lemma *setsum-SucD*: *setsum f A = Suc n ==> EX a:A. 0 < f a*
apply (*case-tac finite A*)
prefer *2* **apply** (*simp add*: *setsum-def*)
apply (*erule rev-mp*)
apply (*erule finite-induct*, *auto*)
done

lemma *setsum-eq-0-iff* [*simp*]:
finite F ==> (setsum f F = 0) = (ALL a:F. f a = (0::nat))
by (*induct set*: *finite*) *auto*

lemma *setsum-eq-Suc0-iff*: *finite A* $\Longrightarrow$
(*setsum f A = Suc 0*) = (*EX a:A. f a = Suc 0 & (ALL b:A. a≠b* $\longrightarrow$ *f b = 0*))
apply(*erule finite-induct*)
apply (*auto simp add*:*add-is-1*)
done

lemmas *setsum-eq-1-iff = setsum-eq-Suc0-iff*[*simplified One-nat-def*[*symmetric*]]

lemma *setsum-Un-nat*: *finite A ==> finite B ==>*
(*setsum f* (*A Un B*) :: *nat*) = *setsum f A + setsum f B − setsum f* (*A Int B*)
— For the natural numbers, we have subtraction.

by (*subst setsum-Un-Int* [*symmetric*], *auto simp add*: *algebra-simps*)

lemma *setsum-Un*: *finite A ==> finite B ==>*
(*setsum f* (*A Un B*) :: *'a* :: *ab-group-add*) =
setsum f A + setsum f B − setsum f (*A Int B*)
by (*subst setsum-Un-Int* [*symmetric*], *auto simp add*: *algebra-simps*)

lemma (**in** *comm-monoid-mult*) *fold-image-1*: *finite S* $\Longrightarrow$ $(\forall x{\in}S.\ f\ x = 1) \Longrightarrow$ *fold-image op ∗ f 1 S = 1*
apply (*induct set*: *finite*)
apply *simp* **by** (*auto simp add*: *fold-image-insert*)

lemma (**in** *comm-monoid-mult*) *fold-image-Un-one*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: $\forall x \in S{\cap}T.\ f\ x = 1$
shows *fold-image* (*op ∗*) *f 1* (*S ∪ T*) = *fold-image* (*op ∗*) *f 1 S ∗ fold-image* (*op ∗*) *f 1 T*
proof−
have *fold-image op ∗ f 1* (*S ∩ T*) = *1*
apply (*rule fold-image-1*)
using *fS fT I0* **by** *auto*
with *fold-image-Un-Int*[*OF fS fT*] **show** *?thesis* **by** *simp*
qed

lemma *setsum-eq-general-reverses*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *kh*: $\bigwedge y.\ y \in T \Longrightarrow k\ y \in S \wedge h\ (k\ y) = y$
and *hk*: $\bigwedge x.\ x \in S \Longrightarrow h\ x \in T \wedge k\ (h\ x) = x \wedge g\ (h\ x) = f\ x$
shows *setsum f S = setsum g T*
apply (*simp add*: *setsum-def fS fT*)
apply (*rule comm-monoid-add.fold-image-eq-general-inverses*[*OF fS*])
apply (*erule kh*)
apply (*erule hk*)
done

lemma *setsum-Un-zero*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: $\forall x \in S{\cap}T.\ f\ x = 0$
shows *setsum f* (*S ∪ T*) = *setsum f S* + *setsum f T*
using *fS fT*
apply (*simp add*: *setsum-def*)
apply (*rule comm-monoid-add.fold-image-Un-one*)
using *I0* **by** *auto*

lemma *setsum-UNION-zero*:
assumes *fS*: *finite S* **and** *fSS*: $\forall\ T \in S.$ *finite T*

and *f0*: $\bigwedge T1\ T2\ x.\ T1{\in}S \Longrightarrow T2{\in}S \Longrightarrow T1 \neq T2 \Longrightarrow x \in T1 \Longrightarrow x \in T2 \Longrightarrow f\ x = 0$
shows $setsum\ f\ (\bigcup S) = setsum\ (\lambda T.\ setsum\ f\ T)\ S$
using *fSS f0*
proof(*induct rule*: *finite-induct*[*OF fS*])
case *1* **thus** *?case* **by** *simp*
next
case (*2 T F*)
then have *fTF*: *finite T* $\forall\ T{\in}F.$ *finite T finite F* **and** *TF*: $T \notin F$
and *H*: $setsum\ f\ (\bigcup F) = setsum\ (setsum\ f)\ F$ **by** (*auto simp add*: *finite-insert*)
from *fTF* **have** *fUF*: *finite* $(\bigcup F)$ **by** (*auto intro*: *finite-Union*)
from *2.prems TF fTF*
show *?case*
by (*auto simp add*: *H*[*symmetric*] *intro*: *setsum-Un-zero*[*OF fTF*(*1*) *fUF*, *of f*])
qed

lemma *setsum-diff1-nat*: $(setsum\ f\ (A - \{a\}) :: nat) =$
(*if a*:*A then setsum f A* $-$ *f a else setsum f A*)
apply (*case-tac finite A*)
prefer *2* **apply** (*simp add*: *setsum-def*)
apply (*erule finite-induct*)
apply (*auto simp add*: *insert-Diff-if*)
apply (*drule-tac a* $=$ *a* **in** *mk-disjoint-insert*, *auto*)
done

lemma *setsum-diff1*: *finite A* $\Longrightarrow$
$(setsum\ f\ (A - \{a\}) :: ('a{::}ab\text{-}group\text{-}add)) =$
(*if a*:*A then setsum f A* $-$ *f a else setsum f A*)
by (*erule finite-induct*) (*auto simp add*: *insert-Diff-if*)

lemma *setsum-diff1'*[*rule-format*]:
finite A $\Longrightarrow a \in A \longrightarrow (\sum\ x \in A.\ f\ x) = f\ a + (\sum\ x \in (A - \{a\}).\ f\ x)$
apply (*erule finite-induct*[**where** $F{=}A$ **and** $P{=}\%\ A.\ (a \in A \longrightarrow (\sum\ x \in A.\ f\ x) = f\ a + (\sum\ x \in (A - \{a\}).\ f\ x))$])
apply (*auto simp add*: *insert-Diff-if add-ac*)
done

lemma *setsum-diff-nat*:
assumes *finite B* **and** $B \subseteq A$
shows $(setsum\ f\ (A - B) :: nat) = (setsum\ f\ A) - (setsum\ f\ B)$
using *assms*
proof *induct*
show $setsum\ f\ (A - \{\}) = (setsum\ f\ A) - (setsum\ f\ \{\})$ **by** *simp*
next
fix *F x* **assume** *finF*: *finite F* **and** *xnotinF*: $x \notin F$

```
    and xFinA: insert x F ⊆ A
    and IH: F ⊆ A ⟹ setsum f (A − F) = setsum f A − setsum f F
  from xnotinF xFinA have xinAF: x ∈ (A − F) by simp
  from xinAF have A: setsum f ((A − F) − {x}) = setsum f (A − F) − f x
    by (simp add: setsum-diff1-nat)
  from xFinA have F ⊆ A by simp
  with IH have setsum f (A − F) = setsum f A − setsum f F by simp
  with A have B: setsum f ((A − F) − {x}) = setsum f A − setsum f F − f x
    by simp
  from xnotinF have A − insert x F = (A − F) − {x} by auto
  with B have C: setsum f (A − insert x F) = setsum f A − setsum f F − f x
    by simp
  from finF xnotinF have setsum f (insert x F) = setsum f F + f x by simp
  with C have setsum f (A − insert x F) = setsum f A − setsum f (insert x F)
    by simp
  thus setsum f (A − insert x F) = setsum f A − setsum f (insert x F) by simp
qed

lemma setsum-diff:
  assumes le: finite A B ⊆ A
  shows setsum f (A − B) = setsum f A − ((setsum f B)::('a::ab-group-add))
proof −
  from le have finiteB: finite B using finite-subset by auto
  show ?thesis using finiteB le
  proof induct
    case empty
    thus ?case by auto
  next
    case (insert x F)
    thus ?case using le finiteB
      by (simp add: Diff-insert[where a=x and B=F] setsum-diff1 insert-absorb)
  qed
qed

lemma setsum-mono:
 assumes le: ⋀i. i∈K ⟹ f (i::'a) ≤ ((g i)::('b::{comm-monoid-add, pordered-ab-semigroup-add}))
  shows (∑ i∈K. f i) ≤ (∑ i∈K. g i)
proof (cases finite K)
  case True
  thus ?thesis using le
  proof induct
    case empty
    thus ?case by simp
  next
    case insert
    thus ?case using add-mono by fastsimp
  qed
next
  case False
```

thus *?thesis*
by (*simp add*: *setsum-def*)
qed

lemma *setsum-strict-mono*:
fixes $f :: 'a \Rightarrow 'b$::{*pordered-cancel-ab-semigroup-add*,*comm-monoid-add*}
assumes *finite* A $A \neq \{\}$
and $!!x.\ x{:}A \Longrightarrow f\ x < g\ x$
shows *setsum* $f\ A <$ *setsum* $g\ A$
using *prems*
proof (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **by** (*auto simp*: *add-strict-mono*)
qed

lemma *setsum-negf*:
setsum $(\%x.\ -\ (f\ x)::'a::$*ab-group-add*$)\ A = -$ *setsum* $f\ A$
proof (*cases finite* A)
case *True* **thus** *?thesis* **by** (*induct set*: *finite*) *auto*
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-subtractf*:
setsum $(\%x.\ ((f\ x)::'a::$*ab-group-add*$)\ -\ g\ x)\ A =$
setsum $f\ A -$ *setsum* $g\ A$
proof (*cases finite* A)
case *True* **thus** *?thesis* **by** (*simp add*: *diff-minus setsum-addf setsum-negf*)
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-nonneg*:
assumes *nn*: $\forall x{\in}A.\ (0::'a::$\{*pordered-ab-semigroup-add*,*comm-monoid-add*\}$) \leq f\ x$
shows $0 \leq$ *setsum* $f\ A$
proof (*cases finite* A)
case *True* **thus** *?thesis* **using** *nn*
proof *induct*
case *empty* **then show** *?case* **by** *simp*
next
case (*insert* $x\ F$)
then have $0 + 0 \leq f\ x +$ *setsum* $f\ F$ **by** (*blast intro*: *add-mono*)
with *insert* **show** *?case* **by** *simp*
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-nonpos*:
assumes *np*: $\forall x{\in}A.\ f\,x \leq (0{::}'a{::}\{pordered\text{-}ab\text{-}semigroup\text{-}add, comm\text{-}monoid\text{-}add\})$
shows $setsum\ f\ A \leq 0$
proof (*cases finite A*)
case *True* **thus** *?thesis* **using** *np*
proof *induct*
case *empty* **then show** *?case* **by** *simp*
next
case (*insert x F*)
then have $f\,x + setsum\ f\ F \leq 0 + 0$ **by** (*blast intro*: *add-mono*)
with *insert* **show** *?case* **by** *simp*
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-mono2*:
fixes $f :: 'a \Rightarrow 'b :: \{pordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le, comm\text{-}monoid\text{-}add\}$
assumes *fin*: *finite B* **and** *sub*: $A \subseteq B$ **and** *nn*: $\bigwedge b.\ b \in B{-}A \Longrightarrow 0 \leq f\,b$
shows $setsum\ f\ A \leq setsum\ f\ B$
proof −
have $setsum\ f\ A \leq setsum\ f\ A + setsum\ f\ (B{-}A)$
by(*simp add*: *add-increasing2*[*OF setsum-nonneg*] *nn Ball-def*)
also have $\ldots = setsum\ f\ (A \cup (B{-}A))$ **using** *fin finite-subset*[*OF sub fin*]
by (*simp add*:*setsum-Un-disjoint del*:*Un-Diff-cancel*)
also have $A \cup (B{-}A) = B$ **using** *sub* **by** *blast*
finally show *?thesis* **.**
qed

lemma *setsum-mono3*: *finite B ==> A <= B ==>*
ALL x: *B − A*.
0 <= ((f x)::'a::{comm-monoid-add,pordered-ab-semigroup-add}) ==>
setsum f A <= setsum f B
apply (*subgoal-tac setsum f B = setsum f A + setsum f (B − A)*)
apply (*erule ssubst*)
apply (*subgoal-tac setsum f A + 0 <= setsum f A + setsum f (B − A)*)
apply *simp*
apply (*rule add-left-mono*)
apply (*erule setsum-nonneg*)
apply (*subst setsum-Un-disjoint* [*THEN sym*])
apply (*erule finite-subset*, *assumption*)
apply (*rule finite-subset*)
prefer *2*
apply *assumption*
apply *auto*
apply (*rule setsum-cong*)
apply *auto*
done

lemma *setsum-right-distrib*:
fixes $f :: 'a => ('b::semiring\text{-}0)$
shows $r * setsum\ f\ A = setsum\ (\%n.\ r * f\ n)\ A$
proof (*cases finite A*)
case *True*
thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*simp add*: *right-distrib*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-left-distrib*:
$setsum\ f\ A * (r::'a::semiring\text{-}0) = (\sum n{\in}A.\ f\ n * r)$
proof (*cases finite A*)
case *True*
then show *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*simp add*: *left-distrib*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-divide-distrib*:
$setsum\ f\ A\ /\ (r::'a::field) = (\sum n{\in}A.\ f\ n\ /\ r)$
proof (*cases finite A*)
case *True*
then show *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*simp add*: *add-divide-distrib*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-abs*[*iff*]:
fixes $f :: 'a => ('b::pordered\text{-}ab\text{-}group\text{-}add\text{-}abs)$
shows $abs\ (setsum\ f\ A) \leq setsum\ (\%i.\ abs(f\ i))\ A$
proof (*cases finite A*)
case *True*

thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*)
thus *?case* **by** (*auto intro*: *abs-triangle-ineq order-trans*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *setsum-abs-ge-zero*[*iff*]:
fixes *f* :: *'a* => (*'b*::*pordered-ab-group-add-abs*)
shows *0* ≤ *setsum* (%*i*. *abs*(*f i*)) *A*
proof (*cases finite A*)
case *True*
thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*) **thus** *?case* **by** (*auto simp*: *add-nonneg-nonneg*)
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

lemma *abs-setsum-abs*[*simp*]:
fixes *f* :: *'a* => (*'b*::*pordered-ab-group-add-abs*)
shows *abs* (∑ *a*∈*A*. *abs*(*f a*)) = (∑ *a*∈*A*. *abs*(*f a*))
proof (*cases finite A*)
case *True*
thus *?thesis*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert a A*)
hence |∑ *a*∈*insert a A*. |*f a*|| = ||*f a*| + (∑ *a*∈*A*. |*f a*|)| **by** *simp*
also have ... = ||*f a*| + |∑ *a*∈*A*. |*f a*||| **using** *insert* **by** *simp*
also have ... = |*f a*| + |∑ *a*∈*A*. |*f a*||
by (*simp del*: *abs-of-nonneg*)
also have ... = (∑ *a*∈*insert a A*. |*f a*|) **using** *insert* **by** *simp*
finally show *?case* **.**
qed
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

Commuting outer and inner summation

lemma *swap-inj-on*:

inj-on (%(*i*, *j*). (*j*, *i*)) (*A* × *B*)
by (*unfold inj-on-def*) *fast*

lemma *swap-product*:
(%(*i*, *j*). (*j*, *i*)) ' (*A* × *B*) = *B* × *A*
by (*simp add*: *split-def image-def*) *blast*

lemma *setsum-commute*:
(∑*i*∈*A*. ∑*j*∈*B*. *f i j*) = (∑*j*∈*B*. ∑*i*∈*A*. *f i j*)
proof (*simp add*: *setsum-cartesian-product*)
have (∑(*x*,*y*) ∈ *A* <*> *B*. *f x y*) =
(∑(*y*,*x*) ∈ (%(*i*, *j*). (*j*, *i*)) ' (*A* × *B*). *f x y*)
(**is** *?s* = -)
apply (*simp add*: *setsum-reindex* [**where** *f* = %(*i*, *j*). (*j*, *i*)] *swap-inj-on*)
apply (*simp add*: *split-def*)
done
also have ... = (∑(*y*,*x*)∈*B* × *A*. *f x y*)
(**is** - = *?t*)
apply (*simp add*: *swap-product*)
done
finally show *?s* = *?t* **.**
qed

lemma *setsum-product*:
fixes *f* :: *'a* => (*'b*::*semiring-0*)
shows *setsum f A* ∗ *setsum g B* = (∑*i*∈*A*. ∑*j*∈*B*. *f i* ∗ *g j*)
by (*simp add*: *setsum-right-distrib setsum-left-distrib*) (*rule setsum-commute*)

17.5 Generalized product over a set

definition *setprod* :: (*'a* => *'b*) => *'a set* => *'b*::*comm-monoid-mult*
where *setprod f A* == *if finite A then fold-image* (*op* ∗) *f 1 A else 1*

abbreviation
Setprod (∏- [*1000*] *999*) **where**
∏*A* == *setprod* (%*x*. *x*) *A*

syntax
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3PROD* -:-. -) [*0*, *51*, *10*] *10*)
syntax (*xsymbols*)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*∏-∈-. -) [*0*, *51*, *10*] *10*)
syntax (*HTML* **output**)
-setprod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* ((*3*∏-∈-. -) [*0*, *51*, *10*] *10*)

translations — Beware of argument permutation!
PROD i:*A*. *b* == *CONST setprod* (%*i*. *b*) *A*

$\prod i{\in}A.\ b == CONST\ setprod\ (\%i.\ b)\ A$

Instead of $\prod x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\prod x|P.\ e$.

syntax
-qsetprod :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3PROD - |/ -./ -) [0,0,10] 10)
syntax (*xsymbols*)
-qsetprod :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∏ - | (-)./ -) [0,0,10] 10)
syntax (*HTML* **output**)
-qsetprod :: pttrn ⇒ bool ⇒ 'a ⇒ 'a ((3∏ - | (-)./ -) [0,0,10] 10)

translations
PROD x|P. t => CONST setprod (%x. t) {x. P}
∏ x|P. t => CONST setprod (%x. t) {x. P}

lemma *setprod-empty* [*simp*]: *setprod f {} = 1*
by (*auto simp add*: *setprod-def*)

lemma *setprod-insert* [*simp*]: [| *finite A*; *a ∉ A* |] ==>
*setprod f (insert a A) = f a * setprod f A*
by (*simp add*: *setprod-def*)

lemma *setprod-infinite* [*simp*]: *~ finite A ==> setprod f A = 1*
by (*simp add*: *setprod-def*)

lemma *setprod-reindex*:
inj-on f B ==> setprod h (f ' B) = setprod (h ∘ f) B
by(*auto simp*: *setprod-def fold-image-reindex dest*!:*finite-imageD*)

lemma *setprod-reindex-id*: *inj-on f B ==> setprod f B = setprod id (f ' B)*
by (*auto simp add*: *setprod-reindex*)

lemma *setprod-cong*:
A = B ==> (!!x. x:B ==> f x = g x) ==> setprod f A = setprod g B
by(*fastsimp simp*: *setprod-def intro*: *fold-image-cong*)

lemma *strong-setprod-cong*[*cong*]:
A = B ==> (!!x. x:B =simp=> f x = g x) ==> setprod f A = setprod g B
by(*fastsimp simp*: *simp-implies-def setprod-def intro*: *fold-image-cong*)

lemma *setprod-reindex-cong*: *inj-on f A ==>*
B = f ' A ==> g = h ∘ f ==> setprod h B = setprod g A
by (*frule setprod-reindex*, *simp*)

lemma *strong-setprod-reindex-cong*: **assumes** *i*: *inj-on f A*
and *B*: *B = f ' A* **and** *eq*: *⋀x. x ∈ A ⟹ g x = (h ∘ f) x*
shows *setprod h B = setprod g A*
proof−
have *setprod h B = setprod (h o f) A*

```
    by (simp add: B setprod-reindex[OF i, of h])
  then show ?thesis apply simp
    apply (rule setprod-cong)
    apply simp
    by (simp add: eq)
qed

lemma setprod-Un-one:
  assumes fS: finite S and fT: finite T
  and I0: ∀ x ∈ S∩T. f x = 1
  shows setprod f (S ∪ T) = setprod f S * setprod f T
  using fS fT
  apply (simp add: setprod-def)
  apply (rule fold-image-Un-one)
  using I0 by auto


lemma setprod-1: setprod (%i. 1) A = 1
apply (case-tac finite A)
apply (erule finite-induct, auto simp add: mult-ac)
done

lemma setprod-1': ALL a:F. f a = 1 ==> setprod f F = 1
apply (subgoal-tac setprod f F = setprod (%x. 1) F)
apply (erule ssubst, rule setprod-1)
apply (rule setprod-cong, auto)
done

lemma setprod-Un-Int: finite A ==> finite B
   ==> setprod g (A Un B) * setprod g (A Int B) = setprod g A * setprod g B
by(simp add: setprod-def fold-image-Un-Int[symmetric])

lemma setprod-Un-disjoint: finite A ==> finite B
  ==> A Int B = {} ==> setprod g (A Un B) = setprod g A * setprod g B
by (subst setprod-Un-Int [symmetric], auto)

lemma setprod-mono-one-left:
  assumes fT: finite T and ST: S ⊆ T
  and z: ∀ i ∈ T - S. f i = 1
  shows setprod f S = setprod f T
proof-
  have eq: T = S ∪ (T - S) using ST by blast
  have d: S ∩ (T - S) = {} using ST by blast
  from fT ST have f: finite S finite (T - S) by (auto intro: finite-subset)
  show ?thesis
  by (simp add: setprod-Un-disjoint[OF f d, unfolded eq[symmetric]] setprod-1'[OF
z])
qed
```

lemmas *setprod-mono-one-right* = *setprod-mono-one-left* [*THEN sym*]

lemma *setprod-delta*:
assumes *fS*: *finite S*
shows *setprod* (λ*k*. *if k=a then b k else 1*) *S* = (*if a* ∈ *S then b a else 1*)
proof−
let *?f* = (λ*k*. *if k=a then b k else 1*)
{assume *a*: *a* ∉ *S*
hence ∀ *k*∈ *S*. *?f k* = *1* **by** *simp*
hence *?thesis* **using** *a* **by** (*simp add*: *setprod-1 cong add*: *setprod-cong*) **}**
moreover
{assume *a*: *a* ∈ *S*
let *?A* = *S* − {*a*}
let *?B* = {*a*}
have *eq*: *S* = *?A* ∪ *?B* **using** *a* **by** *blast*
have *dj*: *?A* ∩ *?B* = {} **by** *simp*
from *fS* **have** *fAB*: *finite ?A finite ?B* **by** *auto*
have *fA1*: *setprod ?f ?A* = *1* **apply** (*rule setprod-1'*) **by** *auto*
have *setprod ?f ?A* ∗ *setprod ?f ?B* = *setprod ?f S*
using *setprod-Un-disjoint*[*OF fAB dj*, *of ?f*, *unfolded eq*[*symmetric*]]
by *simp*
then have *?thesis* **using** *a* **by** (*simp add*: *fA1 cong add*: *setprod-cong cong del*: *if-weak-cong*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *setprod-delta'*:
assumes *fS*: *finite S* **shows**
setprod (λ*k*. *if a* = *k then b k else 1*) *S* =
(*if a*∈ *S then b a else 1*)
using *setprod-delta*[*OF fS*, *of a b*, *symmetric*]
by (*auto intro*: *setprod-cong*)

lemma *setprod-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. *i* ≠ *j* −−> *A i Int A j* = {}) ==>
setprod f (*UNION I A*) = *setprod* (%*i*. *setprod f* (*A i*)) *I*
by(*simp add*: *setprod-def fold-image-UN-disjoint cong*: *setprod-cong*)

lemma *setprod-Union-disjoint*:
[| (*ALL A*:*C*. *finite A*);
(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* −−> *A Int B* = {}) |]
==> *setprod f* (*Union C*) = *setprod* (*setprod f*) *C*
apply (*cases finite C*)
prefer *2* **apply** (*force dest*: *finite-UnionD simp add*: *setprod-def*)
apply (*frule setprod-UN-disjoint* [*of C id f*])
apply (*unfold Union-def id-def*, *assumption*+)
done

lemma *setprod-Sigma*: *finite A ==> ALL x:A. finite (B x) ==>*
(∏ x∈A. (∏ y∈ B x. f x y)) =
(∏ (x,y)∈(SIGMA x:A. B x). f x y)
by(*simp add*:*setprod-def fold-image-Sigma split-def cong*:*setprod-cong*)

Here we can eliminate the finiteness assumptions, by cases.

lemma *setprod-cartesian-product*:
(∏ x∈A. (∏ y∈ B. f x y)) = (∏ (x,y)∈(A <> B). f x y)*
apply (*cases finite A*)
apply (*cases finite B*)
apply (*simp add*: *setprod-Sigma*)
apply (*cases A={}*, *simp*)
apply (*simp add*: *setprod-1*)
apply (*auto simp add*: *setprod-def*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *setprod-timesf*:
*setprod (%x. f x * g x) A = (setprod f A * setprod g A)*
by(*simp add*:*setprod-def fold-image-distrib*)

17.5.1 Properties in more restricted classes of structures

lemma *setprod-eq-1-iff* [*simp*]:
finite F ==> (setprod f F = 1) = (ALL a:F. f a = (1::nat))
by (*induct set*: *finite*) *auto*

lemma *setprod-zero*:
finite A ==> EX x: A. f x = (0::'a::comm-semiring-1) ==> setprod f A = 0
apply (*induct set*: *finite*, *force*, *clarsimp*)
apply (*erule disjE*, *auto*)
done

lemma *setprod-nonneg* [*rule-format*]:
(ALL x: A. (0::'a::ordered-semidom) ≤ f x) --> 0 ≤ setprod f A
by (*cases finite A*, *induct set*: *finite*, *simp-all add*: *mult-nonneg-nonneg*)

lemma *setprod-pos* [*rule-format*]: *(ALL x: A. (0::'a::ordered-semidom) < f x)*
--> 0 < setprod f A
by (*cases finite A*, *induct set*: *finite*, *simp-all add*: *mult-pos-pos*)

lemma *setprod-zero-iff*[*simp*]: *finite A ==>*
(setprod f A = (0::'a::{comm-semiring-1,no-zero-divisors})) =
(EX x: A. f x = 0)
by (*erule finite-induct*, *auto simp*:*no-zero-divisors*)

lemma *setprod-pos-nat*:
finite S ==> (ALL x : S. f x > (0::nat)) ==> setprod f S > 0

using *setprod-zero-iff* **by**(*simp del*:*neq0-conv add*:*neq0-conv*[*symmetric*])

lemma *setprod-pos-nat-iff*[*simp*]:
finite S ==> (setprod f S > 0) = (ALL x : S. f x > (0::nat))
using *setprod-zero-iff* **by**(*simp del*:*neq0-conv add*:*neq0-conv*[*symmetric*])

lemma *setprod-Un*: *finite A ==> finite B ==> (ALL x*: *A Int B. f x ≠ 0) ==>*
(setprod f (A Un B) :: 'a ::{field})
*= setprod f A * setprod f B / setprod f (A Int B)*
by (*subst setprod-Un-Int* [*symmetric*], *auto*)

lemma *setprod-diff1*: *finite A ==> f a ≠ 0 ==>*
(setprod f (A − {a}) :: 'a :: {field}) =
(if a:*A then setprod f A / f a else setprod f A)*
by (*erule finite-induct*) (*auto simp add*: *insert-Diff-if*)

lemma *setprod-inversef*: *finite A ==>*
ALL x: *A. f x ≠ (0::'a::{field,division-by-zero}) ==>*
setprod (inverse ∘ f) A = inverse (setprod f A)
by (*erule finite-induct*) *auto*

lemma *setprod-dividef*:
[|*finite A*;
∀ *x* ∈ *A. g x ≠ (0::'a::{field,division-by-zero})*|]
==> setprod (%x. f x / g x) A = setprod f A / setprod g A
apply (*subgoal-tac*
*setprod (%x. f x / g x) A = setprod (%x. f x * (inverse ∘ g) x) A*)
apply (*erule ssubst*)
apply (*subst divide-inverse*)
apply (*subst setprod-timesf*)
apply (*subst setprod-inversef*, *assumption+*, *rule refl*)
apply (*rule setprod-cong*, *rule refl*)
apply (*subst divide-inverse*, *auto*)
done

lemma *setprod-dvd-setprod* [*rule-format*]:
(ALL x : A. f x dvd g x) ⟶ setprod f A dvd setprod g A
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply (*auto simp add*: *dvd-def*)
apply (*rule-tac x = k * ka* **in** *exI*)
apply (*simp add*: *algebra-simps*)
done

lemma *setprod-dvd-setprod-subset*:
finite B ⟹ A <= B ⟹ setprod f A dvd setprod f B
apply (*subgoal-tac setprod f B = setprod f A * setprod f (B − A)*)
apply (*unfold dvd-def*, *blast*)
apply (*subst setprod-Un-disjoint* [*symmetric*])

apply (*auto elim*: *finite-subset intro*: *setprod-cong*)
done

lemma *setprod-dvd-setprod-subset2*:
finite B $\Longrightarrow$ *A* <= *B* $\Longrightarrow$ *ALL x* : *A*. (*f x*::'*a*::*comm-semiring-1*) *dvd g x* $\Longrightarrow$
setprod f A dvd setprod g B
apply (*rule dvd-trans*)
apply (*rule setprod-dvd-setprod*, *erule* (*1*) *bspec*)
apply (*erule* (*1*) *setprod-dvd-setprod-subset*)
done

lemma *dvd-setprod*: *finite A* $\Longrightarrow$ *i*:*A* $\Longrightarrow$
(*f i* ::'*a*::*comm-semiring-1*) *dvd setprod f A*
by (*induct set*: *finite*) (*auto intro*: *dvd-mult*)

lemma *dvd-setsum* [*rule-format*]: (*ALL i* : *A*. *d dvd f i*) $\longrightarrow$
(*d*::'*a*::*comm-semiring-1*) *dvd* (*SUM x* : *A*. *f x*)
apply (*cases finite A*)
apply (*induct set*: *finite*)
apply *auto*
done

17.6 Finite cardinality

This definition, although traditional, is ugly to work with: *card A* == *LEAST n*. *EX f*. *A* = {*f i* | *i*. *i* < *n*}. But now that we have *setsum* things are easy:

definition *card* :: '*a set* $\Rightarrow$ *nat*
where *card A* = *setsum* (λx. *1*) *A*

lemma *card-empty* [*simp*]: *card* {} = *0*
by (*simp add*: *card-def*)

lemma *card-infinite* [*simp*]: ~ *finite A* ==> *card A* = *0*
by (*simp add*: *card-def*)

lemma *card-eq-setsum*: *card A* = *setsum* (%*x*. *1*) *A*
by (*simp add*: *card-def*)

lemma *card-insert-disjoint* [*simp*]:
finite A ==> *x* $\notin$ *A* ==> *card* (*insert x A*) = *Suc*(*card A*)
by(*simp add*: *card-def*)

lemma *card-insert-if*:
finite A ==> *card* (*insert x A*) = (*if x*:*A then card A else Suc*(*card*(*A*)))
by (*simp add*: *insert-absorb*)

lemma *card-0-eq* [*simp*,*noatp*]: *finite A* ==> (*card A* = *0*) = (*A* = {})
apply *auto*

apply (*drule-tac a = x* **in** *mk-disjoint-insert*, *clarify*, *auto*)
done

lemma *card-eq-0-iff*: (*card A = 0*) = (*A = {}* | ~ *finite A*)
by *auto*

lemma *card-Suc-Diff1*: *finite A ==> x*: *A ==> Suc* (*card* (*A* − {*x*})) = *card A*
apply(*rule-tac t = A* **in** *insert-Diff* [*THEN subst*], *assumption*)
apply(*simp del*:*insert-Diff-single*)
done

lemma *card-Diff-singleton*:
finite A ==> x: *A ==> card* (*A* − {*x*}) = *card A* − *1*
by (*simp add*: *card-Suc-Diff1* [*symmetric*])

lemma *card-Diff-singleton-if*:
finite A ==> card (*A*−{*x*}) = (*if x* : *A then card A* − *1 else card A*)
by (*simp add*: *card-Diff-singleton*)

lemma *card-Diff-insert*[*simp*]:
assumes *finite A* **and** *a*:*A* **and** *a* ~: *B*
shows *card*(*A* − *insert a B*) = *card*(*A* − *B*) − *1*
proof −
have *A* − *insert a B* = (*A* − *B*) − {*a*} **using** *assms* **by** *blast*
then show *?thesis* **using** *assms* **by**(*simp add*:*card-Diff-singleton*)
qed

lemma *card-insert*: *finite A ==> card* (*insert x A*) = *Suc* (*card* (*A* − {*x*}))
by (*simp add*: *card-insert-if card-Suc-Diff1 del*:*card-Diff-insert*)

lemma *card-insert-le*: *finite A ==> card A <= card* (*insert x A*)
by (*simp add*: *card-insert-if*)

lemma *card-mono*: ⟦ *finite B*; *A* ⊆ *B* ⟧ ⟹ *card A* ≤ *card B*
by (*simp add*: *card-def setsum-mono2*)

lemma *card-seteq*: *finite B ==>* (!!*A*. *A <= B ==> card B <= card A ==> A* = *B*)
apply (*induct set*: *finite*, *simp*, *clarify*)
apply (*subgoal-tac finite A* & *A* − {*x*} <= *F*)
prefer *2* **apply** (*blast intro*: *finite-subset*, *atomize*)
apply (*drule-tac x = A* − {*x*} **in** *spec*)
apply (*simp add*: *card-Diff-singleton-if split add*: *split-if-asm*)
apply (*case-tac card A*, *auto*)
done

lemma *psubset-card-mono*: *finite B ==> A < B ==> card A < card B*
apply (*simp add*: *psubset-eq linorder-not-le* [*symmetric*])

apply (*blast dest*: *card-seteq*)
done

lemma *card-Un-Int*: *finite A* ==> *finite B*
==> *card A* + *card B* = *card* (*A Un B*) + *card* (*A Int B*)
by(*simp add*:*card-def setsum-Un-Int*)

lemma *card-Un-disjoint*: *finite A* ==> *finite B*
==> *A Int B* = {} ==> *card* (*A Un B*) = *card A* + *card B*
by (*simp add*: *card-Un-Int*)

lemma *card-Diff-subset*:
finite B ==> *B* <= *A* ==> *card* (*A* − *B*) = *card A* − *card B*
by(*simp add*:*card-def setsum-diff-nat*)

lemma *card-Diff1-less*: *finite A* ==> *x*: *A* ==> *card* (*A* − {*x*}) < *card A*
apply (*rule Suc-less-SucD*)
apply (*simp add*: *card-Suc-Diff1 del*:*card-Diff-insert*)
done

lemma *card-Diff2-less*:
finite A ==> *x*: *A* ==> *y*: *A* ==> *card* (*A* − {*x*} − {*y*}) < *card A*
apply (*case-tac x* = *y*)
apply (*simp add*: *card-Diff1-less del*:*card-Diff-insert*)
apply (*rule less-trans*)
prefer *2* **apply** (*auto intro*!: *card-Diff1-less simp del*:*card-Diff-insert*)
done

lemma *card-Diff1-le*: *finite A* ==> *card* (*A* − {*x*}) <= *card A*
apply (*case-tac x* : *A*)
apply (*simp-all add*: *card-Diff1-less less-imp-le*)
done

lemma *card-psubset*: *finite B* ==> *A* ⊆ *B* ==> *card A* < *card B* ==> *A* < *B*
by (*erule psubsetI*, *blast*)

lemma *insert-partition*:
⟦ *x* ∉ *F*; ∀ *c1* ∈ *insert x F*. ∀ *c2* ∈ *insert x F*. *c1* ≠ *c2* ⟶ *c1* ∩ *c2* = {} ⟧
⟹ *x* ∩ ⋃ *F* = {}
by *auto*

main cardinality theorem

lemma *card-partition* [*rule-format*]:
finite C ==>
finite (⋃ *C*) −−>
(∀ *c*∈*C*. *card c* = *k*) −−>
(∀ *c1* ∈ *C*. ∀ *c2* ∈ *C*. *c1* ≠ *c2* −−> *c1* ∩ *c2* = {}) −−>
k ∗ *card*(*C*) = *card* (⋃ *C*)
apply (*erule finite-induct*, *simp*)

apply (*simp add*: *card-insert-disjoint card-Un-disjoint insert-partition*
finite-subset [*of* - $\bigcup$ (*insert x F*)])
done

The form of a finite set of given cardinality

lemma *card-eq-SucD*:
assumes *card A = Suc k*
shows $\exists$ *b B. A = insert b B & b* $\notin$ *B & card B = k & (k=0* $\longrightarrow$ *B={})*
proof −
have *fin*: *finite A* **using** *assms* **by** (*auto intro*: *ccontr*)
moreover have *card A* $\neq$ *0* **using** *assms* **by** *auto*
ultimately obtain *b* **where** *b*: *b* $\in$ *A* **by** *auto*
show *?thesis*
proof (*intro exI conjI*)
show *A = insert b* (*A*−{*b*}) **using** *b* **by** *blast*
show *b* $\notin$ *A* − {*b*} **by** *blast*
show *card* (*A* − {*b*}) = *k* **and** *k = 0* $\longrightarrow$ *A* − {*b*} = {}
using *assms b fin* **by**(*fastsimp dest*:*mk-disjoint-insert*)+
qed
qed

lemma *card-Suc-eq*:
(*card A = Suc k*) =
(∃ *b B. A = insert b B & b* $\notin$ *B & card B = k & (k=0* $\longrightarrow$ *B={}*))
apply(*rule iffI*)
apply(*erule card-eq-SucD*)
apply(*auto*)
apply(*subst card-insert*)
apply(*auto intro*:*ccontr*)
done

lemma *setsum-constant* [*simp*]: ($\sum$ *x* ∈ *A. y*) = *of-nat*(*card A*) ∗ *y*
apply (*cases finite A*)
apply (*erule finite-induct*)
apply (*auto simp add*: *algebra-simps*)
done

lemma *setprod-constant*: *finite A* ==> ($\prod$ *x*∈ *A.* (*y*::′*a*::{*recpower*, *comm-monoid-mult*}))
= *y* ^(*card A*)
apply (*erule finite-induct*)
apply (*auto simp add*: *power-Suc*)
done

lemma *setprod-gen-delta*:
assumes *fS*: *finite S*
shows *setprod* (λ*k. if k=a then b k else c*) *S* = (*if a* ∈ *S then* (*b a* ::′*a*::{*comm-monoid-mult*,
recpower}) ∗ *c* ^ (*card S* − *1*) *else c* ^ *card S*)
proof−
let *?f* = (λ*k. if k=a then b k else c*)

{assume *a*: $a \notin S$
hence $\forall\ k{\in}\ S.\ ?f\ k = c$ **by** *simp*
hence *?thesis* **using** *a setprod-constant*[*OF fS*, *of c*] **by** (*simp add*: *setprod-1 cong add*: *setprod-cong*) **}**
moreover
{assume *a*: $a \in S$
let $?A = S - \{a\}$
let $?B = \{a\}$
have *eq*: $S = ?A \cup ?B$ **using** *a* **by** *blast*
have *dj*: $?A \cap ?B = \{\}$ **by** *simp*
from *fS* **have** *fAB*: *finite ?A finite ?B* **by** *auto*
have *fA0*:*setprod ?f* $?A$ = *setprod* $(\lambda i.\ c)$ $?A$
apply (*rule setprod-cong*) **by** *auto*
have *cA*: *card* $?A$ = *card* $S - 1$ **using** *fS a* **by** *auto*
have *fA1*: *setprod ?f* $?A = c\ \hat{}$ *card* $?A$ **unfolding** *fA0* **apply** (*rule setprod-constant*) **using** *fS* **by** *auto*
have *setprod ?f* $?A * $ *setprod ?f* $?B$ = *setprod ?f S*
using *setprod-Un-disjoint*[*OF fAB dj*, *of ?f*, *unfolded eq*[*symmetric*]]
by *simp*
then have *?thesis* **using** *a cA*
by (*simp add*: *fA1 ring-simps cong add*: *setprod-cong cong del*: *if-weak-cong*)**}**
ultimately show *?thesis* **by** *blast*
qed

lemma *setsum-bounded*:
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow f\ i \leq (K::'a::\{semiring\text{-}1, pordered\text{-}ab\text{-}semigroup\text{-}add\})$
shows *setsum f A* $\leq$ *of-nat*(*card A*) $*$ *K*
proof (*cases finite A*)
case *True*
thus *?thesis* **using** *le setsum-mono*[**where** $K{=}A$ **and** $g = \%x.\ K$] **by** *simp*
next
case *False* **thus** *?thesis* **by** (*simp add*: *setsum-def*)
qed

17.6.1 Cardinality of unions

lemma *card-UN-disjoint*:
finite I ==> (*ALL i*:*I*. *finite* (*A i*)) ==>
(*ALL i*:*I*. *ALL j*:*I*. $i \neq j$ --> *A i Int A j* = {}) ==> *card* (*UNION I A*) = $(\sum i{\in}I.\ card(A\ i))$
apply (*simp add*: *card-def del*: *setsum-constant*)
apply (*subgoal-tac*
setsum (%*i*. *card* (*A i*)) *I* = *setsum* (%*i*. (*setsum* (%*x*. *1*) (*A i*))) *I*)
apply (*simp add*: *setsum-UN-disjoint del*: *setsum-constant*)
apply (*simp cong*: *setsum-cong*)
done

lemma *card-Union-disjoint*:

finite C ==> (*ALL A*:*C*. *finite A*) ==>
(*ALL A*:*C*. *ALL B*:*C*. *A* ≠ *B* --> *A Int B* = {}) ==> *card* (*Union C*) = *setsum card C*
apply (*frule card-UN-disjoint* [*of C id*])
apply (*unfold Union-def id-def*, *assumption*+)
done

17.6.2 Cardinality of image

The image of a finite set can be expressed using *fold-image*.

lemma *image-eq-fold-image*:
finite A ==> *f* ‘ *A* = *fold-image* (*op Un*) (%*x*. {*f x*}) {} *A*
proof (*induct rule*: *finite-induct*)
case *empty* **then show** *?case* **by** *simp*
next
interpret *ab-semigroup-mult op Un*
proof qed *auto*
case *insert*
then show *?case* **by** *simp*
qed

lemma *card-image-le*: *finite A* ==> *card* (*f* ‘ *A*) <= *card A*
apply (*induct set*: *finite*)
apply *simp*
apply (*simp add*: *le-SucI finite-imageI card-insert-if*)
done

lemma *card-image*: *inj-on f A* ==> *card* (*f* ‘ *A*) = *card A*
by(*simp add*:*card-def setsum-reindex o-def del*:*setsum-constant*)

lemma *endo-inj-surj*: *finite A* ==> *f* ‘ *A* ⊆ *A* ==> *inj-on f A* ==> *f* ‘ *A* = *A*
by (*simp add*: *card-seteq card-image*)

lemma *eq-card-imp-inj-on*:
[| *finite A*; *card*(*f* ‘ *A*) = *card A* |] ==> *inj-on f A*
apply (*induct rule*:*finite-induct*)
apply *simp*
apply(*frule card-image-le*[**where** *f* = *f*])
apply(*simp add*:*card-insert-if split*:*if-splits*)
done

lemma *inj-on-iff-eq-card*:
finite A ==> *inj-on f A* = (*card*(*f* ‘ *A*) = *card A*)
by(*blast intro*: *card-image eq-card-imp-inj-on*)

lemma *card-inj-on-le*:
[|*inj-on f A*; *f* ‘ *A* ⊆ *B*; *finite B* |] ==> *card A* ≤ *card B*
apply (*subgoal-tac finite A*)

apply (*force intro*: *card-mono simp add*: *card-image* [*symmetric*])
apply (*blast intro*: *finite-imageD dest*: *finite-subset*)
done

lemma *card-bij-eq*:
[|*inj-on f A*; *f ' A* ⊆ *B*; *inj-on g B*; *g ' B* ⊆ *A*;
finite A; *finite B* |] ==> *card A* = *card B*
by (*auto intro*: *le-anti-sym card-inj-on-le*)

17.6.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
⟦ *finite A*; *ALL a*:*A*. *finite* (*B a*) ⟧
⟹ *card* (*SIGMA x*: *A*. *B x*) = (∑ *a*∈*A*. *card* (*B a*))
by(*simp add*:*card-def setsum-Sigma del*:*setsum-constant*)

lemma *card-cartesian-product*: *card* (*A* <*> *B*) = *card*(*A*) ∗ *card*(*B*)
apply (*cases finite A*)
apply (*cases finite B*)
apply (*auto simp add*: *card-eq-0-iff*
dest: *finite-cartesian-productD1 finite-cartesian-productD2*)
done

lemma *card-cartesian-product-singleton*: *card*({*x*} <*> *A*) = *card*(*A*)
by (*simp add*: *card-cartesian-product*)

17.6.4 Cardinality of sums

lemma *card-Plus*:
assumes *finite A* **and** *finite B*
shows *card* (*A* <+> *B*) = *card A* + *card B*
proof −
have *Inl*`*A* ∩ *Inr*`*B* = {} **by** *fast*
with *assms* **show** *?thesis*
unfolding *Plus-def*
by (*simp add*: *card-Un-disjoint card-image*)
qed

17.6.5 Cardinality of the Powerset

lemma *card-Pow*: *finite A* ==> *card* (*Pow A*) = *Suc* (*Suc 0*) ^ *card A*
apply (*induct set*: *finite*)
apply (*simp-all add*: *Pow-insert*)
apply (*subst card-Un-disjoint*, *blast*)
apply (*blast intro*: *finite-imageI*, *blast*)
apply (*subgoal-tac inj-on* (*insert x*) (*Pow F*))
apply (*simp add*: *card-image Pow-insert*)
apply (*unfold inj-on-def*)
apply (*blast elim*!: *equalityE*)
done

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma *dvd-partition*:
finite (*Union C*) ==>
ALL c : *C*. *k dvd card c* ==>
(*ALL c1*: *C*. *ALL c2*: *C*. *c1* ≠ *c2* −−> *c1 Int c2* = {}) ==>
k dvd card (*Union C*)
apply(*frule finite-UnionD*)
apply(*rotate-tac* −*1*)
apply (*induct set*: *finite*, *simp-all*, *clarify*)
apply (*subst card-Un-disjoint*)
apply (*auto simp add*: *dvd-add disjoint-eq-subset-Compl*)
done

17.6.6 Relating injectivity and surjectivity

lemma *finite-surj-inj*: *finite*(*A*) ⟹ *A* <= *f*‘*A* ⟹ *inj-on f A*
apply(*rule eq-card-imp-inj-on*, *assumption*)
apply(*frule finite-imageI*)
apply(*drule* (*1*) *card-seteq*)
apply(*erule card-image-le*)
apply *simp*
done

lemma *finite-UNIV-surj-inj*: **fixes** *f* :: ′*a* ⇒ ′*a*
shows *finite*(*UNIV*:: ′*a set*) ⟹ *surj f* ⟹ *inj f*
by (*blast intro*: *finite-surj-inj subset-UNIV dest*:*surj-range*)

lemma *finite-UNIV-inj-surj*: **fixes** *f* :: ′*a* ⇒ ′*a*
shows *finite*(*UNIV*:: ′*a set*) ⟹ *inj f* ⟹ *surj f*
by(*fastsimp simp*:*surj-def dest*!: *endo-inj-surj*)

corollary *infinite-UNIV-nat*: ~*finite*(*UNIV*::*nat set*)
proof
assume *finite*(*UNIV*::*nat set*)
with *finite-UNIV-inj-surj*[*of Suc*]
show *False* **by** *simp* (*blast dest*: *Suc-neq-Zero surjD*)
qed

lemma *infinite-UNIV-char-0*:
¬ *finite* (*UNIV*::′*a*::*semiring-char-0 set*)
proof
assume *finite* (*UNIV*::′*a set*)
with *subset-UNIV* **have** *finite* (*range of-nat*::′*a set*)
by (*rule finite-subset*)
moreover have *inj* (*of-nat*::*nat* ⇒ ′*a*)
by (*simp add*: *inj-on-def*)
ultimately have *finite* (*UNIV*::*nat set*)
by (*rule finite-imageD*)
then show *False*

by (*simp add*: *infinite-UNIV-nat*)
qed

17.7 A fold functional for non-empty sets

Does not require start value.

inductive
fold1Set :: ('*a* => '*a* => '*a*) => '*a set* => '*a* => *bool*
for *f* :: '*a* => '*a* => '*a*
where
fold1Set-insertI [*intro*]:
⟦ *fold-graph f a A x*; *a* ∉ *A* ⟧ ⟹ *fold1Set f* (*insert a A*) *x*

constdefs
fold1 :: ('*a* => '*a* => '*a*) => '*a set* => '*a*
fold1 f A == *THE x*. *fold1Set f A x*

lemma *fold1Set-nonempty*:
fold1Set f A x ⟹ *A* ≠ {}
by(*erule fold1Set.cases*, *simp-all*)

inductive-cases *empty-fold1SetE* [*elim*!]: *fold1Set f* {} *x*

inductive-cases *insert-fold1SetE* [*elim*!]: *fold1Set f* (*insert a X*) *x*

lemma *fold1Set-sing* [*iff*]: (*fold1Set f* {*a*} *b*) = (*a* = *b*)
by (*blast intro*: *fold-graph.intros elim*: *fold-graph.cases*)

lemma *fold1-singleton* [*simp*]: *fold1 f* {*a*} = *a*
by (*unfold fold1-def*) *blast*

lemma *finite-nonempty-imp-fold1Set*:
⟦ *finite A*; *A* ≠ {} ⟧ ⟹ *EX x*. *fold1Set f A x*
apply (*induct A rule*: *finite-induct*)
apply (*auto dest*: *finite-imp-fold-graph* [*of* - *f*])
done

First, some lemmas about *fold-graph*.

context *ab-semigroup-mult*
begin

lemma *fun-left-comm*: *fun-left-comm*(*op* ∗)
by *unfold-locales* (*simp add*: *mult-ac*)

lemma *fold-graph-insert-swap*:
assumes *fold*: *fold-graph times* (*b*::'*a*) *A y* **and** *b* ∉ *A*
shows *fold-graph times z* (*insert b A*) (*z* ∗ *y*)
proof −

```
  interpret fun-left-comm op ∗::'a ⇒ 'a ⇒ 'a by (rule fun-left-comm)
from assms show ?thesis
proof (induct rule: fold-graph.induct)
  case emptyI thus ?case by (force simp add: fold-insert-aux mult-commute)
next
  case (insertI x A y)
    have fold-graph times z (insert x (insert b A)) (x ∗ (z ∗ y))
      using insertI by force  — how does id get unfolded?
    thus ?case by (simp add: insert-commute mult-ac)
qed
qed

lemma fold-graph-permute-diff:
assumes fold: fold-graph times b A x
shows !!a. ⟦a ∈ A; b ∉ A⟧ ⟹ fold-graph times a (insert b (A−{a})) x
using fold
proof (induct rule: fold-graph.induct)
  case emptyI thus ?case by simp
next
  case (insertI x A y)
  have a = x ∨ a ∈ A using insertI by simp
  thus ?case
  proof
    assume a = x
    with insertI show ?thesis
      by (simp add: id-def [symmetric], blast intro: fold-graph-insert-swap)
  next
    assume ainA: a ∈ A
    hence fold-graph times a (insert x (insert b (A − {a}))) (x ∗ y)
      using insertI by force
    moreover
    have insert x (insert b (A − {a})) = insert b (insert x A − {a})
      using ainA insertI by blast
    ultimately show ?thesis by simp
  qed
qed

lemma fold1-eq-fold:
assumes finite A a ∉ A shows fold1 times (insert a A) = fold times a A
proof −
  interpret fun-left-comm op ∗::'a ⇒ 'a ⇒ 'a by (rule fun-left-comm)
  from assms show ?thesis
apply (simp add: fold1-def fold-def)
apply (rule the-equality)
apply (best intro: fold-graph-determ theI dest: finite-imp-fold-graph [of - times])
apply (rule sym, clarify)
apply (case-tac Aa=A)
 apply (best intro: the-equality fold-graph-determ)
apply (subgoal-tac fold-graph times a A x)
```

```
 apply (best intro: the-equality fold-graph-determ)
apply (subgoal-tac insert aa (Aa − {a}) = A)
 prefer 2 apply (blast elim: equalityE)
apply (auto dest: fold-graph-permute-diff [where a=a])
done
qed

lemma nonempty-iff: (A ≠ {}) = (∃x B. A = insert x B & x ∉ B)
apply safe
 apply simp
 apply (drule-tac x=x in spec)
 apply (drule-tac x=A−{x} in spec, auto)
done

lemma fold1-insert:
  assumes nonempty: A ≠ {} and A: finite A x ∉ A
  shows fold1 times (insert x A) = x ∗ fold1 times A
proof −
  interpret fun-left-comm op ∗::'a ⇒ 'a ⇒ 'a by (rule fun-left-comm)
  from nonempty obtain a A' where A = insert a A' & a ~: A'
    by (auto simp add: nonempty-iff)
  with A show ?thesis
    by (simp add: insert-commute [of x] fold1-eq-fold eq-commute)
qed

end

context ab-semigroup-idem-mult
begin

lemma fun-left-comm-idem: fun-left-comm-idem(op ∗)
apply unfold-locales
 apply (simp add: mult-ac)
apply (simp add: mult-idem mult-assoc[symmetric])
done


lemma fold1-insert-idem [simp]:
  assumes nonempty: A ≠ {} and A: finite A
  shows fold1 times (insert x A) = x ∗ fold1 times A
proof −
  interpret fun-left-comm-idem op ∗::'a ⇒ 'a ⇒ 'a
    by (rule fun-left-comm-idem)
  from nonempty obtain a A' where A': A = insert a A' & a ~: A'
    by (auto simp add: nonempty-iff)
  show ?thesis
  proof cases
    assume a = x
    thus ?thesis
```

```
    proof cases
      assume A' = {}
      with prems show ?thesis by (simp add: mult-idem)
    next
      assume A' ≠ {}
      with prems show ?thesis
        by (simp add: fold1-insert mult-assoc [symmetric] mult-idem)
    qed
  next
    assume a ≠ x
    with prems show ?thesis
      by (simp add: insert-commute fold1-eq-fold fold-insert-idem)
  qed
qed

lemma hom-fold1-commute:
assumes hom: !!x y. h (x * y) = h x * h y
and N: finite N N ≠ {} shows h (fold1 times N) = fold1 times (h ' N)
using N proof (induct rule: finite-ne-induct)
  case singleton thus ?case by simp
next
  case (insert n N)
  then have h (fold1 times (insert n N)) = h (n * fold1 times N) by simp
  also have ... = h n * h (fold1 times N) by(rule hom)
  also have h (fold1 times N) = fold1 times (h ' N) by(rule insert)
  also have times (h n) ... = fold1 times (insert (h n) (h ' N))
    using insert by(simp)
  also have insert (h n) (h ' N) = h ' insert n N by simp
  finally show ?case .
qed

end
```

Now the recursion rules for definitions:

```
lemma fold1-singleton-def: g = fold1 f ⟹ g {a} = a
by(simp add:fold1-singleton)

lemma (in ab-semigroup-mult) fold1-insert-def:
  ⟦ g = fold1 times; finite A; x ∉ A; A ≠ {} ⟧ ⟹ g (insert x A) = x * g A
by (simp add:fold1-insert)

lemma (in ab-semigroup-idem-mult) fold1-insert-idem-def:
  ⟦ g = fold1 times; finite A; A ≠ {} ⟧ ⟹ g (insert x A) = x * g A
by simp
```

17.7.1 Determinacy for *fold1Set*

```
declare
  empty-fold-graphE [rule del]  fold-graph.intros [rule del]
```

empty-fold1SetE [*rule del*] *insert-fold1SetE* [*rule del*]
— No more proofs involve these relations.

17.7.2 Lemmas about *fold1*

context *ab-semigroup-mult*
begin

lemma *fold1-Un*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹ *A Int B* = {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
using *A* **by** (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert mult-assoc*)

lemma *fold1-in*:
assumes *A*: *finite* (*A*) *A* ≠ {} **and** *elem*: ⋀*x y*. *x* ∗ *y* ∈ {*x*,*y*}
shows *fold1 times A* ∈ *A*
using *A*
proof (*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **using** *elem* **by** (*force simp add*:*fold1-insert*)
qed

end

lemma (**in** *ab-semigroup-idem-mult*) *fold1-Un2*:
assumes *A*: *finite A A* ≠ {}
shows *finite B* ⟹ *B* ≠ {} ⟹
fold1 times (*A Un B*) = *fold1 times A* ∗ *fold1 times B*
using *A*
proof(*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **by** (*simp add*: *mult-assoc*)
qed

17.7.3 Fold1 in lattices with *inf* and *sup*

As an application of *fold1* we define infimum and supremum in (not necessarily complete!) lattices over (non-empty) sets by means of *fold1*.

context *lower-semilattice*
begin

lemma *ab-semigroup-idem-mult-inf*:
ab-semigroup-idem-mult inf
proof qed (*rule inf-assoc inf-commute inf-idem*)+

lemma *below-fold1-iff*:
assumes *finite A* $A \neq \{\}$
shows $x \leq$ *fold1 inf A* $\longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
proof −
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
show *?thesis* **using** *assms* **by** (*induct rule*: *finite-ne-induct*) *simp-all*
qed

lemma *fold1-belowI*:
assumes *finite A*
and $a \in A$
shows *fold1 inf A* $\leq a$
proof −
from *assms* **have** $A \neq \{\}$ **by** *auto*
from ‹*finite A*› ‹$A \neq \{\}$› ‹$a \in A$› **show** *?thesis*
proof (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
case (*insert x F*)
from *insert*(*5*) **have** $a = x \vee a \in F$ **by** *simp*
thus *?case*
proof
assume $a = x$ **thus** *?thesis* **using** *insert*
by (*simp add*: *mult-ac*)
next
assume $a \in F$
hence *bel*: *fold1 inf F* $\leq a$ **by** (*rule insert*)
have *inf* (*fold1 inf* (*insert x F*)) $a =$ *inf x* (*inf* (*fold1 inf F*) *a*)
using *insert* **by** (*simp add*: *mult-ac*)
also have *inf* (*fold1 inf F*) $a =$ *fold1 inf F*
using *bel* **by** (*auto intro*: *antisym*)
also have *inf x* ... = *fold1 inf* (*insert x F*)
using *insert* **by** (*simp add*: *mult-ac*)
finally have *aux*: *inf* (*fold1 inf* (*insert x F*)) $a =$ *fold1 inf* (*insert x F*) **.**
moreover have *inf* (*fold1 inf* (*insert x F*)) $a \leq a$ **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
qed
qed

end

lemma (**in** *upper-semilattice*) *ab-semigroup-idem-mult-sup*:
ab-semigroup-idem-mult sup
by (*rule lower-semilattice.ab-semigroup-idem-mult-inf*)
(*rule dual-lattice*)

context *lattice*
begin

definition
Inf-fin :: *'a set* ⇒ *'a* ($\bigsqcap_{fin}$- [*900*] *900*)
where
Inf-fin = *fold1 inf*

definition
Sup-fin :: *'a set* ⇒ *'a* ($\bigsqcup_{fin}$- [*900*] *900*)
where
Sup-fin = *fold1 sup*

lemma *Inf-le-Sup* [*simp*]: ⟦ *finite A*; *A* ≠ {} ⟧ ⟹ $\bigsqcap_{fin} A \leq \bigsqcup_{fin} A$
apply(*unfold Sup-fin-def Inf-fin-def*)
apply(*subgoal-tac EX a*. *a*:*A*)
prefer *2* **apply** *blast*
apply(*erule exE*)
apply(*rule order-trans*)
apply(*erule* (*1*) *fold1-belowI*)
apply(*erule* (*1*) *lower-semilattice.fold1-belowI* [*OF dual-lattice*])
done

lemma *sup-Inf-absorb* [*simp*]:
finite A ⟹ *a* ∈ *A* ⟹ *sup a* ($\bigsqcap_{fin} A$) = *a*
apply(*subst sup-commute*)
apply(*simp add*: *Inf-fin-def sup-absorb2 fold1-belowI*)
done

lemma *inf-Sup-absorb* [*simp*]:
finite A ⟹ *a* ∈ *A* ⟹ *inf a* ($\bigsqcup_{fin} A$) = *a*
by (*simp add*: *Sup-fin-def inf-absorb1*
lower-semilattice.fold1-belowI [*OF dual-lattice*])

end

context *distrib-lattice*
begin

lemma *sup-Inf1-distrib*:
assumes *finite A*
and *A* ≠ {}
shows *sup x* ($\bigsqcap_{fin} A$) = $\bigsqcap_{fin}$ {*sup x a*|*a*. *a* ∈ *A*}
proof −
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
from *assms* **show** *?thesis*
by (*simp add*: *Inf-fin-def image-def*

hom-fold1-commute [**where** *h=sup x*, *OF sup-inf-distrib1*])
(*rule arg-cong* [**where** *f=fold1 inf*], *blast*)
qed

lemma *sup-Inf2-distrib*:
assumes *A*: *finite A A* $\neq$ {} **and** *B*: *finite B B* $\neq$ {}
shows *sup* ($\bigsqcap_{fin} A$) ($\bigsqcap_{fin} B$) = $\bigsqcap_{fin}$ {*sup a b*|*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*}
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by (*simp add*: *sup-Inf1-distrib* [*OF B*] *fold1-singleton-def* [*OF Inf-fin-def*])
next
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
case (*insert x A*)
have *finB*: *finite* {*sup x b* |*b*. *b* $\in$ *B*}
by(*rule finite-surj*[**where** *f* = *sup x*, *OF B*(*1*)], *auto*)
have *finAB*: *finite* {*sup a b* |*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*}
proof −
have {*sup a b* |*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*} = (*UN a*:*A*. *UN b*:*B*. {*sup a b*})
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: {*sup a b* |*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*} $\neq$ {} **using** *insert B* **by** *blast*
have *sup* ($\bigsqcap_{fin}$(*insert x A*)) ($\bigsqcap_{fin} B$) = *sup* (*inf x* ($\bigsqcap_{fin} A$)) ($\bigsqcap_{fin} B$)
using *insert* **by** (*simp add*: *fold1-insert-idem-def* [*OF Inf-fin-def*])
also have ... = *inf* (*sup x* ($\bigsqcap_{fin} B$)) (*sup* ($\bigsqcap_{fin} A$) ($\bigsqcap_{fin} B$)) **by**(*rule sup-inf-distrib2*)
also have ... = *inf* ($\bigsqcap_{fin}$ {*sup x b*|*b*. *b* $\in$ *B*}) ($\bigsqcap_{fin}$ {*sup a b*|*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*})
using *insert* **by**(*simp add*:*sup-Inf1-distrib*[*OF B*])
also have ... = $\bigsqcap_{fin}$({*sup x b* |*b*. *b* $\in$ *B*} $\cup$ {*sup a b* |*a b*. *a* $\in$ *A* $\wedge$ *b* $\in$ *B*})
(**is** - = $\bigsqcap_{fin}$ *?M*)
using *B insert*
by (*simp add*: *Inf-fin-def fold1-Un2* [*OF finB - finAB ne*])
also have *?M* = {*sup a b* |*a b*. *a* $\in$ *insert x A* $\wedge$ *b* $\in$ *B*}
by *blast*
finally show *?case* **.**
qed

lemma *inf-Sup1-distrib*:
assumes *finite A* **and** *A* $\neq$ {}
shows *inf x* ($\bigsqcup_{fin} A$) = $\bigsqcup_{fin}$ {*inf x a*|*a*. *a* $\in$ *A*}
proof −
interpret *ab-semigroup-idem-mult sup*
by (*rule ab-semigroup-idem-mult-sup*)
from *assms* **show** *?thesis*
by (*simp add*: *Sup-fin-def image-def hom-fold1-commute* [**where** *h=inf x*, *OF inf-sup-distrib1*])
(*rule arg-cong* [**where** *f=fold1 sup*], *blast*)
qed

lemma *inf-Sup2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows *inf* $(\bigsqcup_{fin} A)$ $(\bigsqcup_{fin} B) = \bigsqcup_{fin}\{inf\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by(*simp add*: *inf-Sup1-distrib* [*OF B*] *fold1-singleton-def* [*OF Sup-fin-def*])
next
case (*insert x A*)
have *finB*: *finite* $\{inf\ x\ b\ | b.\ b \in B\}$
by(*rule finite-surj*[**where** $f = \%b.\ inf\ x\ b$, *OF B*(*1*)], *auto*)
have *finAB*: *finite* $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\}$
proof −
have $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\} = (UN\ a{:}A.\ UN\ b{:}B.\ \{inf\ a\ b\})$
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: $\{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B* **by** *blast*
interpret *ab-semigroup-idem-mult sup*
by (*rule ab-semigroup-idem-mult-sup*)
have *inf* $(\bigsqcup_{fin}(insert\ x\ A))$ $(\bigsqcup_{fin} B) = inf\ (sup\ x\ (\bigsqcup_{fin} A))\ (\bigsqcup_{fin} B)$
using *insert* **by** (*simp add*: *fold1-insert-idem-def* [*OF Sup-fin-def*])
also have $\ldots = sup\ (inf\ x\ (\bigsqcup_{fin} B))\ (inf\ (\bigsqcup_{fin} A)\ (\bigsqcup_{fin} B))$ **by**(*rule inf-sup-distrib2*)
also have $\ldots = sup\ (\bigsqcup_{fin}\{inf\ x\ b | b.\ b \in B\})\ (\bigsqcup_{fin}\{inf\ a\ b | a\ b.\ a \in A \wedge b \in B\})$
using *insert* **by**(*simp add*:*inf-Sup1-distrib*[*OF B*])
also have $\ldots = \bigsqcup_{fin}(\{inf\ x\ b\ | b.\ b \in B\} \cup \{inf\ a\ b\ | a\ b.\ a \in A \wedge b \in B\})$
(**is** - = $\bigsqcup_{fin}$ *?M*)
using *B insert*
by (*simp add*: *Sup-fin-def fold1-Un2* [*OF finB - finAB ne*])
also have *?M* = $\{inf\ a\ b\ | a\ b.\ a \in insert\ x\ A \wedge b \in B\}$
by *blast*
finally show *?case* **.**
qed

end

context *complete-lattice*
begin

Coincidence on finite sets in complete lattices:

lemma *Inf-fin-Inf*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcap_{fin} A = Inf\ A$
proof −
interpret *ab-semigroup-idem-mult inf*
by (*rule ab-semigroup-idem-mult-inf*)
from *assms* **show** *?thesis*
unfolding *Inf-fin-def* **by** (*induct A set*: *finite*)

(*simp-all add*: *Inf-insert-simp*)
qed

lemma *Sup-fin-Sup*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcup_{fin} A = Sup\ A$
proof −
interpret *ab-semigroup-idem-mult sup*
by (*rule ab-semigroup-idem-mult-sup*)
from *assms* **show** *?thesis*
unfolding *Sup-fin-def* **by** (*induct A set*: *finite*)
(*simp-all add*: *Sup-insert-simp*)
qed

end

17.7.4 Fold1 in linear orders with *min* and *max*

As an application of *fold1* we define minimum and maximum in (not necessarily complete!) linear orders over (non-empty) sets by means of *fold1*.

context *linorder*
begin

lemma *ab-semigroup-idem-mult-min*:
ab-semigroup-idem-mult min
proof qed (*auto simp add*: *min-def*)

lemma *ab-semigroup-idem-mult-max*:
ab-semigroup-idem-mult max
proof qed (*auto simp add*: *max-def*)

lemma *min-lattice*:
lower-semilattice (*op* ≤) (*op* <) *min*
proof qed (*auto simp add*: *min-def*)

lemma *max-lattice*:
lower-semilattice (*op* ≥) (*op* >) *max*
proof qed (*auto simp add*: *max-def*)

lemma *dual-max*:
ord.max (*op* ≥) = *min*
by (*auto simp add*: *ord.max-def-raw min-def-raw expand-fun-eq*)

lemma *dual-min*:
ord.min (*op* ≥) = *max*
by (*auto simp add*: *ord.min-def-raw max-def-raw expand-fun-eq*)

lemma *strict-below-fold1-iff*:
assumes *finite A* **and** $A \neq \{\}$

shows $x < \mathit{fold1}\ \mathit{min}\ A \longleftrightarrow (\forall\, a{\in}A.\ x < a)$
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert*)
qed

lemma *fold1-below-iff*:
assumes *finite* A **and** $A \neq \{\}$
shows $\mathit{fold1}\ \mathit{min}\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert min-le-iff-disj*)
qed

lemma *fold1-strict-below-iff*:
assumes *finite* A **and** $A \neq \{\}$
shows $\mathit{fold1}\ \mathit{min}\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*induct rule*: *finite-ne-induct*)
(*simp-all add*: *fold1-insert min-less-iff-disj*)
qed

lemma *fold1-antimono*:
assumes $A \neq \{\}$ **and** $A \subseteq B$ **and** *finite* B
shows $\mathit{fold1}\ \mathit{min}\ B \leq \mathit{fold1}\ \mathit{min}\ A$
proof *cases*
assume $A = B$ **thus** *?thesis* **by** *simp*
next
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
assume $A \neq B$
have B: $B = A \cup (B{-}A)$ **using** ‹$A \subseteq B$› **by** *blast*
have $\mathit{fold1}\ \mathit{min}\ B = \mathit{fold1}\ \mathit{min}\ (A \cup (B{-}A))$ **by**(*subst* B)(*rule refl*)
also have $\ldots = \mathit{min}\ (\mathit{fold1}\ \mathit{min}\ A)\ (\mathit{fold1}\ \mathit{min}\ (B{-}A))$
proof −
have *finite* A **by**(*rule finite-subset*[*OF* ‹$A \subseteq B$› ‹*finite* B›])
moreover have *finite*$(B{-}A)$ **by**(*rule finite-Diff*[*OF* ‹*finite* B›])
moreover have $(B{-}A) \neq \{\}$ **using** *prems* **by** *blast*
moreover have $A\ \mathit{Int}\ (B{-}A) = \{\}$ **using** *prems* **by** *blast*
ultimately show *?thesis* **using** ‹$A \neq \{\}$› **by** (*rule-tac fold1-Un*)

qed
also have ... $\leq$ *fold1 min A* **by** (*simp add*: *min-le-iff-disj*)
finally show *?thesis* **.**
qed

definition
Min :: *'a set* $\Rightarrow$ *'a*
where
Min = *fold1 min*

definition
Max :: *'a set* $\Rightarrow$ *'a*
where
Max = *fold1 max*

lemmas *Min-singleton* [*simp*] = *fold1-singleton-def* [*OF Min-def*]
lemmas *Max-singleton* [*simp*] = *fold1-singleton-def* [*OF Max-def*]

lemma *Min-insert* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows *Min* (*insert x A*) = *min x* (*Min A*)
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis* **by** (*rule fold1-insert-idem-def* [*OF Min-def*])
qed

lemma *Max-insert* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows *Max* (*insert x A*) = *max x* (*Max A*)
proof −
interpret *ab-semigroup-idem-mult max*
by (*rule ab-semigroup-idem-mult-max*)
from *assms* **show** *?thesis* **by** (*rule fold1-insert-idem-def* [*OF Max-def*])
qed

lemma *Min-in* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows *Min A* $\in$ *A*
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms fold1-in* **show** *?thesis* **by** (*fastsimp simp*: *Min-def min-def*)
qed

lemma *Max-in* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows *Max A* $\in$ *A*
proof −

interpret *ab-semigroup-idem-mult max*
by (*rule ab-semigroup-idem-mult-max*)
from *assms fold1-in* [*of A*] **show** *?thesis* **by** (*fastsimp simp*: *Max-def max-def*)
qed

lemma *Min-Un*:
assumes *finite A* **and** $A \neq \{\}$ **and** *finite B* **and** $B \neq \{\}$
shows $Min\ (A \cup B) = min\ (Min\ A)\ (Min\ B)$
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-Un2*)
qed

lemma *Max-Un*:
assumes *finite A* **and** $A \neq \{\}$ **and** *finite B* **and** $B \neq \{\}$
shows $Max\ (A \cup B) = max\ (Max\ A)\ (Max\ B)$
proof −
interpret *ab-semigroup-idem-mult max*
by (*rule ab-semigroup-idem-mult-max*)
from *assms* **show** *?thesis*
by (*simp add*: *Max-def fold1-Un2*)
qed

lemma *hom-Min-commute*:
assumes $\bigwedge x\ y.\ h\ (min\ x\ y) = min\ (h\ x)\ (h\ y)$
and *finite N* **and** $N \neq \{\}$
shows $h\ (Min\ N) = Min\ (h\ `\ N)$
proof −
interpret *ab-semigroup-idem-mult min*
by (*rule ab-semigroup-idem-mult-min*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def hom-fold1-commute*)
qed

lemma *hom-Max-commute*:
assumes $\bigwedge x\ y.\ h\ (max\ x\ y) = max\ (h\ x)\ (h\ y)$
and *finite N* **and** $N \neq \{\}$
shows $h\ (Max\ N) = Max\ (h\ `\ N)$
proof −
interpret *ab-semigroup-idem-mult max*
by (*rule ab-semigroup-idem-mult-max*)
from *assms* **show** *?thesis*
by (*simp add*: *Max-def hom-fold1-commute* [*of h*])
qed

lemma *Min-le* [*simp*]:
assumes *finite A* **and** $x \in A$

shows *Min A* ≤ *x*
proof −
interpret *lower-semilattice op* ≤ *op* < *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def fold1-belowI*)
qed

lemma *Max-ge* [*simp*]:
assumes *finite A* **and** *x* ∈ *A*
shows *x* ≤ *Max A*
proof −
interpret *lower-semilattice op* ≥ *op* > *max*
by (*rule max-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Max-def fold1-belowI*)
qed

lemma *Min-ge-iff* [*simp*, *noatp*]:
assumes *finite A* **and** *A* ≠ {}
shows *x* ≤ *Min A* ⟷ (∀ *a*∈*A*. *x* ≤ *a*)
proof −
interpret *lower-semilattice op* ≤ *op* < *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def below-fold1-iff*)
qed

lemma *Max-le-iff* [*simp*, *noatp*]:
assumes *finite A* **and** *A* ≠ {}
shows *Max A* ≤ *x* ⟷ (∀ *a*∈*A*. *a* ≤ *x*)
proof −
interpret *lower-semilattice op* ≥ *op* > *max*
by (*rule max-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Max-def below-fold1-iff*)
qed

lemma *Min-gr-iff* [*simp*, *noatp*]:
assumes *finite A* **and** *A* ≠ {}
shows *x* < *Min A* ⟷ (∀ *a*∈*A*. *x* < *a*)
proof −
interpret *lower-semilattice op* ≤ *op* < *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def strict-below-fold1-iff*)
qed

lemma *Max-less-iff* [*simp*, *noatp*]:
assumes *finite A* **and** *A* ≠ {}
shows *Max A* < *x* ⟷ (∀ *a*∈*A*. *a* < *x*)
proof −
note *Max* = *Max-def*
interpret *linorder op* ≥ *op* >

by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max strict-below-fold1-iff* [*folded dual-max*])
qed

lemma *Min-le-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
proof −
interpret *lower-semilattice op* $\leq$ *op* $<$ *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-below-iff*)
qed

lemma *Max-ge-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x \leq Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x \leq a)$
proof −
note *Max* = *Max-def*
interpret *linorder op* $\geq$ *op* $>$
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-below-iff* [*folded dual-max*])
qed

lemma *Min-less-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
proof −
interpret *lower-semilattice op* $\leq$ *op* $<$ *min*
by (*rule min-lattice*)
from *assms* **show** *?thesis*
by (*simp add*: *Min-def fold1-strict-below-iff*)
qed

lemma *Max-gr-iff* [*noatp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $x < Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x < a)$
proof −
note *Max* = *Max-def*
interpret *linorder op* $\geq$ *op* $>$
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-strict-below-iff* [*folded dual-max*])
qed

lemma *Min-eqI*:
assumes *finite A*

assumes $\bigwedge y.\ y \in A \Longrightarrow y \geq x$
and $x \in A$
shows $Min\ A = x$
proof (*rule antisym*)
from ‹$x \in A$› **have** $A \neq \{\}$ **by** *auto*
with *assms* **show** $Min\ A \geq x$ **by** *simp*
next
from *assms* **show** $x \geq Min\ A$ **by** *simp*
qed

lemma *Max-eqI*:
assumes *finite A*
assumes $\bigwedge y.\ y \in A \Longrightarrow y \leq x$
and $x \in A$
shows $Max\ A = x$
proof (*rule antisym*)
from ‹$x \in A$› **have** $A \neq \{\}$ **by** *auto*
with *assms* **show** $Max\ A \leq x$ **by** *simp*
next
from *assms* **show** $x \leq Max\ A$ **by** *simp*
qed

lemma *Min-antimono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Min\ N \leq Min\ M$
proof −
interpret *distrib-lattice op* $\leq$ *op* $<$ *min max*
by (*rule distrib-lattice-min-max*)
from *assms* **show** *?thesis* **by** (*simp add*: *Min-def fold1-antimono*)
qed

lemma *Max-mono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Max\ M \leq Max\ N$
proof −
note $Max = Max\text{-}def$
interpret *linorder op* $\geq$ *op* $>$
by (*rule dual-linorder*)
from *assms* **show** *?thesis*
by (*simp add*: *Max fold1-antimono* [*folded dual-max*])
qed

lemma *finite-linorder-induct*[*consumes 1*, *case-names empty insert*]:
finite A $\Longrightarrow$ *P* $\{\}$ $\Longrightarrow$
(!!*A b*. *finite A* $\Longrightarrow$ *ALL a*:*A*. *a* $<$ *b* $\Longrightarrow$ *P A* $\Longrightarrow$ *P*(*insert b A*))
$\Longrightarrow$ *P A*
proof (*induct A rule*: *measure-induct-rule*[**where** *f*=*card*])
fix *A* :: *'a set*
assume *IH*: !! *B*. *card B* $<$ *card A* $\Longrightarrow$ *finite B* $\Longrightarrow$ *P* $\{\}$ $\Longrightarrow$

```
            (!!A b. finite A ⟹ (∀ a∈A. a<b) ⟹ P A ⟹ P (insert b A))
             ⟹ P B
  and finite A and P {}
  and step: !!A b. ⟦finite A; ∀ a∈A. a < b; P A⟧ ⟹ P (insert b A)
  show P A
  proof (cases A = {})
    assume A = {} thus P A using ‹P {}› by simp
  next
    let ?B = A - {Max A} let ?A = insert (Max A) ?B
    assume A ≠ {}
    with ‹finite A› have Max A : A by auto
    hence A: ?A = A using insert-Diff-single insert-absorb by auto
    note card-Diff1-less[OF ‹finite A› ‹Max A : A›]
    moreover have finite ?B using ‹finite A› by simp
    ultimately have P ?B using ‹P {}› step IH by blast
    moreover have ∀ a∈?B. a < Max A
      using Max-ge [OF ‹finite A›] by fastsimp
    ultimately show P A
      using A insert-Diff-single step[OF ‹finite ?B›] by fastsimp
  qed
qed

end

context ordered-ab-semigroup-add
begin

lemma add-Min-commute:
  fixes k
  assumes finite N and N ≠ {}
  shows k + Min N = Min {k + m | m. m ∈ N}
proof -
  have ⋀x y. k + min x y = min (k + x) (k + y)
    by (simp add: min-def not-le)
      (blast intro: antisym less-imp-le add-left-mono)
  with assms show ?thesis
    using hom-Min-commute [of plus k N]
    by simp (blast intro: arg-cong [where f = Min])
qed

lemma add-Max-commute:
  fixes k
  assumes finite N and N ≠ {}
  shows k + Max N = Max {k + m | m. m ∈ N}
proof -
  have ⋀x y. k + max x y = max (k + x) (k + y)
    by (simp add: max-def not-le)
      (blast intro: antisym less-imp-le add-left-mono)
  with assms show ?thesis
```

using *hom-Max-commute* [*of plus k N*]
by *simp* (*blast intro*: *arg-cong* [**where** *f* = *Max*])
qed

end

end

18 Relation: Relations

theory *Relation*
imports *Datatype Finite-Set*
begin

18.1 Definitions

definition
converse :: ('*a* * '*b*) *set* => ('*b* * '*a*) *set*
((-^-1) [*1000*] *999*) **where**
r^-1 == {(*y*, *x*). (*x*, *y*) : *r*}

notation (*xsymbols*)
converse ((-$^{-1}$) [*1000*] *999*)

definition
rel-comp :: [('*b* * '*c*) *set*, ('*a* * '*b*) *set*] => ('*a* * '*c*) *set*
(**infixr** *O* *75*) **where**
r O s == {(*x*,*z*). *EX y*. (*x*, *y*) : *s* & (*y*, *z*) : *r*}

definition
Image :: [('*a* * '*b*) *set*, '*a set*] => '*b set*
(**infixl** `` *90*) **where**
r `` *s* == {*y*. *EX x*:*s*. (*x*,*y*):*r*}

definition
Id :: ('*a* * '*a*) *set* **where** — the identity relation
Id == {*p*. *EX x*. *p* = (*x*,*x*)}

definition
Id-on :: '*a set* => ('*a* * '*a*) *set* **where** — diagonal: identity over a set
Id-on A == $\bigcup x \in A$. {(*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-on :: *['a set, ('a * 'a) set] => bool* **where** — reflexivity over a set
refl-on A r == r ⊆ A × A & (ALL x: A. (x,x) : r)

abbreviation
refl :: *('a * 'a) set => bool* **where** — reflexivity over a type
refl == refl-on UNIV

definition
sym :: *('a * 'a) set => bool* **where** — symmetry predicate
sym r == ALL x y. (x,y): r −−> (y,x): r

definition
antisym :: *('a * 'a) set => bool* **where** — antisymmetry predicate
antisym r == ALL x y. (x,y):r −−> (y,x):r −−> x=y

definition
trans :: *('a * 'a) set => bool* **where** — transitivity predicate
trans r == (ALL x y z. (x,y):r −−> (y,z):r −−> (x,z):r)

definition
irrefl :: *('a * 'a) set => bool* **where**
irrefl r ≡ ∀ x. (x,x) ∉ r

definition
total-on :: *'a set => ('a * 'a) set => bool* **where**
total-on A r ≡ ∀ x∈A.∀ y∈A. x≠y ⟶ (x,y)∈r ∨ (y,x)∈r

abbreviation *total ≡ total-on UNIV*

definition
single-valued :: *('a * 'b) set => bool* **where**
single-valued r == ALL x y. (x,y):r −−> (ALL z. (x,z):r −−> y=z)

definition
inv-image :: *('b * 'b) set => ('a => 'b) => ('a * 'a) set* **where**
inv-image r f == {(x, y). (f x, f y) : r}

18.2 The identity relation

lemma *IdI* [*intro*]: *(a, a) : Id*
by (*simp add*: *Id-def*)

lemma *IdE* [*elim*!]: $p : Id \Longrightarrow (!!x.\ p = (x,\ x) \Longrightarrow P) \Longrightarrow P$
by (*unfold Id-def*) (*iprover elim*: *CollectE*)

lemma *pair-in-Id-conv* [*iff*]: $((a,\ b) : Id) = (a = b)$
by (*unfold Id-def*) *blast*

lemma *refl-Id*: *refl Id*
by (*simp add*: *refl-on-def*)

lemma *antisym-Id*: *antisym Id*
— A strange result, since *Id* is also symmetric.
by (*simp add*: *antisym-def*)

lemma *sym-Id*: *sym Id*
by (*simp add*: *sym-def*)

lemma *trans-Id*: *trans Id*
by (*simp add*: *trans-def*)

18.3 Diagonal: identity over a set

lemma *Id-on-empty* [*simp*]: *Id-on* $\{\} = \{\}$
by (*simp add*: *Id-on-def*)

lemma *Id-on-eqI*: $a = b \Longrightarrow a : A \Longrightarrow (a,\ b) :$ *Id-on A*
by (*simp add*: *Id-on-def*)

lemma *Id-onI* [*intro*!,*noatp*]: $a : A \Longrightarrow (a,\ a) :$ *Id-on A*
by (*rule Id-on-eqI*) (*rule refl*)

lemma *Id-onE* [*elim*!]:
$c :$ *Id-on A* $\Longrightarrow (!!x.\ x : A \Longrightarrow c = (x,\ x) \Longrightarrow P) \Longrightarrow P$
— The general elimination rule.
by (*unfold Id-on-def*) (*iprover elim*!: *UN-E singletonE*)

lemma *Id-on-iff*: $((x,\ y) :$ *Id-on A*$) = (x = y\ \&\ x : A)$
by *blast*

lemma *Id-on-subset-Times*: *Id-on A* $\subseteq A \times A$
by *blast*

18.4 Composition of two relations

lemma *rel-compI* [*intro*]:
$(a,\ b) : s \Longrightarrow (b,\ c) : r \Longrightarrow (a,\ c) : r\ O\ s$
by (*unfold rel-comp-def*) *blast*

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s \Longrightarrow$
$(!!x\ y\ z.\ xz = (x,\ z) \Longrightarrow (x,\ y) : s \Longrightarrow (y,\ z) : r\ \Longrightarrow P) \Longrightarrow P$
by (*unfold rel-comp-def*) (*iprover elim*!: *CollectE splitE exE conjE*)

lemma *rel-compEpair*:
$(a,\ c) : r\ O\ s \Longrightarrow (!!y.\ (a,\ y) : s \Longrightarrow (y,\ c) : r \Longrightarrow P) \Longrightarrow P$
by (*iprover elim*: *rel-compE Pair-inject ssubst*)

lemma *R-O-Id* [*simp*]: $R\ O\ Id = R$
by *fast*

lemma *Id-O-R* [*simp*]: $Id\ O\ R = R$
by *fast*

lemma *rel-comp-empty1*[*simp*]: $\{\}\ O\ R = \{\}$
by *blast*

lemma *rel-comp-empty2*[*simp*]: $R\ O\ \{\} = \{\}$
by *blast*

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
by *blast*

lemma *trans-O-subset*: *trans* $r \Longrightarrow r\ O\ r \subseteq r$
by (*unfold trans-def*) *blast*

lemma *rel-comp-mono*: $r' \subseteq r \Longrightarrow s' \subseteq s \Longrightarrow (r'\ O\ s') \subseteq (r\ O\ s)$
by *blast*

lemma *rel-comp-subset-Sigma*:
$s \subseteq A \times B \Longrightarrow r \subseteq B \times C \Longrightarrow (r\ O\ s) \subseteq A \times C$
by *blast*

lemma *rel-comp-distrib*[*simp*]: $R\ O\ (S \cup T) = (R\ O\ S) \cup (R\ O\ T)$
by *auto*

lemma *rel-comp-distrib2*[*simp*]: $(S \cup T)\ O\ R = (S\ O\ R) \cup (T\ O\ R)$
by *auto*

18.5 Reflexivity

lemma *refl-onI*: $r \subseteq A \times A \Longrightarrow (!!x.\ x : A \Longrightarrow (x,\ x) : r) \Longrightarrow$ *refl-on* $A\ r$
by (*unfold refl-on-def*) (*iprover intro*!: *ballI*)

lemma *refl-onD*: *refl-on* $A\ r \Longrightarrow a : A \Longrightarrow (a,\ a) : r$
by (*unfold refl-on-def*) *blast*

lemma *refl-onD1*: *refl-on* $A\ r \Longrightarrow (x,\ y) : r \Longrightarrow x : A$
by (*unfold refl-on-def*) *blast*

lemma *refl-onD2*: *refl-on* $A\ r \Longrightarrow (x,\ y) : r \Longrightarrow y : A$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-Int*: *refl-on A r ==> refl-on B s ==> refl-on* $(A \cap B)$ $(r \cap s)$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-Un*: *refl-on A r ==> refl-on B s ==> refl-on* $(A \cup B)$ $(r \cup s)$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-INTER*:
ALL x:S. refl-on (A x) (r x) ==> refl-on (INTER S A) (INTER S r)
by (*unfold refl-on-def*) *fast*

lemma *refl-on-UNION*:
ALL x:S. refl-on (A x) (r x) $\Longrightarrow$ *refl-on (UNION S A) (UNION S r)*
by (*unfold refl-on-def*) *blast*

lemma *refl-on-empty*[*simp*]: *refl-on* {} {}
by(*simp add*:*refl-on-def*)

lemma *refl-on-Id-on*: *refl-on A* (*Id-on A*)
by (*rule refl-onI* [*OF Id-on-subset-Times Id-onI*])

18.6 Antisymmetry

lemma *antisymI*:
(!!*x y*. (*x*, *y*) : *r* ==> (*y*, *x*) : *r* ==> *x*=*y*) ==> *antisym r*
by (*unfold antisym-def*) *iprover*

lemma *antisymD*: *antisym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r* ==> *a* = *b*
by (*unfold antisym-def*) *iprover*

lemma *antisym-subset*: $r \subseteq s$ ==> *antisym s* ==> *antisym r*
by (*unfold antisym-def*) *blast*

lemma *antisym-empty* [*simp*]: *antisym* {}
by (*unfold antisym-def*) *blast*

lemma *antisym-Id-on* [*simp*]: *antisym* (*Id-on A*)
by (*unfold antisym-def*) *blast*

18.7 Symmetry

lemma *symI*: (!!*a b*. (*a*, *b*) : *r* ==> (*b*, *a*) : *r*) ==> *sym r*
by (*unfold sym-def*) *iprover*

lemma *symD*: *sym r* ==> (*a*, *b*) : *r* ==> (*b*, *a*) : *r*
by (*unfold sym-def*, *blast*)

lemma *sym-Int*: *sym r* ==> *sym s* ==> *sym* $(r \cap s)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-Un*: *sym r ==> sym s ==> sym* $(r \cup s)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-INTER*: *ALL x*:*S*. *sym* $(r\ x)$ *==> sym* $(INTER\ S\ r)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-UNION*: *ALL x*:*S*. *sym* $(r\ x)$ *==> sym* $(UNION\ S\ r)$
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-Id-on* [*simp*]: *sym* (*Id-on A*)
by (*rule symI*) *clarify*

18.8 Transitivity

lemma *transI*:
(!!*x y z*. $(x, y) : r$ *==>* $(y, z) : r$ *==>* $(x, z) : r$) *==> trans r*
by (*unfold trans-def*) *iprover*

lemma *transD*: *trans r ==>* $(a, b) : r$ *==>* $(b, c) : r$ *==>* $(a, c) : r$
by (*unfold trans-def*) *iprover*

lemma *trans-Int*: *trans r ==> trans s ==> trans* $(r \cap s)$
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-INTER*: *ALL x*:*S*. *trans* $(r\ x)$ *==> trans* $(INTER\ S\ r)$
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-Id-on* [*simp*]: *trans* (*Id-on A*)
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-diff-Id*: *trans r* $\Longrightarrow$ *antisym r* $\Longrightarrow$ *trans* $(r - Id)$
unfolding *antisym-def trans-def* **by** *blast*

18.9 Irreflexivity

lemma *irrefl-diff-Id*[*simp*]: *irrefl*$(r - Id)$
by(*simp add*:*irrefl-def*)

18.10 Totality

lemma *total-on-empty*[*simp*]: *total-on* $\{\}$ *r*
by(*simp add*:*total-on-def*)

lemma *total-on-diff-Id*[*simp*]: *total-on A* $(r - Id)$ = *total-on A r*
by(*simp add*: *total-on-def*)

18.11 Converse

lemma *converse-iff* [*iff*]: $((a,b): r\hat{}{-}1) = ((b,a) : r)$
by (*simp add*: *converse-def*)

lemma *converseI*[*sym*]: $(a,\ b) : r \Longrightarrow (b,\ a) : r\hat{}-1$
by (*simp add*: *converse-def*)

lemma *converseD*[*sym*]: $(a,b) : r\hat{}-1 \Longrightarrow (b,\ a) : r$
by (*simp add*: *converse-def*)

lemma *converseE* [*elim*!]:
$yx : r\hat{}-1 \Longrightarrow (!!x\ y.\ yx = (y,\ x) \Longrightarrow (x,\ y) : r \Longrightarrow P) \Longrightarrow P$
— More general than *converseD*, as it "splits" the member of the relation.
by (*unfold converse-def*) (*iprover elim*!: *CollectE splitE bexE*)

lemma *converse-converse* [*simp*]: $(r\hat{}-1)\hat{}-1 = r$
by (*unfold converse-def*) *blast*

lemma *converse-rel-comp*: $(r\ O\ s)\hat{}-1 = s\hat{}-1\ O\ r\hat{}-1$
by *blast*

lemma *converse-Int*: $(r \cap s)\hat{}-1 = r\hat{}-1 \cap s\hat{}-1$
by *blast*

lemma *converse-Un*: $(r \cup s)\hat{}-1 = r\hat{}-1 \cup s\hat{}-1$
by *blast*

lemma *converse-INTER*: $(INTER\ S\ r)\hat{}-1 = (INT\ x{:}S.\ (r\ x)\hat{}-1)$
by *fast*

lemma *converse-UNION*: $(UNION\ S\ r)\hat{}-1 = (UN\ x{:}S.\ (r\ x)\hat{}-1)$
by *blast*

lemma *converse-Id* [*simp*]: $Id\hat{}-1 = Id$
by *blast*

lemma *converse-Id-on* [*simp*]: $(Id\text{-}on\ A)\hat{}-1 = Id\text{-}on\ A$
by *blast*

lemma *refl-on-converse* [*simp*]: *refl-on A* (*converse r*) = *refl-on A r*
by (*unfold refl-on-def*) *auto*

lemma *sym-converse* [*simp*]: *sym* (*converse r*) = *sym r*
by (*unfold sym-def*) *blast*

lemma *antisym-converse* [*simp*]: *antisym* (*converse r*) = *antisym r*
by (*unfold antisym-def*) *blast*

lemma *trans-converse* [*simp*]: *trans* (*converse r*) = *trans r*
by (*unfold trans-def*) *blast*

lemma *sym-conv-converse-eq*: *sym r* = $(r\hat{}-1 = r)$

by (*unfold sym-def*) *fast*

lemma *sym-Un-converse*: *sym* (*r* ∪ *r*^−1)
by (*unfold sym-def*) *blast*

lemma *sym-Int-converse*: *sym* (*r* ∩ *r*^−1)
by (*unfold sym-def*) *blast*

lemma *total-on-converse*[*simp*]: *total-on A* (*r*^−1) = *total-on A r*
by (*auto simp*: *total-on-def*)

18.12 Domain

declare *Domain-def* [*noatp*]

lemma *Domain-iff*: (*a* : *Domain r*) = (*EX y*. (*a*, *y*) : *r*)
by (*unfold Domain-def*) *blast*

lemma *DomainI* [*intro*]: (*a*, *b*) : *r* ==> *a* : *Domain r*
by (*iprover intro*!: *iffD2* [*OF Domain-iff*])

lemma *DomainE* [*elim*!]:
a : *Domain r* ==> (!!*y*. (*a*, *y*) : *r* ==> *P*) ==> *P*
by (*iprover dest*!: *iffD1* [*OF Domain-iff*])

lemma *Domain-empty* [*simp*]: *Domain* {} = {}
by *blast*

lemma *Domain-insert*: *Domain* (*insert* (*a*, *b*) *r*) = *insert a* (*Domain r*)
by *blast*

lemma *Domain-Id* [*simp*]: *Domain Id* = *UNIV*
by *blast*

lemma *Domain-Id-on* [*simp*]: *Domain* (*Id-on A*) = *A*
by *blast*

lemma *Domain-Un-eq*: *Domain*(*A* ∪ *B*) = *Domain*(*A*) ∪ *Domain*(*B*)
by *blast*

lemma *Domain-Int-subset*: *Domain*(*A* ∩ *B*) ⊆ *Domain*(*A*) ∩ *Domain*(*B*)
by *blast*

lemma *Domain-Diff-subset*: *Domain*(*A*) − *Domain*(*B*) ⊆ *Domain*(*A* − *B*)
by *blast*

lemma *Domain-Union*: *Domain* (*Union S*) = (⋃ *A*∈*S*. *Domain A*)
by *blast*

lemma *Domain-converse*[*simp*]: *Domain*($r\hat{}-1$) = *Range r*
by(*auto simp*:*Range-def*)

lemma *Domain-mono*: $r \subseteq s$ ==> *Domain r* $\subseteq$ *Domain s*
by *blast*

lemma *fst-eq-Domain*: *fst* ' *R* = *Domain R*
by (*auto intro*!:*image-eqI*)

lemma *Domain-dprod* [*simp*]: *Domain* (*dprod r s*) = *uprod* (*Domain r*) (*Domain s*)
by *auto*

lemma *Domain-dsum* [*simp*]: *Domain* (*dsum r s*) = *usum* (*Domain r*) (*Domain s*)
by *auto*

18.13 Range

lemma *Range-iff*: (*a* : *Range r*) = (*EX y*. (*y*, *a*) : *r*)
by (*simp add*: *Domain-def Range-def*)

lemma *RangeI* [*intro*]: (*a*, *b*) : *r* ==> *b* : *Range r*
by (*unfold Range-def*) (*iprover intro*!: *converseI DomainI*)

lemma *RangeE* [*elim*!]: *b* : *Range r* ==> (!!*x*. (*x*, *b*) : *r* ==> *P*) ==> *P*
by (*unfold Range-def*) (*iprover elim*!: *DomainE dest*!: *converseD*)

lemma *Range-empty* [*simp*]: *Range* {} = {}
by *blast*

lemma *Range-insert*: *Range* (*insert* (*a*, *b*) *r*) = *insert b* (*Range r*)
by *blast*

lemma *Range-Id* [*simp*]: *Range Id* = *UNIV*
by *blast*

lemma *Range-Id-on* [*simp*]: *Range* (*Id-on A*) = *A*
by *auto*

lemma *Range-Un-eq*: *Range*($A \cup B$) = *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*) = (⋃ *A*∈*S*. *Range A*)
by *blast*

lemma *Range-converse*[*simp*]: *Range*(*r*^−1) = *Domain r*
by *blast*

lemma *snd-eq-Range*: *snd* ' *R* = *Range R*
by (*auto intro*!:*image-eqI*)

18.14 Field

lemma *mono-Field*: *r* ⊆ *s* ⟹ *Field r* ⊆ *Field s*
by(*auto simp*:*Field-def Domain-def Range-def*)

lemma *Field-empty*[*simp*]: *Field* {} = {}
by(*auto simp*:*Field-def*)

lemma *Field-insert*[*simp*]: *Field* (*insert* (*a*,*b*) *r*) = {*a*,*b*} ∪ *Field r*
by(*auto simp*:*Field-def*)

lemma *Field-Un*[*simp*]: *Field* (*r* ∪ *s*) = *Field r* ∪ *Field s*
by(*auto simp*:*Field-def*)

lemma *Field-Union*[*simp*]: *Field* (⋃ *R*) = ⋃(*Field* ' *R*)
by(*auto simp*:*Field-def*)

lemma *Field-converse*[*simp*]: *Field*(*r*^−1) = *Field r*
by(*auto simp*:*Field-def*)

18.15 Image of a set under a relation

declare *Image-def* [*noatp*]

lemma *Image-iff*: (*b* : *r*``*A*) = (*EX x*:*A*. (*x*, *b*) : *r*)
by (*simp add*: *Image-def*)

lemma *Image-singleton*: *r*``{*a*} = {*b*. (*a*, *b*) : *r*}
by (*simp add*: *Image-def*)

lemma *Image-singleton-iff* [*iff*]: (*b* : *r*``{*a*}) = ((*a*, *b*) : *r*)
by (*rule Image-iff* [*THEN trans*]) *simp*

lemma *ImageI* [*intro*,*noatp*]: (*a*, *b*) : *r* ==> *a* : *A* ==> *b* : *r*``*A*
by (*unfold Image-def*) *blast*

lemma *ImageE* [*elim*!]:
b : *r* `` *A* ==> (!!*x*. (*x*, *b*) : *r* ==> *x* : *A* ==> *P*) ==> *P*
by (*unfold Image-def*) (*iprover elim*!: *CollectE bexE*)

lemma *rev-ImageI*: *a* : *A* ==> (*a*, *b*) : *r* ==> *b* : *r* `` *A*

— This version's more effective when we already have the required a
by *blast*

lemma *Image-empty* [*simp*]: $R``\{\} = \{\}$
by *blast*

lemma *Image-Id* [*simp*]: $Id `` A = A$
by *blast*

lemma *Image-Id-on* [*simp*]: $Id\text{-}on\ A `` B = A \cap B$
by *blast*

lemma *Image-Int-subset*: $R `` (A \cap B) \subseteq R `` A \cap R `` B$
by *blast*

lemma *Image-Int-eq*:
$single\text{-}valued\ (converse\ R) ==> R `` (A \cap B) = R `` A \cap R `` B$
by (*simp add*: *single-valued-def*, *blast*)

lemma *Image-Un*: $R `` (A \cup B) = R `` A \cup R `` B$
by *blast*

lemma *Un-Image*: $(R \cup S) `` A = R `` A \cup S `` A$
by *blast*

lemma *Image-subset*: $r \subseteq A \times B ==> r``C \subseteq B$
by (*iprover intro*!: *subsetI elim*!: *ImageE dest*!: *subsetD SigmaD2*)

lemma *Image-eq-UN*: $r``B = (\bigcup y\in B.\ r``\{y\})$
— NOT suitable for rewriting
by *blast*

lemma *Image-mono*: $r' \subseteq r ==> A' \subseteq A ==> (r' `` A') \subseteq (r `` A)$
by *blast*

lemma *Image-UN*: $(r `` (UNION\ A\ B)) = (\bigcup x\in A.\ r `` (B\ x))$
by *blast*

lemma *Image-INT-subset*: $(r `` INTER\ A\ B) \subseteq (\bigcap x\in A.\ r `` (B\ x))$
by *blast*

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
$[|single\text{-}valued\ (r^{-1}); A\neq\{\}|] ==> r `` INTER\ A\ B = (\bigcap x\in A.\ r `` B\ x)$
apply (*rule equalityI*)
apply (*rule Image-INT-subset*)
apply (*simp add*: *single-valued-def*, *blast*)
done

lemma *Image-subset-eq*: $(r``A \subseteq B) = (A \subseteq - ((r\hat{}-1) `` (-B)))$
by *blast*

18.16 Single valued relations

lemma *single-valuedI*:
$ALL\ x\ y.\ (x,y):r \longrightarrow (ALL\ z.\ (x,z):r \longrightarrow y=z) \Longrightarrow single\text{-}valued\ r$
by (*unfold single-valued-def*)

lemma *single-valuedD*:
$single\text{-}valued\ r \Longrightarrow (x,\ y) : r \Longrightarrow (x,\ z) : r \Longrightarrow y = z$
by (*simp add*: *single-valued-def*)

lemma *single-valued-rel-comp*:
$single\text{-}valued\ r \Longrightarrow single\text{-}valued\ s \Longrightarrow single\text{-}valued\ (r\ O\ s)$
by (*unfold single-valued-def*) *blast*

lemma *single-valued-subset*:
$r \subseteq s \Longrightarrow single\text{-}valued\ s \Longrightarrow single\text{-}valued\ r$
by (*unfold single-valued-def*) *blast*

lemma *single-valued-Id* [*simp*]: *single-valued Id*
by (*unfold single-valued-def*) *blast*

lemma *single-valued-Id-on* [*simp*]: *single-valued* (*Id-on A*)
by (*unfold single-valued-def*) *blast*

18.17 Graphs given by *Collect*

lemma *Domain-Collect-split* [*simp*]: $Domain\{(x,y).\ P\ x\ y\} = \{x.\ EX\ y.\ P\ x\ y\}$
by *auto*

lemma *Range-Collect-split* [*simp*]: $Range\{(x,y).\ P\ x\ y\} = \{y.\ EX\ x.\ P\ x\ y\}$
by *auto*

lemma *Image-Collect-split* [*simp*]: $\{(x,y).\ P\ x\ y\} `` A = \{y.\ EX\ x{:}A.\ P\ x\ y\}$
by *auto*

18.18 Inverse image

lemma *sym-inv-image*: $sym\ r \Longrightarrow sym\ (inv\text{-}image\ r\ f)$
by (*unfold sym-def inv-image-def*) *blast*

lemma *trans-inv-image*: $trans\ r \Longrightarrow trans\ (inv\text{-}image\ r\ f)$
apply (*unfold trans-def inv-image-def*)
apply (*simp* (*no-asm*))
apply *blast*
done

18.19 Finiteness

lemma *finite-converse* [*iff*]: *finite* (*r*^−1) = *finite r*
apply (*subgoal-tac r*^−1 = (%(*x*,*y*). (*y*,*x*)) ‘*r*)
apply *simp*
apply (*rule iffI*)
apply (*erule finite-imageD* [*unfolded inj-on-def*])
apply (*simp split add*: *split-split*)
apply (*erule finite-imageI*)
apply (*simp add*: *converse-def image-def*, *auto*)
apply (*rule bexI*)
prefer *2* **apply** *assumption*
apply *simp*
done

Finiteness of transitive closure (Thanks to Sidi Ehmety)

lemma *finite-Field*: *finite r* ==> *finite* (*Field r*)
— A finite relation has a finite field (= *domain* ∪ *range*.
apply (*induct set*: *finite*)
apply (*auto simp add*: *Field-def Domain-insert Range-insert*)
done

18.20 Version of *lfp-induct* for binary relations

lemmas *lfp-induct2* =
lfp-induct-set [*of* (*a*, *b*), *split-format* (*complete*)]

end

19 Predicate: Predicates as relations and enumerations

theory *Predicate*
imports *Inductive Relation*
begin

notation
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅ - [*900*] *900*) **and**
Sup (⨆ - [*900*] *900*) **and**
top (⊤) **and**
bot (⊥)

19.1 Predicates as (complete) lattices

19.1.1 *op* ⊔ **on** *bool*

lemma *sup-boolI1*:
$P \Longrightarrow P \sqcup Q$
by (*simp add*: *sup-bool-eq*)

lemma *sup-boolI2*:
$Q \Longrightarrow P \sqcup Q$
by (*simp add*: *sup-bool-eq*)

lemma *sup-boolE*:
$P \sqcup Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
by (*auto simp add*: *sup-bool-eq*)

19.1.2 Equality and Subsets

lemma *pred-equals-eq*: $((\lambda x.\ x \in R) = (\lambda x.\ x \in S)) = (R = S)$
by (*simp add*: *mem-def*)

lemma *pred-equals-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) = (\lambda x\ y.\ (x,\ y) \in S)) = (R = S)$
by (*simp add*: *expand-fun-eq mem-def*)

lemma *pred-subset-eq*: $((\lambda x.\ x \in R) <= (\lambda x.\ x \in S)) = (R <= S)$
by (*simp add*: *mem-def*)

lemma *pred-subset-eq2* [*pred-set-conv*]: $((\lambda x\ y.\ (x,\ y) \in R) <= (\lambda x\ y.\ (x,\ y) \in S)) = (R <= S)$
by *fast*

19.1.3 Top and bottom elements

lemma *top1I* [*intro*!]: *top x*
by (*simp add*: *top-fun-eq top-bool-eq*)

lemma *top2I* [*intro*!]: *top x y*
by (*simp add*: *top-fun-eq top-bool-eq*)

lemma *bot1E* [*elim*!]: *bot x* $\Longrightarrow P$
by (*simp add*: *bot-fun-eq bot-bool-eq*)

lemma *bot2E* [*elim*!]: *bot x y* $\Longrightarrow P$
by (*simp add*: *bot-fun-eq bot-bool-eq*)

19.1.4 The empty set

lemma *bot-empty-eq*: $bot = (\lambda x.\ x \in \{\})$
by (*auto simp add*: *expand-fun-eq*)

lemma *bot-empty-eq2*: $bot = (\lambda x\ y.\ (x,\ y) \in \{\})$
by (*auto simp add*: *expand-fun-eq*)

19.1.5 Binary union

lemma *sup1-iff* [*simp*]: $sup\ A\ B\ x \longleftrightarrow A\ x \mid B\ x$
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup2-iff* [*simp*]: $sup\ A\ B\ x\ y \longleftrightarrow A\ x\ y \mid B\ x\ y$
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup-Un-eq* [*pred-set-conv*]: $sup\ (\lambda x.\ x \in R)\ (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cup S)$
by (*simp add*: *expand-fun-eq*)

lemma *sup-Un-eq2* [*pred-set-conv*]: $sup\ (\lambda x\ y.\ (x,\ y) \in R)\ (\lambda x\ y.\ (x,\ y) \in S) = (\lambda x\ y.\ (x,\ y) \in R \cup S)$
by (*simp add*: *expand-fun-eq*)

lemma *sup1I1* [*elim?*]: $A\ x \Longrightarrow sup\ A\ B\ x$
by *simp*

lemma *sup2I1* [*elim?*]: $A\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
by *simp*

lemma *sup1I2* [*elim?*]: $B\ x \Longrightarrow sup\ A\ B\ x$
by *simp*

lemma *sup2I2* [*elim?*]: $B\ x\ y \Longrightarrow sup\ A\ B\ x\ y$
by *simp*

Classical introduction rule: no commitment to A vs B.

lemma *sup1CI* [*intro!*]: $(\sim B\ x \Longrightarrow A\ x) \Longrightarrow sup\ A\ B\ x$
by *auto*

lemma *sup2CI* [*intro!*]: $(\sim B\ x\ y \Longrightarrow A\ x\ y) \Longrightarrow sup\ A\ B\ x\ y$
by *auto*

lemma *sup1E* [*elim!*]: $sup\ A\ B\ x \Longrightarrow (A\ x \Longrightarrow P) \Longrightarrow (B\ x \Longrightarrow P) \Longrightarrow P$
by *simp iprover*

lemma *sup2E* [*elim!*]: $sup\ A\ B\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow P) \Longrightarrow (B\ x\ y \Longrightarrow P) \Longrightarrow P$
by *simp iprover*

19.1.6 Binary intersection

lemma *inf1-iff* [*simp*]: $inf\ A\ B\ x \longleftrightarrow A\ x \wedge B\ x$

by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf2-iff* [*simp*]: *inf A B x y* ⟷ *A x y* ∧ *B x y*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf-Int-eq* [*pred-set-conv*]: *inf* (λ*x*. *x* ∈ *R*) (λ*x*. *x* ∈ *S*) = (λ*x*. *x* ∈ *R* ∩ *S*)
by (*simp add*: *expand-fun-eq*)

lemma *inf-Int-eq2* [*pred-set-conv*]: *inf* (λ*x y*. (*x*, *y*) ∈ *R*) (λ*x y*. (*x*, *y*) ∈ *S*) = (λ*x y*. (*x*, *y*) ∈ *R* ∩ *S*)
by (*simp add*: *expand-fun-eq*)

lemma *inf1I* [*intro*!]: *A x* ==> *B x* ==> *inf A B x*
by *simp*

lemma *inf2I* [*intro*!]: *A x y* ==> *B x y* ==> *inf A B x y*
by *simp*

lemma *inf1D1*: *inf A B x* ==> *A x*
by *simp*

lemma *inf2D1*: *inf A B x y* ==> *A x y*
by *simp*

lemma *inf1D2*: *inf A B x* ==> *B x*
by *simp*

lemma *inf2D2*: *inf A B x y* ==> *B x y*
by *simp*

lemma *inf1E* [*elim*!]: *inf A B x* ==> (*A x* ==> *B x* ==> *P*) ==> *P*
by *simp*

lemma *inf2E* [*elim*!]: *inf A B x y* ==> (*A x y* ==> *B x y* ==> *P*) ==> *P*
by *simp*

19.1.7 Unions of families

lemma *SUP1-iff* [*simp*]: (*SUP x*:*A*. *B x*) *b* = (*EX x*:*A*. *B x b*)
by (*simp add*: *SUPR-def Sup-fun-def Sup-bool-def*) *blast*

lemma *SUP2-iff* [*simp*]: (*SUP x*:*A*. *B x*) *b c* = (*EX x*:*A*. *B x b c*)
by (*simp add*: *SUPR-def Sup-fun-def Sup-bool-def*) *blast*

lemma *SUP1-I* [*intro*]: *a* : *A* ==> *B a b* ==> (*SUP x*:*A*. *B x*) *b*
by *auto*

lemma *SUP2-I* [*intro*]: *a* : *A* ==> *B a b c* ==> (*SUP x*:*A*. *B x*) *b c*
by *auto*

lemma *SUP1-E* [*elim*!]: (*SUP* x:*A*. *B* x) b ==> (!!x. x : *A* ==> *B* x b ==> *R*) ==> *R*
by *auto*

lemma *SUP2-E* [*elim*!]: (*SUP* x:*A*. *B* x) b c ==> (!!x. x : *A* ==> *B* x b c ==> *R*) ==> *R*
by *auto*

lemma *SUP-UN-eq*: (*SUP* i. (λx. x ∈ r i)) = (λx. x ∈ (*UN* i. r i))
by (*simp add*: *expand-fun-eq*)

lemma *SUP-UN-eq2*: (*SUP* i. (λx y. (x, y) ∈ r i)) = (λx y. (x, y) ∈ (*UN* i. r i))
by (*simp add*: *expand-fun-eq*)

19.1.8 Intersections of families

lemma *INF1-iff* [*simp*]: (*INF* x:*A*. *B* x) b = (*ALL* x:*A*. *B* x b)
by (*simp add*: *INFI-def Inf-fun-def Inf-bool-def*) *blast*

lemma *INF2-iff* [*simp*]: (*INF* x:*A*. *B* x) b c = (*ALL* x:*A*. *B* x b c)
by (*simp add*: *INFI-def Inf-fun-def Inf-bool-def*) *blast*

lemma *INF1-I* [*intro*!]: (!!x. x : *A* ==> *B* x b) ==> (*INF* x:*A*. *B* x) b
by *auto*

lemma *INF2-I* [*intro*!]: (!!x. x : *A* ==> *B* x b c) ==> (*INF* x:*A*. *B* x) b c
by *auto*

lemma *INF1-D* [*elim*]: (*INF* x:*A*. *B* x) b ==> a : *A* ==> *B* a b
by *auto*

lemma *INF2-D* [*elim*]: (*INF* x:*A*. *B* x) b c ==> a : *A* ==> *B* a b c
by *auto*

lemma *INF1-E* [*elim*]: (*INF* x:*A*. *B* x) b ==> (*B* a b ==> *R*) ==> (a ~: *A* ==> *R*) ==> *R*
by *auto*

lemma *INF2-E* [*elim*]: (*INF* x:*A*. *B* x) b c ==> (*B* a b c ==> *R*) ==> (a ~: *A* ==> *R*) ==> *R*
by *auto*

lemma *INF-INT-eq*: (*INF* i. (λx. x ∈ r i)) = (λx. x ∈ (*INT* i. r i))
by (*simp add*: *expand-fun-eq*)

lemma *INF-INT-eq2*: (*INF* i. (λx y. (x, y) ∈ r i)) = (λx y. (x, y) ∈ (*INT* i. r i))
by (*simp add*: *expand-fun-eq*)

19.2 Predicates as relations

19.2.1 Composition

inductive
pred-comp :: [*'b => 'c => bool*, *'a => 'b => bool*] => *'a => 'c => bool*
(**infixr** *OO 75*)
for *r* :: *'b => 'c => bool* **and** *s* :: *'a => 'b => bool*
where
pred-compI [*intro*]: *s a b ==> r b c ==> (r OO s) a c*

inductive-cases *pred-compE* [*elim!*]: *(r OO s) a c*

lemma *pred-comp-rel-comp-eq* [*pred-set-conv*]:
((λx y. (x, y) ∈ r) OO (λx y. (x, y) ∈ s)) = (λx y. (x, y) ∈ r O s)
by (*auto simp add*: *expand-fun-eq elim*: *pred-compE*)

19.2.2 Converse

inductive
conversep :: *('a => 'b => bool) => 'b => 'a => bool*
(*(-^--1) [1000] 1000*)
for *r* :: *'a => 'b => bool*
where
conversepI: *r a b ==> r^--1 b a*

notation (*xsymbols*)
conversep (*(-$^{-1-1}$) [1000] 1000*)

lemma *conversepD*:
assumes *ab*: *r^--1 a b*
shows *r b a* **using** *ab*
by *cases simp*

lemma *conversep-iff* [*iff*]: *r^--1 a b = r b a*
by (*iprover intro*: *conversepI dest*: *conversepD*)

lemma *conversep-converse-eq* [*pred-set-conv*]:
(λx y. (x, y) ∈ r)^--1 = (λx y. (x, y) ∈ r^-1)
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-conversep* [*simp*]: *(r^--1)^--1 = r*
by (*iprover intro*: *order-antisym conversepI dest*: *conversepD*)

lemma *converse-pred-comp*: *(r OO s)^--1 = s^--1 OO r^--1*
by (*iprover intro*: *order-antisym conversepI pred-compI*
elim: *pred-compE dest*: *conversepD*)

lemma *converse-meet*: *(inf r s)^--1 = inf r^--1 s^--1*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *converse-join*: (*sup r s*) ^−−1 = *sup r* ^−−1 *s* ^−−1
by (*simp add*: *sup-fun-eq sup-bool-eq*)
(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *conversep-noteq* [*simp*]: (*op* ~=) ^−−1 = *op* ~=
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-eq* [*simp*]: (*op* =) ^−−1 = *op* =
by (*auto simp add*: *expand-fun-eq*)

19.2.3 Domain

inductive
DomainP :: ('*a* => '*b* => *bool*) => '*a* => *bool*
for *r* :: '*a* => '*b* => *bool*
where
DomainPI [*intro*]: *r a b* ==> *DomainP r a*

inductive-cases *DomainPE* [*elim*!]: *DomainP r a*

lemma *DomainP-Domain-eq* [*pred-set-conv*]: *DomainP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Domain r*)
by (*blast intro*!: *Orderings.order-antisym predicate1I*)

19.2.4 Range

inductive
RangeP :: ('*a* => '*b* => *bool*) => '*b* => *bool*
for *r* :: '*a* => '*b* => *bool*
where
RangePI [*intro*]: *r a b* ==> *RangeP r b*

inductive-cases *RangePE* [*elim*!]: *RangeP r b*

lemma *RangeP-Range-eq* [*pred-set-conv*]: *RangeP* (λ*x y*. (*x*, *y*) ∈ *r*) = (λ*x*. *x* ∈ *Range r*)
by (*blast intro*!: *Orderings.order-antisym predicate1I*)

19.2.5 Inverse image

definition
inv-imagep :: ('*b* => '*b* => *bool*) => ('*a* => '*b*) => '*a* => '*a* => *bool* **where**
inv-imagep r f == %*x y*. *r* (*f x*) (*f y*)

lemma [*pred-set-conv*]: *inv-imagep* (λ*x y*. (*x*, *y*) ∈ *r*) *f* = (λ*x y*. (*x*, *y*) ∈ *inv-image r f*)
by (*simp add*: *inv-image-def inv-imagep-def*)

lemma *in-inv-imagep* [*simp*]: *inv-imagep r f x y = r (f x) (f y)*
by (*simp add*: *inv-imagep-def*)

19.2.6 Powerset

definition *Powp* :: (*'a ⇒ bool*) *⇒ 'a set ⇒ bool* **where**
Powp A == λB. ∀ x ∈ B. A x

lemma *Powp-Pow-eq* [*pred-set-conv*]: *Powp (λx. x ∈ A) = (λx. x ∈ Pow A)*
by (*auto simp add*: *Powp-def expand-fun-eq*)

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred pred-subset-eq*]

19.2.7 Properties of relations

abbreviation *antisymP* :: (*'a => 'a => bool*) *=> bool* **where**
antisymP r == antisym {(x, y). r x y}

abbreviation *transP* :: (*'a => 'a => bool*) *=> bool* **where**
transP r == trans {(x, y). r x y}

abbreviation *single-valuedP* :: (*'a => 'b => bool*) *=> bool* **where**
single-valuedP r == single-valued {(x, y). r x y}

19.3 Predicates as enumerations

19.3.1 The type of predicate enumerations (a monad)

datatype *'a pred = Pred 'a ⇒ bool*

primrec *eval* :: *'a pred ⇒ 'a ⇒ bool* **where**
eval-pred: *eval (Pred f) = f*

lemma *Pred-eval* [*simp*]:
Pred (eval x) = x
by (*cases x*) *simp*

lemma *eval-inject*: *eval x = eval y ⟷ x = y*
by (*cases x*) *auto*

definition *single* :: *'a ⇒ 'a pred* **where**
single x = Pred ((op =) x)

definition *bind* :: *'a pred ⇒ ('a ⇒ 'b pred) ⇒ 'b pred* (**infixl** *⨠= 70*) **where**
P ⨠= f = Pred (λx. (∃ y. eval P y ∧ eval (f y) x))

instantiation *pred* :: (*type*) *complete-lattice*
begin

definition

$P \leq Q \longleftrightarrow \mathit{eval}\ P \leq \mathit{eval}\ Q$

definition
$P < Q \longleftrightarrow \mathit{eval}\ P < \mathit{eval}\ Q$

definition
$\bot = \mathit{Pred}\ \bot$

definition
$\top = \mathit{Pred}\ \top$

definition
$P \sqcap Q = \mathit{Pred}\ (\mathit{eval}\ P \sqcap \mathit{eval}\ Q)$

definition
$P \sqcup Q = \mathit{Pred}\ (\mathit{eval}\ P \sqcup \mathit{eval}\ Q)$

definition
$\bigsqcap A = \mathit{Pred}\ (\mathit{INFI}\ A\ \mathit{eval})$

definition
$\bigsqcup A = \mathit{Pred}\ (\mathit{SUPR}\ A\ \mathit{eval})$

instance by *default*
(*auto simp add*: *less-eq-pred-def less-pred-def*
inf-pred-def sup-pred-def bot-pred-def top-pred-def
Inf-pred-def Sup-pred-def,
auto simp add: *le-fun-def less-fun-def le-bool-def less-bool-def*
eval-inject mem-def)

end

lemma *bind-bind*:
$(P \gg= Q) \gg= R = P \gg= (\lambda x.\ Q\ x \gg= R)$
by (*auto simp add*: *bind-def expand-fun-eq*)

lemma *bind-single*:
$P \gg= \mathit{single} = P$
by (*simp add*: *bind-def single-def*)

lemma *single-bind*:
$\mathit{single}\ x \gg= P = P\ x$
by (*simp add*: *bind-def single-def*)

lemma *bottom-bind*:
$\bot \gg= P = \bot$
by (*auto simp add*: *bot-pred-def bind-def expand-fun-eq*)

lemma *sup-bind*:

$(P \sqcup Q) \gg\!= R = P \gg\!= R \sqcup Q \gg\!= R$
by (*auto simp add*: *bind-def sup-pred-def expand-fun-eq*)

lemma *Sup-bind*: $(\bigsqcup A \gg\!= f) = \bigsqcup ((\lambda x.\ x \gg\!= f)\ `\ A)$
by (*auto simp add*: *bind-def Sup-pred-def expand-fun-eq*)

lemma *pred-iffI*:
assumes $\bigwedge x.\ eval\ A\ x \Longrightarrow eval\ B\ x$
and $\bigwedge x.\ eval\ B\ x \Longrightarrow eval\ A\ x$
shows $A = B$
proof −
from *assms* **have** $\bigwedge x.\ eval\ A\ x \longleftrightarrow eval\ B\ x$ **by** *blast*
then show *?thesis* **by** (*cases A*, *cases B*) (*simp add*: *expand-fun-eq*)
qed

lemma *singleI*: $eval\ (single\ x)\ x$
unfolding *single-def* **by** *simp*

lemma *singleI-unit*: $eval\ (single\ ())\ x$
by *simp* (*rule singleI*)

lemma *singleE*: $eval\ (single\ x)\ y \Longrightarrow (y = x \Longrightarrow P) \Longrightarrow P$
unfolding *single-def* **by** *simp*

lemma *singleE'*: $eval\ (single\ x)\ y \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$
by (*erule singleE*) *simp*

lemma *bindI*: $eval\ P\ x \Longrightarrow eval\ (Q\ x)\ y \Longrightarrow eval\ (P \gg\!= Q)\ y$
unfolding *bind-def* **by** *auto*

lemma *bindE*: $eval\ (R \gg\!= Q)\ y \Longrightarrow (\bigwedge x.\ eval\ R\ x \Longrightarrow eval\ (Q\ x)\ y \Longrightarrow P) \Longrightarrow P$
unfolding *bind-def* **by** *auto*

lemma *botE*: $eval\ \bot\ x \Longrightarrow P$
unfolding *bot-pred-def* **by** *auto*

lemma *supI1*: $eval\ A\ x \Longrightarrow eval\ (A \sqcup B)\ x$
unfolding *sup-pred-def* **by** *simp*

lemma *supI2*: $eval\ B\ x \Longrightarrow eval\ (A \sqcup B)\ x$
unfolding *sup-pred-def* **by** *simp*

lemma *supE*: $eval\ (A \sqcup B)\ x \Longrightarrow (eval\ A\ x \Longrightarrow P) \Longrightarrow (eval\ B\ x \Longrightarrow P) \Longrightarrow P$
unfolding *sup-pred-def* **by** *auto*

19.3.2 Derived operations

definition *if-pred* :: *bool* ⇒ *unit pred* **where**

if-pred-eq: *if-pred b* = (*if b then single* () *else* ⊥)

definition *not-pred* :: *unit pred* ⇒ *unit pred* **where**
not-pred-eq: *not-pred P* = (*if eval P* () *then* ⊥ *else single* ())

lemma *if-predI*: *P* ⟹ *eval* (*if-pred P*) ()
unfolding *if-pred-eq* **by** (*auto intro*: *singleI*)

lemma *if-predE*: *eval* (*if-pred b*) *x* ⟹ (*b* ⟹ *x* = () ⟹ *P*) ⟹ *P*
unfolding *if-pred-eq* **by** (*cases b*) (*auto elim*: *botE*)

lemma *not-predI*: ¬ *P* ⟹ *eval* (*not-pred* (*Pred* (λ*u*. *P*))) ()
unfolding *not-pred-eq eval-pred* **by** (*auto intro*: *singleI*)

lemma *not-predI'*: ¬ *eval P* () ⟹ *eval* (*not-pred P*) ()
unfolding *not-pred-eq* **by** (*auto intro*: *singleI*)

lemma *not-predE*: *eval* (*not-pred* (*Pred* (λ*u*. *P*))) *x* ⟹ (¬ *P* ⟹ *thesis*) ⟹ *thesis*
unfolding *not-pred-eq*
by (*auto split*: *split-if-asm elim*: *botE*)

lemma *not-predE'*: *eval* (*not-pred P*) *x* ⟹ (¬ *eval P x* ⟹ *thesis*) ⟹ *thesis*
unfolding *not-pred-eq*
by (*auto split*: *split-if-asm elim*: *botE*)

19.3.3 Implementation

datatype *'a seq* = *Empty* | *Insert 'a 'a pred* | *Join 'a pred 'a seq*

primrec *pred-of-seq* :: *'a seq* ⇒ *'a pred* **where**
pred-of-seq Empty = ⊥
| *pred-of-seq* (*Insert x P*) = *single x* ⊔ *P*
| *pred-of-seq* (*Join P xq*) = *P* ⊔ *pred-of-seq xq*

definition *Seq* :: (*unit* ⇒ *'a seq*) ⇒ *'a pred* **where**
Seq f = *pred-of-seq* (*f* ())

code-datatype *Seq*

primrec *member* :: *'a seq* ⇒ *'a* ⇒ *bool* **where**
member Empty x ⟷ *False*
| *member* (*Insert y P*) *x* ⟷ *x* = *y* ∨ *eval P x*
| *member* (*Join P xq*) *x* ⟷ *eval P x* ∨ *member xq x*

lemma *eval-member*:
member xq = *eval* (*pred-of-seq xq*)
proof (*induct xq*)
case *Empty* **show** *?case*

```
  by (auto simp add: expand-fun-eq elim: botE)
next
  case Insert show ?case
  by (auto simp add: expand-fun-eq elim: supE singleE intro: supI1 supI2 singleI)
next
  case Join then show ?case
  by (auto simp add: expand-fun-eq elim: supE intro: supI1 supI2)
qed

lemma eval-code [code]: eval (Seq f) = member (f ())
  unfolding Seq-def by (rule sym, rule eval-member)

lemma single-code [code]:
  single x = Seq (λu. Insert x ⊥)
  unfolding Seq-def by simp

primrec apply :: ('a ⇒ 'b Predicate.pred) ⇒ 'a seq ⇒ 'b seq where
    apply f Empty = Empty
  | apply f (Insert x P) = Join (f x) (Join (P ⤜ f) Empty)
  | apply f (Join P xq) = Join (P ⤜ f) (apply f xq)

lemma apply-bind:
  pred-of-seq (apply f xq) = pred-of-seq xq ⤜ f
proof (induct xq)
  case Empty show ?case
    by (simp add: bottom-bind)
next
  case Insert show ?case
    by (simp add: single-bind sup-bind)
next
  case Join then show ?case
    by (simp add: sup-bind)
qed

lemma bind-code [code]:
  Seq g ⤜ f = Seq (λu. apply f (g ()))
  unfolding Seq-def by (rule sym, rule apply-bind)

lemma bot-set-code [code]:
  ⊥ = Seq (λu. Empty)
  unfolding Seq-def by simp

primrec adjunct :: 'a pred ⇒ 'a seq ⇒ 'a seq where
    adjunct P Empty = Join P Empty
  | adjunct P (Insert x Q) = Insert x (Q ⊔ P)
  | adjunct P (Join Q xq) = Join Q (adjunct P xq)

lemma adjunct-sup:
  pred-of-seq (adjunct P xq) = P ⊔ pred-of-seq xq
```

by (*induct xq*) (*simp-all add*: *sup-assoc sup-commute sup-left-commute*)

lemma *sup-code* [*code*]:
Seq f ⊔ *Seq g* = *Seq* (λ*u*. *case f* ()
of Empty ⇒ *g* ()
| *Insert x P* ⇒ *Insert x* (*P* ⊔ *Seq g*)
| *Join P xq* ⇒ *adjunct* (*Seq g*) (*Join P xq*))
proof (*cases f* ())
case *Empty*
thus *?thesis*
unfolding *Seq-def* **by** (*simp add*: *sup-commute* [*of* ⊥] *sup-bot*)
next
case *Insert*
thus *?thesis*
unfolding *Seq-def* **by** (*simp add*: *sup-assoc*)
next
case *Join*
thus *?thesis*
unfolding *Seq-def*
by (*simp add*: *adjunct-sup sup-assoc sup-commute sup-left-commute*)
qed

primrec *contained* :: *'a seq* ⇒ *'a pred* ⇒ *bool* **where**
contained Empty Q ⟷ *True*
| *contained* (*Insert x P*) *Q* ⟷ *eval Q x* ∧ *P* ≤ *Q*
| *contained* (*Join P xq*) *Q* ⟷ *P* ≤ *Q* ∧ *contained xq Q*

lemma *single-less-eq-eval*:
single x ≤ *P* ⟷ *eval P x*
by (*auto simp add*: *single-def less-eq-pred-def mem-def*)

lemma *contained-less-eq*:
contained xq Q ⟷ *pred-of-seq xq* ≤ *Q*
by (*induct xq*) (*simp-all add*: *single-less-eq-eval*)

lemma *less-eq-pred-code* [*code*]:
Seq f ≤ *Q* = (*case f* ()
of Empty ⇒ *True*
| *Insert x P* ⇒ *eval Q x* ∧ *P* ≤ *Q*
| *Join P xq* ⇒ *P* ≤ *Q* ∧ *contained xq Q*)
by (*cases f* ())
(*simp-all add*: *Seq-def single-less-eq-eval contained-less-eq*)

lemma *eq-pred-code* [*code*]:
fixes *P Q* :: *'a*::*eq pred*
shows *eq-class.eq P Q* ⟷ *P* ≤ *Q* ∧ *Q* ≤ *P*
unfolding *eq* **by** *auto*

lemma [*code*]:

pred-case f P = f (eval P)
by (*cases P*) *simp*

lemma [*code*]:
pred-rec f P = f (eval P)
by (*cases P*) *simp*

no-notation
inf (**infixl** ⊓ *70*) **and**
sup (**infixl** ⊔ *65*) **and**
Inf (⨅- [*900*] *900*) **and**
Sup (⨆- [*900*] *900*) **and**
top (⊤) **and**
bot (⊥) **and**
bind (**infixl** ⤜ *70*)

hide (**open**) *type pred seq*
hide (**open**) *const Pred eval single bind if-pred not-pred*
Empty Insert Join Seq member pred-of-seq apply adjunct

end

20 Transitive-Closure: Reflexive and Transitive closure of a relation

theory *Transitive-Closure*
imports *Predicate*
uses ~~/*src/Provers/trancl.ML*
begin

rtrancl is reflexive/transitive closure, *trancl* is transitive closure, *reflcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

inductive-set
rtrancl :: ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ((-^*) [*1000*] *999*)
for *r* :: ('*a* × '*a*) *set*
where
rtrancl-refl [*intro*!, *Pure.intro*!, *simp*]: (*a*, *a*) : *r*^*
| *rtrancl-into-rtrancl* [*Pure.intro*]: (*a*, *b*) : *r*^* ==> (*b*, *c*) : *r* ==> (*a*, *c*) : *r*^*

inductive-set
trancl :: ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* ((-^+) [*1000*] *999*)
for *r* :: ('*a* × '*a*) *set*
where
r-into-trancl [*intro*, *Pure.intro*]: (*a*, *b*) : *r* ==> (*a*, *b*) : *r*^+
| *trancl-into-trancl* [*Pure.intro*]: (*a*, *b*) : *r*^+ ==> (*b*, *c*) : *r* ==> (*a*, *c*) : *r*^+

notation
rtranclp ((-^**) [*1000*] *1000*) **and**
tranclp ((-^++) [*1000*] *1000*)

abbreviation
reflclp :: ('*a* => '*a* => *bool*) => '*a* => '*a* => *bool* ((-^==) [*1000*] *1000*)
where
r^== == *sup r op* =

abbreviation
reflcl :: ('*a* × '*a*) *set* => ('*a* × '*a*) *set* ((-^=) [*1000*] *999*) **where**
r^= == *r* ∪ *Id*

notation (*xsymbols*)
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-⁺⁺) [*1000*] *1000*) **and**
reflclp ((-==) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-⁺) [*1000*] *999*) **and**
reflcl ((-=) [*1000*] *999*)

notation (*HTML* **output**)
rtranclp ((-**) [*1000*] *1000*) **and**
tranclp ((-⁺⁺) [*1000*] *1000*) **and**
reflclp ((-==) [*1000*] *1000*) **and**
rtrancl ((-*) [*1000*] *999*) **and**
trancl ((-⁺) [*1000*] *999*) **and**
reflcl ((-=) [*1000*] *999*)

20.1 Reflexive closure

lemma *refl-reflcl*[*simp*]: *refl*(*r*^=)
by(*simp add*:*refl-on-def*)

lemma *antisym-reflcl*[*simp*]: *antisym*(*r*^=) = *antisym r*
by(*simp add*:*antisym-def*)

lemma *trans-reflclI*[*simp*]: *trans r* ⟹ *trans*(*r*^=)
unfolding *trans-def* **by** *blast*

20.2 Reflexive-transitive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: (*sup* (λ*x y*. (*x*, *y*) ∈ *r*) *op* =) = (λ*x y*. (*x*, *y*) ∈ *r Un Id*)
by (*simp add*: *expand-fun-eq*)

lemma *r-into-rtrancl* [*intro*]: !!*p*. *p* ∈ *r* ==> *p* ∈ *r*^*
— *rtrancl* of *r* contains *r*
apply (*simp only*: *split-tupled-all*)

apply (*erule rtrancl-refl* [*THEN rtrancl-into-rtrancl*])
done

lemma *r-into-rtranclp* [*intro*]: *r x y ==> r^** x y*
— *rtrancl* of *r* contains *r*
by (*erule rtranclp.rtrancl-refl* [*THEN rtranclp.rtrancl-into-rtrancl*])

lemma *rtranclp-mono*: *r ≤ s ==> r^** ≤ s^***
— monotonicity of *rtrancl*
apply (*rule predicate2I*)
apply (*erule rtranclp.induct*)
apply (*rule-tac* [*2*] *rtranclp.rtrancl-into-rtrancl*, *blast+*)
done

lemmas *rtrancl-mono = rtranclp-mono* [*to-set*]

theorem *rtranclp-induct* [*consumes 1*, *case-names base step*, *induct set*: *rtranclp*]:
assumes *a*: *r^** a b*
and *cases*: *P a !!y z. [| r^** a y; r y z; P y |] ==> P z*
shows *P b*
proof −
from *a* **have** *a = a −−> P b*
by (*induct %x y. x = a −−> P y a b*) (*iprover intro*: *cases*)+
then show *?thesis* **by** *iprover*
qed

lemmas *rtrancl-induct* [*induct set*: *rtrancl*] = *rtranclp-induct* [*to-set*]

lemmas *rtranclp-induct2* =
rtranclp-induct[*of - (ax,ay) (bx,by)*, *split-rule*,
consumes 1, *case-names refl step*]

lemmas *rtrancl-induct2* =
rtrancl-induct[*of (ax,ay) (bx,by)*, *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *refl-rtrancl*: *refl* (*r^**)
by (*unfold refl-on-def*) *fast*

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* (*r^**)
proof (*rule transI*)
fix *x y z*
assume (*x*, *y*) ∈ *r**
assume (*y*, *z*) ∈ *r**
then show (*x*, *z*) ∈ *r**
proof *induct*
case *base*
show (*x*, *y*) ∈ *r** **by** *fact*

```
  next
    case (step u v)
    from ‹(x, u) ∈ r*› and ‹(u, v) ∈ r›
    show (x, v) ∈ r* ..
  qed
qed

lemmas rtrancl-trans = trans-rtrancl [THEN transD, standard]

lemma rtranclp-trans:
  assumes xy: r ^** x y
  and yz: r ^** y z
  shows r ^** x z using yz xy
  by induct iprover+

lemma rtranclE [cases set: rtrancl]:
  assumes major: (a::'a, b) : r ^*
  obtains
    (base) a = b
  | (step) y where (a, y) : r ^* and (y, b) : r
  — elimination of rtrancl – by induction on a special formula
  apply (subgoal-tac (a::'a) = b | (EX y. (a,y) : r ^* & (y,b) : r))
   apply (rule-tac [2] major [THEN rtrancl-induct])
    prefer 2 apply blast
   prefer 2 apply blast
  apply (erule asm-rl exE disjE conjE base step)+
  done

lemma rtrancl-Int-subset: [| Id ⊆ s; r O (r ^* ∩ s) ⊆ s|] ==> r ^* ⊆ s
  apply (rule subsetI)
  apply (rule-tac p=x in PairE, clarify)
  apply (erule rtrancl-induct, auto)
  done

lemma converse-rtranclp-into-rtranclp:
  r a b ⟹ r** b c ⟹ r** a c
  by (rule rtranclp-trans) iprover+

lemmas converse-rtrancl-into-rtrancl = converse-rtranclp-into-rtranclp [to-set]
```

More r^* equations and inclusions.

```
lemma rtranclp-idemp [simp]: (r ^**) ^** = r ^**
  apply (auto intro!: order-antisym)
  apply (erule rtranclp-induct)
   apply (rule rtranclp.rtrancl-refl)
  apply (blast intro: rtranclp-trans)
  done

lemmas rtrancl-idemp [simp] = rtranclp-idemp [to-set]
```

lemma *rtrancl-idemp-self-comp* [*simp*]: $R\hat{}* \; O \; R\hat{}* = R\hat{}*$
apply (*rule set-ext*)
apply (*simp only*: *split-tupled-all*)
apply (*blast intro*: *rtrancl-trans*)
done

lemma *rtrancl-subset-rtrancl*: $r \subseteq s\hat{}* \Longrightarrow r\hat{}* \subseteq s\hat{}*$
apply (*drule rtrancl-mono*)
apply *simp*
done

lemma *rtranclp-subset*: $R \leq S \Longrightarrow S \leq R\hat{}** \Longrightarrow S\hat{}** = R\hat{}**$
apply (*drule rtranclp-mono*)
apply (*drule rtranclp-mono*)
apply *simp*
done

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: $(sup \; (R\hat{}**) \; (S\hat{}**))\hat{}** = (sup \; R \; S)\hat{}**$
by (*blast intro*!: *rtranclp-subset intro*: *rtranclp-mono* [*THEN predicate2D*])

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

lemma *rtranclp-reflcl* [*simp*]: $(R\hat{}==)\hat{}** = R\hat{}**$
by (*blast intro*!: *rtranclp-subset*)

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflcl* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - Id)\hat{}* = r\hat{}*$
apply (*rule sym*)
apply (*rule rtrancl-subset*, *blast*, *clarify*)
apply (*rename-tac a b*)
apply (*case-tac a = b*)
apply *blast*
apply (*blast intro*!: *r-into-rtrancl*)
done

lemma *rtranclp-r-diff-Id*: $(inf \; r \; op \; \tilde{}=)\hat{}** = r\hat{}**$
apply (*rule sym*)
apply (*rule rtranclp-subset*)
apply *blast+*
done

theorem *rtranclp-converseD*:
assumes *r*: $(r\hat{}--1)\hat{}** \; x \; y$
shows $r\hat{}** \; y \; x$
proof −

```
  from r show ?thesis
    by induct (iprover intro: rtranclp-trans dest!: conversepD)+
qed

lemmas rtrancl-converseD = rtranclp-converseD [to-set]

theorem rtranclp-converseI:
  assumes r^** y x
  shows (r^--1)^** x y
  using assms
  by induct (iprover intro: rtranclp-trans conversepI)+

lemmas rtrancl-converseI = rtranclp-converseI [to-set]

lemma rtrancl-converse: (r^-1)^* = (r^*)^-1
  by (fast dest!: rtrancl-converseD intro!: rtrancl-converseI)

lemma sym-rtrancl: sym r ==> sym (r^*)
  by (simp only: sym-conv-converse-eq rtrancl-converse [symmetric])

theorem converse-rtranclp-induct[consumes 1]:
  assumes major: r^** a b
    and cases: P b !!y z. [| r y z; r^** z b; P z |] ==> P y
  shows P a
  using rtranclp-converseI [OF major]
  by induct (iprover intro: cases dest!: conversepD rtranclp-converseD)+

lemmas converse-rtrancl-induct = converse-rtranclp-induct [to-set]

lemmas converse-rtranclp-induct2 =
  converse-rtranclp-induct [of - (ax,ay) (bx,by), split-rule,
              consumes 1, case-names refl step]

lemmas converse-rtrancl-induct2 =
  converse-rtrancl-induct [of (ax,ay) (bx,by), split-format (complete),
              consumes 1, case-names refl step]

lemma converse-rtranclpE:
  assumes major: r^** x z
    and cases: x=z ==> P
      !!y. [| r x y; r^** y z |] ==> P
  shows P
  apply (subgoal-tac x = z | (EX y. r x y & r^** y z))
   apply (rule-tac [2] major [THEN converse-rtranclp-induct])
    prefer 2 apply iprover
   prefer 2 apply iprover
  apply (erule asm-rl exE disjE conjE cases)+
  done
```

lemmas *converse-rtranclE = converse-rtranclpE* [*to-set*]

lemmas *converse-rtranclpE2 = converse-rtranclpE* [*of* - (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemmas *converse-rtranclE2 = converse-rtranclE* [*of* (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemma *r-comp-rtrancl-eq*: *r O r^* = r^* O r*
by (*blast elim*: *rtranclE converse-rtranclE*
intro: *rtrancl-into-rtrancl converse-rtrancl-into-rtrancl*)

lemma *rtrancl-unfold*: *r^* = Id Un r O r^**
by (*auto intro*: *rtrancl-into-rtrancl elim*: *rtranclE*)

20.3 Transitive closure

lemma *trancl-mono*: !!*p*. *p ∈ r^+ ==> r ⊆ s ==> p ∈ s^+*
apply (*simp add*: *split-tupled-all*)
apply (*erule trancl.induct*)
apply (*iprover dest*: *subsetD*)+
done

lemma *r-into-trancl'*: !!*p*. *p : r ==> p : r^+*
by (*simp only*: *split-tupled-all*) (*erule r-into-trancl*)

Conversions between *trancl* and *rtrancl*.

lemma *tranclp-into-rtranclp*: *r^++ a b ==> r^** a b*
by (*erule tranclp.induct*) *iprover*+

lemmas *trancl-into-rtrancl = tranclp-into-rtranclp* [*to-set*]

lemma *rtranclp-into-tranclp1*: **assumes** *r*: *r^** a b*
shows !!*c*. *r b c ==> r^++ a c* **using** *r*
by *induct iprover*+

lemmas *rtrancl-into-trancl1 = rtranclp-into-tranclp1* [*to-set*]

lemma *rtranclp-into-tranclp2*: [| *r a b*; *r^** b c* |] *==> r^++ a c*
— intro rule from *r* and *rtrancl*
apply (*erule rtranclp.cases*)
apply *iprover*
apply (*rule rtranclp-trans* [*THEN rtranclp-into-tranclp1*])
apply (*simp* | *rule r-into-rtranclp*)+
done

lemmas *rtrancl-into-trancl2 = rtranclp-into-tranclp2* [*to-set*]

Nice induction rule for *trancl*

lemma *tranclp-induct* [*consumes 1*, *case-names base step*, *induct pred*: *tranclp*]:
assumes *r^++ a b*

and *cases*: !!*y*. *r a y* ==> *P y*
!!*y z*. *r*^++ *a y* ==> *r y z* ==> *P y* ==> *P z*
shows *P b*
proof −
from ‹*r*^++ *a b*› **have** *a* = *a* −−> *P b*
by (*induct* %*x y*. *x* = *a* −−> *P y a b*) (*iprover intro*: *cases*)+
then show *?thesis* **by** *iprover*
qed

lemmas *trancl-induct* [*induct set*: *trancl*] = *tranclp-induct* [*to-set*]

lemmas *tranclp-induct2* =
tranclp-induct [*of* - (*ax*,*ay*) (*bx*,*by*), *split-rule*,
consumes 1, *case-names base step*]

lemmas *trancl-induct2* =
trancl-induct [*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*),
consumes 1, *case-names base step*]

lemma *tranclp-trans-induct*:
assumes *major*: *r*^++ *x y*
and *cases*: !!*x y*. *r x y* ==> *P x y*
!!*x y z*. [| *r*^++ *x y*; *P x y*; *r*^++ *y z*; *P y z* |] ==> *P x z*
shows *P x y*
— Another induction rule for trancl, incorporating transitivity
by (*iprover intro*: *major* [*THEN tranclp-induct*] *cases*)

lemmas *trancl-trans-induct* = *tranclp-trans-induct* [*to-set*]

lemma *tranclE* [*cases set*: *trancl*]:
assumes (*a*, *b*) : *r*^+
obtains
(*base*) (*a*, *b*) : *r*
| (*step*) *c* **where** (*a*, *c*) : *r*^+ **and** (*c*, *b*) : *r*
using *assms* **by** *cases simp-all*

lemma *trancl-Int-subset*: [| *r* ⊆ *s*; *r O* (*r*^+ ∩ *s*) ⊆ *s*|] ==> *r*^+ ⊆ *s*
apply (*rule subsetI*)
apply (*rule-tac p* = *x* **in** *PairE*)
apply *clarify*
apply (*erule trancl-induct*)
apply *auto*
done

lemma *trancl-unfold*: *r*^+ = *r Un r O r*^+
by (*auto intro*: *trancl-into-trancl elim*: *tranclE*)

Transitivity of r^+

lemma *trans-trancl* [*simp*]: *trans* (*r*^+)

proof (*rule transI*)
fix *x y z*
assume $(x, y) \in r\hat{}+$
assume $(y, z) \in r\hat{}+$
then show $(x, z) \in r\hat{}+$
proof *induct*
case (*base u*)
from ‹$(x, y) \in r\hat{}+$› **and** ‹$(y, u) \in r$›
show $(x, u) \in r\hat{}+$ **..**
next
case (*step u v*)
from ‹$(x, u) \in r\hat{}+$› **and** ‹$(u, v) \in r$›
show $(x, v) \in r\hat{}+$ **..**
qed
qed

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*, *standard*]

lemma *tranclp-trans*:
assumes *xy*: $r\hat{}++\ x\ y$
and *yz*: $r\hat{}++\ y\ z$
shows $r\hat{}++\ x\ z$ **using** *yz xy*
by *induct iprover+*

lemma *trancl-id* [*simp*]: *trans* $r \Longrightarrow r\hat{}+ = r$
apply *auto*
apply (*erule trancl-induct*)
apply *assumption*
apply (*unfold trans-def*)
apply *blast*
done

lemma *rtranclp-tranclp-tranclp*:
assumes $r\hat{}**\ x\ y$
shows !!*z*. $r\hat{}++\ y\ z$ ==> $r\hat{}++\ x\ z$ **using** *assms*
by *induct* (*iprover intro*: *tranclp-trans*)+

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: *r a b* ==> $r\hat{}++\ b\ c$ ==> $r\hat{}++\ a\ c$
by (*erule tranclp-trans* [*OF tranclp.r-into-trancl*])

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *trancl-insert*:
(*insert* $(y, x)\ r)\hat{}+ = r\hat{}+ \cup \{(a, b).\ (a, y) \in r\hat{}* \wedge (x, b) \in r\hat{}*\}$
— primitive recursion for *trancl* over finite relations
apply (*rule equalityI*)
apply (*rule subsetI*)

```
 apply (simp only: split-tupled-all)
 apply (erule trancl-induct, blast)
apply (blast intro: rtrancl-into-trancl1 trancl-into-rtrancl r-into-trancl trancl-trans)
apply (rule subsetI)
apply (blast intro: trancl-mono rtrancl-mono
  [THEN [2] rev-subsetD] rtrancl-trancl-trancl rtrancl-into-trancl2)
done

lemma tranclp-converseI: (r^++)^--1 x y ==> (r^--1)^++ x y
apply (drule conversepD)
apply (erule tranclp-induct)
apply (iprover intro: conversepI tranclp-trans)+
done

lemmas trancl-converseI = tranclp-converseI [to-set]

lemma tranclp-converseD: (r^--1)^++ x y ==> (r^++)^--1 x y
apply (rule conversepI)
apply (erule tranclp-induct)
apply (iprover dest: conversepD intro: tranclp-trans)+
done

lemmas trancl-converseD = tranclp-converseD [to-set]

lemma tranclp-converse: (r^--1)^++ = (r^++)^--1
by (fastsimp simp add: expand-fun-eq
  intro!: tranclp-converseI dest!: tranclp-converseD)

lemmas trancl-converse = tranclp-converse [to-set]

lemma sym-trancl: sym r ==> sym (r^+)
by (simp only: sym-conv-converse-eq trancl-converse [symmetric])

lemma converse-tranclp-induct:
assumes major: r^++ a b
  and cases: !!y. r y b ==> P(y)
    !!y z.[| r y z;  r^++ z b;  P(z) |] ==> P(y)
shows P a
apply (rule tranclp-induct [OF tranclp-converseI, OF conversepI, OF major])
 apply (rule cases)
 apply (erule conversepD)
apply (blast intro: prems dest!: tranclp-converseD conversepD)
done

lemmas converse-trancl-induct = converse-tranclp-induct [to-set]

lemma tranclpD: R^++ x y ==> EX z. R x z ∧ R^** z y
apply (erule converse-tranclp-induct)
 apply auto
```

apply (*blast intro*: *rtranclp-trans*)
done

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *tranclD2*:
$(x, y) \in R^+ \Longrightarrow \exists z.\ (x, z) \in R^* \wedge (z, y) \in R$
by (*blast elim*: *tranclE intro*: *trancl-into-rtrancl*)

lemma *irrefl-tranclI*: $r\hat{}-1 \cap r\hat{}* = \{\} ==> (x, x) \notin r\hat{}+$
by (*blast elim*: *tranclE dest*: *trancl-into-rtrancl*)

lemma *irrefl-trancl-rD*: $!!X.\ ALL\ x.\ (x, x) \notin r\hat{}+ ==> (x, y) \in r ==> x \neq y$
by (*blast dest*: *r-into-trancl*)

lemma *trancl-subset-Sigma-aux*:
$(a, b) \in r\hat{}* ==> r \subseteq A \times A ==> a = b \vee a \in A$
by (*induct rule*: *rtrancl-induct*) *auto*

lemma *trancl-subset-Sigma*: $r \subseteq A \times A ==> r\hat{}+ \subseteq A \times A$
apply (*rule subsetI*)
apply (*simp only*: *split-tupled-all*)
apply (*erule tranclE*)
apply (*blast dest*!: *trancl-into-rtrancl trancl-subset-Sigma-aux*)+
done

lemma *reflcl-tranclp* [*simp*]: $(r\hat{}++)\hat{}== = r\hat{}**$
apply (*safe intro*!: *order-antisym*)
apply (*erule tranclp-into-rtranclp*)
apply (*blast elim*: *rtranclp.cases dest*: *rtranclp-into-tranclp1*)
done

lemmas *reflcl-trancl* [*simp*] = *reflcl-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r\hat{}=)\hat{}+ = r\hat{}*$
apply *safe*
apply (*drule trancl-into-rtrancl*, *simp*)
apply (*erule rtranclE*, *safe*)
apply (*rule r-into-trancl*, *simp*)
apply (*rule rtrancl-into-trancl1*)
apply (*erule rtrancl-reflcl* [*THEN equalityD2*, *THEN subsetD*], *fast*)
done

lemma *trancl-empty* [*simp*]: $\{\}\hat{}+ = \{\}$
by (*auto elim*: *trancl-induct*)

lemma *rtrancl-empty* [*simp*]: $\{\}\hat{}* = Id$
by (*rule subst* [*OF reflcl-trancl*]) *simp*

lemma *rtranclpD*: $R\hat{}** a\ b ==> a = b \vee a \neq b \wedge R\hat{}++ a\ b$
by (*force simp add*: *reflcl-tranclp* [*symmetric*] *simp del*: *reflcl-tranclp*)

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*:
$(x,y) \in R^* = (x=y \vee x\neq y \wedge (x,y) \in R^+)$
by (*fast elim*: *trancl-into-rtrancl dest*: *rtranclD*)

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: *Domain* $(R\hat{}*) = UNIV$
by *blast*

lemma *Range-rtrancl* [*simp*]: *Range* $(R\hat{}*) = UNIV$
by *blast*

lemma *rtrancl-Un-subset*: $(R\hat{}* \cup S\hat{}*) \subseteq (R\ Un\ S)\hat{}*$
by (*rule rtrancl-Un-rtrancl* [*THEN subst*]) *fast*

lemma *in-rtrancl-UnI*: $x \in R\hat{}* \vee x \in S\hat{}* ==> x \in (R \cup S)\hat{}*$
by (*blast intro*: *subsetD* [*OF rtrancl-Un-subset*])

lemma *trancl-domain* [*simp*]: *Domain* $(r\hat{}+) = Domain\ r$
by (*unfold Domain-def*) (*blast dest*: *tranclD*)

lemma *trancl-range* [*simp*]: *Range* $(r\hat{}+) = Range\ r$
unfolding *Range-def* **by**(*simp add*: *trancl-converse* [*symmetric*])

lemma *Not-Domain-rtrancl*:
$x \sim$: *Domain* $R ==> ((x,\ y) : R\hat{}*) = (x = y)$
apply *auto*
apply (*erule rev-mp*)
apply (*erule rtrancl-induct*)
apply *auto*
done

lemma *trancl-subset-Field2*: $r\hat{}+ <= Field\ r \times Field\ r$
apply *clarify*
apply (*erule trancl-induct*)
apply (*auto simp add*: *Field-def*)
done

lemma *finite-trancl*: *finite* $(r\hat{}+) =$ *finite* r
apply *auto*
prefer *2*
apply (*rule trancl-subset-Field2* [*THEN finite-subset*])
apply (*rule finite-SigmaI*)
prefer *3*
apply (*blast intro*: *r-into-trancl'* *finite-subset*)

apply (*auto simp add*: *finite-Field*)
done

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
⟦ *single-valued r*; (*x*,*y*) ∈ *r*^*; (*x*,*z*) ∈ *r*^* ⟧
⟹ (*y*,*z*) ∈ *r*^* ∨ (*z*,*y*) ∈ *r*^*
apply (*erule rtrancl-induct*)
apply *simp*
apply (*erule disjE*)
apply (*blast elim*:*converse-rtranclE dest*:*single-valuedD*)
apply(*blast intro*:*rtrancl-trans*)
done

lemma *r-r-into-trancl*: (*a*, *b*) ∈ *R* ==> (*b*, *c*) ∈ *R* ==> (*a*, *c*) ∈ *R*^+
by (*fast intro*: *trancl-trans*)

lemma *trancl-into-trancl* [*rule-format*]:
(*a*, *b*) ∈ *r*^+ ==> (*b*, *c*) ∈ *r* --> (*a*,*c*) ∈ *r*^+
apply (*erule trancl-induct*)
apply (*fast intro*: *r-r-into-trancl*)
apply (*fast intro*: *r-r-into-trancl trancl-trans*)
done

lemma *tranclp-rtranclp-tranclp*:
r^++ *a b* ==> *r*^** *b c* ==> *r*^++ *a c*
apply (*drule tranclpD*)
apply (*elim exE conjE*)
apply (*drule rtranclp-trans*, *assumption*)
apply (*drule rtranclp-into-tranclp2*, *assumption*, *assumption*)
done

lemmas *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl.trancl-into-trancl trancl-into-trancl2
rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

20.4 Setup of transitivity reasoner

ML ⟪

```
structure Trancl-Tac = Trancl-Tac-Fun (
  struct
    val r-into-trancl = @{thm trancl.r-into-trancl};
    val trancl-trans  = @{thm trancl-trans};
    val rtrancl-refl = @{thm rtrancl.rtrancl-refl};
    val r-into-rtrancl = @{thm r-into-rtrancl};
    val trancl-into-rtrancl = @{thm trancl-into-rtrancl};
    val rtrancl-trancl-trancl = @{thm rtrancl-trancl-trancl};
    val trancl-rtrancl-trancl = @{thm trancl-rtrancl-trancl};
    val rtrancl-trans = @{thm rtrancl-trans};

  fun decomp (@{const Trueprop} $ t) =
    let fun dec (Const (op :, -) $ (Const (Pair, -) $ a $ b) $ rel ) =
        let fun decr (Const (Transitive-Closure.rtrancl, - ) $ r) = (r,r*)
              | decr (Const (Transitive-Closure.trancl, - ) $ r)  = (r,r+)
              | decr r = (r,r);
            val (rel,r) = decr (Envir.beta-eta-contract rel);
        in SOME (a,b,rel,r) end
      | dec - =  NONE
    in dec t end
    | decomp - = NONE;

  end);

structure Tranclp-Tac = Trancl-Tac-Fun (
  struct
    val r-into-trancl = @{thm tranclp.r-into-trancl};
    val trancl-trans  = @{thm tranclp-trans};
    val rtrancl-refl = @{thm rtranclp.rtrancl-refl};
    val r-into-rtrancl = @{thm r-into-rtranclp};
    val trancl-into-rtrancl = @{thm tranclp-into-rtranclp};
    val rtrancl-trancl-trancl = @{thm rtranclp-tranclp-tranclp};
    val trancl-rtrancl-trancl = @{thm tranclp-rtranclp-tranclp};
    val rtrancl-trans = @{thm rtranclp-trans};

  fun decomp (@{const Trueprop} $ t) =
    let fun dec (rel $ a $ b) =
        let fun decr (Const (Transitive-Closure.rtranclp, - ) $ r) = (r,r*)
              | decr (Const (Transitive-Closure.tranclp, - ) $ r)  = (r,r+)
              | decr r = (r,r);
            val (rel,r) = decr rel;
        in SOME (a, b, rel, r) end
      | dec - =  NONE
    in dec t end
    | decomp - = NONE;
```

```
  end);
⟩⟩

declaration ⟨⟨ fn - =>
  Simplifier.map-ss (fn ss => ss
    addSolver (mk-solver Trancl (fn - => Trancl-Tac.trancl-tac))
    addSolver (mk-solver Rtrancl (fn - => Trancl-Tac.rtrancl-tac))
    addSolver (mk-solver Tranclp (fn - => Tranclp-Tac.trancl-tac))
    addSolver (mk-solver Rtranclp (fn - => Tranclp-Tac.rtrancl-tac)))
⟩⟩



method-setup trancl =
  ⟨⟨ Scan.succeed (K (SIMPLE-METHOD' Trancl-Tac.trancl-tac)) ⟩⟩
  ⟨⟨ simple transitivity reasoner ⟩⟩
method-setup rtrancl =
  ⟨⟨ Scan.succeed (K (SIMPLE-METHOD' Trancl-Tac.rtrancl-tac)) ⟩⟩
  ⟨⟨ simple transitivity reasoner ⟩⟩
method-setup tranclp =
  ⟨⟨ Scan.succeed (K (SIMPLE-METHOD' Tranclp-Tac.trancl-tac)) ⟩⟩
  ⟨⟨ simple transitivity reasoner (predicate version) ⟩⟩
method-setup rtranclp =
  ⟨⟨ Scan.succeed (K (SIMPLE-METHOD' Tranclp-Tac.rtrancl-tac)) ⟩⟩
  ⟨⟨ simple transitivity reasoner (predicate version) ⟩⟩

end
```

21 Wellfounded: Well-founded Recursion

```
theory Wellfounded
imports Finite-Set Transitive-Closure Nat
uses (Tools/function-package/size.ML)
begin
```

21.1 Basic Definitions

```
inductive
  wfrec-rel :: ('a * 'a) set => (('a => 'b) => 'a => 'b) => 'a => 'b => bool
  for R :: ('a * 'a) set
  and F :: ('a => 'b) => 'a => 'b
where
  wfrecI: ALL z. (z, x) : R --> wfrec-rel R F z (g z) ==>
          wfrec-rel R F x (F g x)

constdefs
  wf         :: ('a * 'a)set => bool
  wf(r) == (!P. (!x. (!y. (y,x):r --> P(y)) --> P(x)) --> (!x. P(x)))
```

wfP :: (*'a* => *'a* => *bool*) => *bool*
wfP r == *wf* {(*x*, *y*). *r x y*}

acyclic :: (*'a*∗*'a*)*set* => *bool*
acyclic r == !*x*. (*x*,*x*) ~: *r*^+

cut :: (*'a* => *'b*) => (*'a* ∗ *'a*)*set* => *'a* => *'a* => *'b*
cut f r x == (%*y*. *if* (*y*,*x*):*r then f y else undefined*)

adm-wf :: (*'a* ∗ *'a*) *set* => ((*'a* => *'b*) => *'a* => *'b*) => *bool*
adm-wf R F == *ALL f g x*.
(*ALL z*. (*z*, *x*) : *R* −−> *f z* = *g z*) −−> *F f x* = *F g x*

wfrec :: (*'a* ∗ *'a*) *set* => ((*'a* => *'b*) => *'a* => *'b*) => *'a* => *'b*
[*code del*]: *wfrec R F* == %*x*. *THE y*. *wfrec-rel R* (%*f x*. *F* (*cut f R x*) *x*) *x y*

abbreviation *acyclicP* :: (*'a* => *'a* => *bool*) => *bool* **where**
acyclicP r == *acyclic* {(*x*, *y*). *r x y*}

lemma *wfP-wf-eq* [*pred-set-conv*]: *wfP* (λ*x y*. (*x*, *y*) ∈ *r*) = *wf r*
by (*simp add*: *wfP-def*)

lemma *wfUNIVI*:
(!!*P x*. (*ALL x*. (*ALL y*. (*y*,*x*) : *r* −−> *P*(*y*)) −−> *P*(*x*)) ==> *P*(*x*)) ==> *wf*(*r*)
unfolding *wf-def* **by** *blast*

lemmas *wfPUNIVI* = *wfUNIVI* [*to-pred*]

Restriction to domain *A* and range *B*. If *r* is well-founded over their intersection, then *wf r*

lemma *wfI*:
[| *r* ⊆ *A* <∗> *B*;
!!*x P*. [|∀ *x*. (∀ *y*. (*y*,*x*) : *r* −−> *P y*) −−> *P x*; *x* : *A*; *x* : *B* |] ==> *P x* |]
==> *wf r*
unfolding *wf-def* **by** *blast*

lemma *wf-induct*:
[| *wf*(*r*);
!!*x*.[| *ALL y*. (*y*,*x*): *r* −−> *P*(*y*) |] ==> *P*(*x*)
|] ==> *P*(*a*)
unfolding *wf-def* **by** *blast*

lemmas *wfP-induct* = *wf-induct* [*to-pred*]

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *consumes 1*, *case-names less*, *induct set*: *wf*]

lemmas *wfP-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set*: *wfP*]

lemma *wf-not-sym*: *wf r* ==> (*a*, *x*) : *r* ==> (*x*, *a*) ~: *r*
by (*induct a arbitrary*: *x set*: *wf*) *blast*

lemmas *wf-asym* = *wf-not-sym* [*elim-format*]

lemma *wf-not-refl* [*simp*]: *wf r* ==> (*a*, *a*) ~: *r*
by (*blast elim*: *wf-asym*)

lemmas *wf-irrefl* = *wf-not-refl* [*elim-format*]

lemma *wf-wellorderI*:
assumes *wf*: *wf* {(*x*::'*a*::*ord*, *y*). *x* < *y*}
assumes *lin*: *OFCLASS*('*a*::*ord*, *linorder-class*)
shows *OFCLASS*('*a*::*ord*, *wellorder-class*)
using *lin* **by** (*rule wellorder-class.intro*)
(*blast intro*: *wellorder-axioms.intro wf-induct-rule* [*OF wf*])

lemma (**in** *wellorder*) *wf*:
wf {(*x*, *y*). *x* < *y*}
unfolding *wf-def* **by** (*blast intro*: *less-induct*)

21.2 Basic Results

transitive closure of a well-founded relation is well-founded!

lemma *wf-trancl*:
assumes *wf r*
shows *wf* (*r*^+)
proof −
{
fix *P* **and** *x*
assume *induct-step*: !!*x*. (!!*y*. (*y*, *x*) : *r*^+ ==> *P y*) ==> *P x*
have *P x*
proof (*rule induct-step*)
fix *y* **assume** (*y*, *x*) : *r*^+
with ‹*wf r*› **show** *P y*
proof (*induct x arbitrary*: *y*)
case (*less x*)
note *hyp* = ‹⋀*x'* *y'*. (*x'*, *x*) : *r* ==> (*y'*, *x'*) : *r*^+ ==> *P y'*›
from ‹(*y*, *x*) : *r*^+› **show** *P y*
proof *cases*
case *base*
show *P y*
proof (*rule induct-step*)
fix *y'* **assume** (*y'*, *y*) : *r*^+
with ‹(*y*, *x*) : *r*› **show** *P y'* **by** (*rule hyp* [*of y y'*])

```
        qed
      next
        case step
        then obtain x' where (x', x) : r and (y, x') : r^+ by simp
        then show P y by (rule hyp [of x' y])
      qed
    qed
  qed
 } then show ?thesis unfolding wf-def by blast
qed

lemmas wfP-trancl = wf-trancl [to-pred]

lemma wf-converse-trancl: wf (r^-1) ==> wf ((r^+)^-1)
 apply (subst trancl-converse [symmetric])
 apply (erule wf-trancl)
 done
```

Minimal-element characterization of well-foundedness

```
lemma wf-eq-minimal: wf r = (∀Q x. x∈Q --> (∃z∈Q. ∀y. (y,z)∈r -->
y∉Q))
proof (intro iffI strip)
 fix Q :: 'a set and x
 assume wf r and x ∈ Q
 then show ∃z∈Q. ∀y. (y, z) ∈ r ⟶ y ∉ Q
   unfolding wf-def
   by (blast dest: spec [of - %x. x∈Q ⟶ (∃z∈Q. ∀y. (y,z) ∈ r ⟶ y∉Q)])
next
 assume 1: ∀Q x. x ∈ Q ⟶ (∃z∈Q. ∀y. (y, z) ∈ r ⟶ y ∉ Q)
 show wf r
 proof (rule wfUNIVI)
   fix P :: 'a ⇒ bool and x
   assume 2: ∀x. (∀y. (y, x) ∈ r ⟶ P y) ⟶ P x
   let ?Q = {x. ¬ P x}
   have x ∈ ?Q ⟶ (∃z ∈ ?Q. ∀y. (y, z) ∈ r ⟶ y ∉ ?Q)
     by (rule 1 [THEN spec, THEN spec])
   then have ¬ P x ⟶ (∃z. ¬ P z ∧ (∀y. (y, z) ∈ r ⟶ P y)) by simp
   with 2 have ¬ P x ⟶ (∃z. ¬ P z ∧ P z) by fast
   then show P x by simp
 qed
qed

lemma wfE-min:
 assumes wf R x ∈ Q
 obtains z where z ∈ Q ⋀y. (y, z) ∈ R ⟹ y ∉ Q
 using assms unfolding wf-eq-minimal by blast

lemma wfI-min:
 (⋀x Q. x ∈ Q ⟹ ∃z∈Q. ∀y. (y, z) ∈ R ⟶ y ∉ Q)
```

$\Longrightarrow$ *wf R*
unfolding *wf-eq-minimal* **by** *blast*

lemmas *wfP-eq-minimal* = *wf-eq-minimal* [*to-pred*]

Well-foundedness of subsets

lemma *wf-subset*: [| *wf*(*r*); *p*<=*r* |] ==> *wf*(*p*)
apply (*simp* (*no-asm-use*) *add*: *wf-eq-minimal*)
apply *fast*
done

lemmas *wfP-subset* = *wf-subset* [*to-pred*]

Well-foundedness of the empty relation

lemma *wf-empty* [*iff*]: *wf*({})
by (*simp add*: *wf-def*)

lemmas *wfP-empty* [*iff*] =
wf-empty [*to-pred bot-empty-eq2*, *simplified bot-fun-eq bot-bool-eq*]

lemma *wf-Int1*: *wf r* ==> *wf* (*r Int r'*)
apply (*erule wf-subset*)
apply (*rule Int-lower1*)
done

lemma *wf-Int2*: *wf r* ==> *wf* (*r' Int r*)
apply (*erule wf-subset*)
apply (*rule Int-lower2*)
done

Well-foundedness of insert

lemma *wf-insert* [*iff*]: *wf*(*insert* (*y*,*x*) *r*) = (*wf*(*r*) & (*x*,*y*) ~: *r*^*)
apply (*rule iffI*)
apply (*blast elim*: *wf-trancl* [*THEN wf-irrefl*]
intro: *rtrancl-into-trancl1 wf-subset*
rtrancl-mono [*THEN* [*2*] *rev-subsetD*])
apply (*simp add*: *wf-eq-minimal*, *safe*)
apply (*rule allE*, *assumption*, *erule impE*, *blast*)
apply (*erule bexE*)
apply (*rename-tac a*, *case-tac a* = *x*)
prefer *2*
apply *blast*
apply (*case-tac y*:*Q*)
prefer *2* **apply** *blast*
apply (*rule-tac x* = {*z*. *z*:*Q* & (*z*,*y*) : *r*^*} **in** *allE*)
apply *assumption*
apply (*erule-tac V* = *ALL Q*. (*EX x*. *x* : *Q*) −−> *?P Q* **in** *thin-rl*)
— essential for speed

Blast with new substOccur fails

apply (*fast intro*: *converse-rtrancl-into-rtrancl*)
done

Well-foundedness of image

lemma *wf-prod-fun-image*: [| *wf r*; *inj f* |] ==> *wf*(*prod-fun f f* ‘ *r*)
apply (*simp only*: *wf-eq-minimal*, *clarify*)
apply (*case-tac EX p*. *f p* : *Q*)
apply (*erule-tac x* = {*p*. *f p* : *Q*} **in** *allE*)
apply (*fast dest*: *inj-onD*, *blast*)
done

21.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:
assumes *wf R wf S*
assumes *S O R* ⊆ *R*
shows *wf* (*R* ∪ *S*)
proof (*rule wfI-min*)
fix *x* :: *'a* **and** *Q*
let *?Q'* = {*x* ∈ *Q*. ∀ *y*. (*y*, *x*) ∈ *R* ⟶ *y* ∉ *Q*}
assume *x* ∈ *Q*
obtain *a* **where** *a* ∈ *?Q'*
by (*rule wfE-min* [*OF* ‹*wf R*› ‹*x* ∈ *Q*›]) *blast*
with ‹*wf S*›
obtain *z* **where** *z* ∈ *?Q'* **and** *zmin*: ⋀*y*. (*y*, *z*) ∈ *S* ⟹ *y* ∉ *?Q'* **by** (*erule wfE-min*)
{
fix *y* **assume** (*y*, *z*) ∈ *S*
then have *y* ∉ *?Q'* **by** (*rule zmin*)

have *y* ∉ *Q*
proof
assume *y* ∈ *Q*
with ‹*y* ∉ *?Q'*›
obtain *w* **where** (*w*, *y*) ∈ *R* **and** *w* ∈ *Q* **by** *auto*
from ‹(*w*, *y*) ∈ *R*› ‹(*y*, *z*) ∈ *S*› **have** (*w*, *z*) ∈ *S O R* **by** (*rule rel-compI*)
with ‹*S O R* ⊆ *R*› **have** (*w*, *z*) ∈ *R* **..**
with ‹*z* ∈ *?Q'*› **have** *w* ∉ *Q* **by** *blast*
with ‹*w* ∈ *Q*› **show** *False* **by** *contradiction*
qed
}
with ‹*z* ∈ *?Q'*› **show** ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *R* ∪ *S* ⟶ *y* ∉ *Q* **by** *blast*
qed

Well-foundedness of indexed union with disjoint domains and ranges

lemma *wf-UN*: [| *ALL i*:*I*. *wf*(*r i*);
ALL i:*I*. *ALL j*:*I*. *r i* ~= *r j* --> *Domain*(*r i*) *Int Range*(*r j*) = {}
|] ==> *wf*(*UN i*:*I*. *r i*)

apply (*simp only*: *wf-eq-minimal*, *clarify*)
apply (*rename-tac A a*, *case-tac EX i:I. EX a:A. EX b:A.* (*b*,*a*) : *r i*)
prefer *2*
apply *force*
apply *clarify*
apply (*drule bspec*, *assumption*)
apply (*erule-tac x*={*a. a*:*A* & (*EX b*:*A.* (*b*,*a*) : *r i*) } **in** *allE*)
apply (*blast elim*!: *allE*)
done

lemmas *wfP-SUP* = *wf-UN* [**where** *I*=*UNIV* **and** *r*=λ*i.* {(*x*, *y*). *r i x y*},
to-pred SUP-UN-eq2 bot-empty-eq pred-equals-eq, *simplified*, *standard*]

lemma *wf-Union*:
[| *ALL r*:*R. wf r*;
ALL r:*R. ALL s*:*R. r* ~= *s* −−> *Domain r Int Range s* = {}
|] ==> *wf*(*Union R*)
apply (*simp add*: *Union-def*)
apply (*blast intro*: *wf-UN*)
done

lemma *wf-Un*:
[| *wf r*; *wf s*; *Domain r Int Range s* = {} |] ==> *wf*(*r Un s*)
using *wf-union-compatible*[*of s r*]
by (*auto simp*: *Un-ac*)

lemma *wf-union-merge*:
wf (*R* ∪ *S*) = *wf* (*R O R* ∪ *R O S* ∪ *S*) (**is** *wf ?A* = *wf ?B*)
proof
assume *wf ?A*
with *wf-trancl* **have** *wfT*: *wf* (*?A* ^+) **.**
moreover have *?B* ⊆ *?A* ^+
by (*subst trancl-unfold*, *subst trancl-unfold*) *blast*
ultimately show *wf ?B* **by** (*rule wf-subset*)
next
assume *wf ?B*

show *wf ?A*
proof (*rule wfI-min*)
fix *Q* :: *'a set* **and** *x*
assume *x* ∈ *Q*

with ‹*wf ?B*›
obtain *z* **where** *z* ∈ *Q* **and** ⋀*y.* (*y*, *z*) ∈ *?B* ⟹ *y* ∉ *Q*
by (*erule wfE-min*)
then have *A1*: ⋀*y.* (*y*, *z*) ∈ *R O R* ⟹ *y* ∉ *Q*
and *A2*: ⋀*y.* (*y*, *z*) ∈ *R O S* ⟹ *y* ∉ *Q*
and *A3*: ⋀*y.* (*y*, *z*) ∈ *S* ⟹ *y* ∉ *Q*

```
    by auto

  show ∃z∈Q. ∀y. (y, z) ∈ ?A ⟶ y ∉ Q
  proof (cases ∀y. (y, z) ∈ R ⟶ y ∉ Q)
    case True
    with ‹z ∈ Q› A3 show ?thesis by blast
  next
    case False
    then obtain z' where z'∈Q (z', z) ∈ R by blast

    have ∀y. (y, z') ∈ ?A ⟶ y ∉ Q
    proof (intro allI impI)
      fix y assume (y, z') ∈ ?A
      then show y ∉ Q
      proof
        assume (y, z') ∈ R
        then have (y, z) ∈ R O R using ‹(z', z) ∈ R› ..
        with A1 show y ∉ Q .
      next
        assume (y, z') ∈ S
        then have (y, z) ∈ R O S using ‹(z', z) ∈ R› ..
        with A2 show y ∉ Q .
      qed
    qed
    with ‹z' ∈ Q› show ?thesis ..
  qed
 qed
qed

lemma wf-comp-self: wf R = wf (R O R)  — special case
  by (rule wf-union-merge [where S = {}, simplified])
```

21.3.1 acyclic

```
lemma acyclicI: ALL x. (x, x) ~: r^+ ==> acyclic r
  by (simp add: acyclic-def)

lemma wf-acyclic: wf r ==> acyclic r
apply (simp add: acyclic-def)
apply (blast elim: wf-trancl [THEN wf-irrefl])
done

lemmas wfP-acyclicP = wf-acyclic [to-pred]

lemma acyclic-insert [iff]:
     acyclic(insert (y,x) r) = (acyclic r & (x,y) ~: r^*)
apply (simp add: acyclic-def trancl-insert)
apply (blast intro: rtrancl-trans)
done
```

lemma *acyclic-converse* [*iff*]: *acyclic*(*r*^−1) = *acyclic r*
by (*simp add*: *acyclic-def trancl-converse*)

lemmas *acyclicP-converse* [*iff*] = *acyclic-converse* [*to-pred*]

lemma *acyclic-impl-antisym-rtrancl*: *acyclic r* ==> *antisym*(*r*^*)
apply (*simp add*: *acyclic-def antisym-def*)
apply (*blast elim*: *rtranclE intro*: *rtrancl-into-trancl1 rtrancl-trancl-trancl*)
done

lemma *acyclic-subset*: [| *acyclic s*; *r* <= *s* |] ==> *acyclic r*
apply (*simp add*: *acyclic-def*)
apply (*blast intro*: *trancl-mono*)
done

Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf* [*rule-format*]: *finite r* ==> *acyclic r* −−> *wf r*
apply (*erule finite-induct*, *blast*)
apply (*simp* (*no-asm-simp*) *only*: *split-tupled-all*)
apply *simp*
done

lemma *finite-acyclic-wf-converse*: [|*finite r*; *acyclic r*|] ==> *wf* (*r*^−1)
apply (*erule finite-converse* [*THEN iffD2*, *THEN finite-acyclic-wf*])
apply (*erule acyclic-converse* [*THEN iffD2*])
done

lemma *wf-iff-acyclic-if-finite*: *finite r* ==> *wf r* = *acyclic r*
by (*blast intro*: *finite-acyclic-wf wf-acyclic*)

21.4 Well-Founded Recursion

cut

lemma *cuts-eq*: (*cut f r x* = *cut g r x*) = (*ALL y*. (*y*,*x*):*r* −−> *f*(*y*)=*g*(*y*))
by (*simp add*: *expand-fun-eq cut-def*)

lemma *cut-apply*: (*x*,*a*):*r* ==> (*cut f r a*)(*x*) = *f*(*x*)
by (*simp add*: *cut-def*)

Inductive characterization of wfrec combinator; for details see: John Harrison, "Inductive definitions: automation and application"

lemma *wfrec-unique*: [| *adm-wf R F*; *wf R* |] ==> *EX*! *y*. *wfrec-rel R F x y*
apply (*simp add*: *adm-wf-def*)
apply (*erule-tac a*=*x* **in** *wf-induct*)
apply (*rule ex1I*)

apply (*rule-tac g = %x. THE y. wfrec-rel R F x y* **in** *wfrec-rel.wfrecI*)
apply (*fast dest*!: *theI'*)
apply (*erule wfrec-rel.cases*, *simp*)
apply (*erule allE*, *erule allE*, *erule allE*, *erule mp*)
apply (*fast intro*: *the-equality* [*symmetric*])
done

lemma *adm-lemma*: *adm-wf R* (*%f x. F* (*cut f R x*) *x*)
apply (*simp add*: *adm-wf-def*)
apply (*intro strip*)
apply (*rule cuts-eq* [*THEN iffD2*, *THEN subst*], *assumption*)
apply (*rule refl*)
done

lemma *wfrec*: *wf*(*r*) ==> *wfrec r H a = H* (*cut* (*wfrec r H*) *r a*) *a*
apply (*simp add*: *wfrec-def*)
apply (*rule adm-lemma* [*THEN wfrec-unique*, *THEN the1-equality*], *assumption*)
apply (*rule wfrec-rel.wfrecI*)
apply (*intro strip*)
apply (*erule adm-lemma* [*THEN wfrec-unique*, *THEN theI'*])
done

21.5 Code generator setup

consts-code
wfrec (⟨**module**⟩*wfrec?*)
attach ⟪
fun wfrec f x = f (*wfrec f*) *x*;
⟫

21.6 *nat* is well-founded

lemma *less-nat-rel*: *op <* = (*λm n. n = Suc m*) ^++
proof (*rule ext*, *rule ext*, *rule iffI*)
fix *n m* :: *nat*
assume *m < n*
then show (*λm n. n = Suc m*) ^++ *m n*
proof (*induct n*)
case *0* **then show** *?case* **by** *auto*
next
case (*Suc n*) **then show** *?case*
by (*auto simp add*: *less-Suc-eq-le le-less intro*: *tranclp.trancl-into-trancl*)
qed
next
fix *n m* :: *nat*
assume (*λm n. n = Suc m*) ^++ *m n*
then show *m < n*
by (*induct n*)
(*simp-all add*: *less-Suc-eq-le reflexive le-less*)
qed

definition
pred-nat :: $(nat * nat)$ *set* **where**
pred-nat = $\{(m, n).\ n = Suc\ m\}$

definition
less-than :: $(nat * nat)$ *set* **where**
less-than = *pred-nat*^+

lemma *less-eq*: $(m, n) \in$ *pred-nat*^+ $\longleftrightarrow$ $m < n$
unfolding *less-nat-rel pred-nat-def trancl-def* **by** *simp*

lemma *pred-nat-trancl-eq-le*:
$(m, n) \in$ *pred-nat*^* $\longleftrightarrow$ $m \leq n$
unfolding *less-eq rtrancl-eq-or-trancl* **by** *auto*

lemma *wf-pred-nat*: *wf pred-nat*
apply (*unfold wf-def pred-nat-def*, *clarify*)
apply (*induct-tac x*, *blast*+)
done

lemma *wf-less-than* [*iff*]: *wf less-than*
by (*simp add*: *less-than-def wf-pred-nat* [*THEN wf-trancl*])

lemma *trans-less-than* [*iff*]: *trans less-than*
by (*simp add*: *less-than-def trans-trancl*)

lemma *less-than-iff* [*iff*]: ((*x*,*y*): *less-than*) = (*x*<*y*)
by (*simp add*: *less-than-def less-eq*)

lemma *wf-less*: *wf* $\{(x, y{::}nat).\ x < y\}$
using *wf-less-than* **by** (*simp add*: *less-than-def less-eq* [*symmetric*])

21.7 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [**?**].

inductive-set
acc :: $('a * 'a)$ *set* => $'a$ *set*
for *r* :: $('a * 'a)$ *set*
where
accI: $(!!y.\ (y, x) : r$ ==> $y : acc\ r)$ ==> $x : acc\ r$

abbreviation
termip :: $('a$ => $'a$ => *bool*) => $'a$ => *bool* **where**
termip r == *accp* (r^{-1-1})

abbreviation
termi :: $('a * 'a)$ *set* => $'a$ *set* **where**
termi r == *acc* (r^{-1})

lemmas *accpI* = *accp.accI*

Induction rules

theorem *accp-induct*:
assumes *major*: *accp r a*
assumes *hyp*: !!*x*. *accp r x* ==> ∀ *y*. *r y x* −−> *P y* ==> *P x*
shows *P a*
apply (*rule major* [*THEN accp.induct*])
apply (*rule hyp*)
apply (*rule accp.accI*)
apply *fast*
apply *fast*
done

theorems *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: *accp r b* ==> *r a b* ==> *accp r a*
apply (*erule accp.cases*)
apply *fast*
done

lemma *not-accp-down*:
assumes *na*: ¬ *accp R x*
obtains *z* **where** *R z x* **and** ¬ *accp R z*
proof −
assume *a*: ⋀*z*. ⟦*R z x*; ¬ *accp R z*⟧ ⟹ *thesis*

show *thesis*
proof (*cases* ∀ *z*. *R z x* ⟶ *accp R z*)
case *True*
hence ⋀*z*. *R z x* ⟹ *accp R z* **by** *auto*
hence *accp R x*
by (*rule accp.accI*)
with *na* **show** *thesis* **..**
next
case *False* **then obtain** *z* **where** *R z x* **and** ¬ *accp R z*
by *auto*
with *a* **show** *thesis* **.**
qed
qed

lemma *accp-downwards-aux*: *r*** *b a* ==> *accp r a* −−> *accp r b*
apply (*erule rtranclp-induct*)
apply *blast*
apply (*blast dest*: *accp-downward*)
done

theorem *accp-downwards*: *accp r a* ==> *r*** *b a* ==> *accp r b*

```
  apply (blast dest: accp-downwards-aux)
  done

theorem accp-wfPI: ∀ 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 ≤ accp R
    and dcl: ⋀x z. ⟦D x; R z x⟧ ⟹ D z
    and D x
    and istep: ⋀x. ⟦D x; (⋀z. R z x ⟹ P z)⟧ ⟹ P x
  shows P x
proof −
  from subset and ‹D x›
  have accp R x ..
  then show P x using ‹D x›
  proof (induct x)
    fix x
```

assume $D\ x$
and $\bigwedge y.\ R\ y\ x \Longrightarrow D\ y \Longrightarrow P\ y$
with *dcl* **and** *istep* **show** $P\ x$ **by** *blast*
qed
qed

Set versions of the above theorems

lemmas *acc-induct* = *accp-induct* [*to-set*]

lemmas *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

lemmas *acc-downward* = *accp-downward* [*to-set*]

lemmas *not-acc-down* = *not-accp-down* [*to-set*]

lemmas *acc-downwards-aux* = *accp-downwards-aux* [*to-set*]

lemmas *acc-downwards* = *accp-downwards* [*to-set*]

lemmas *acc-wfI* = *accp-wfPI* [*to-set*]

lemmas *acc-wfD* = *accp-wfPD* [*to-set*]

lemmas *wf-acc-iff* = *wfP-accp-iff* [*to-set*]

lemmas *acc-subset* = *accp-subset* [*to-set pred-subset-eq*]

lemmas *acc-subset-induct* = *accp-subset-induct* [*to-set pred-subset-eq*]

21.8 Tools for building wellfounded relations

Inverse Image

lemma *wf-inv-image* [*simp*,*intro*!]: *wf*(*r*) ==> *wf*(*inv-image r* (*f*::'*a*=>'*b*))
apply (*simp* (*no-asm-use*) *add*: *inv-image-def wf-eq-minimal*)
apply *clarify*
apply (*subgoal-tac EX* (*w*::'*b*) . *w* : {*w*. *EX* (*x*::'*a*) . *x*: *Q* & (*f x* = *w*) })
prefer *2* **apply** (*blast del*: *allE*)
apply (*erule allE*)
apply (*erule* (*1*) *notE impE*)
apply *blast*
done

lemma *in-inv-image*[*simp*]: ((*x*,*y*) : *inv-image r f*) = ((*f x*, *f y*) : *r*)
by (*auto simp*:*inv-image-def*)

Measure functions into *nat*

definition *measure* :: ('*a* => *nat*) => ('*a* * '*a*)*set*
where *measure* == *inv-image less-than*

lemma *in-measure*[*simp*]: $((x,y) : measure\ f) = (f\ x < f\ y)$
by (*simp add*:*measure-def*)

lemma *wf-measure* [*iff*]: *wf* (*measure f*)
apply (*unfold measure-def*)
apply (*rule wf-less-than* [*THEN wf-inv-image*])
done

Lexicographic combinations

definition
lex-prod :: [('*a*×'*a*)*set*, ('*b*×'*b*)*set*] => (('*a*×'*b*)×('*a*×'*b*))*set*
(**infixr** <*lex*> 80)
where
ra <*lex*> *rb* == {((*a*,*b*),(*a'*,*b'*)). (*a*,*a'*) : *ra* | *a*=*a'* & (*b*,*b'*) : *rb*}

lemma *wf-lex-prod* [*intro*!]: [| *wf*(*ra*); *wf*(*rb*) |] ==> *wf*(*ra* <*lex*> *rb*)
apply (*unfold wf-def lex-prod-def*)
apply (*rule allI*, *rule impI*)
apply (*simp* (*no-asm-use*) *only*: *split-paired-All*)
apply (*drule spec*, *erule mp*)
apply (*rule allI*, *rule impI*)
apply (*drule spec*, *erule mp*, *blast*)
done

lemma *in-lex-prod*[*simp*]:
(((*a*,*b*),(*a'*,*b'*)): *r* <*lex*> *s*) = ((*a*,*a'*): *r* ∨ (*a* = *a'* ∧ (*b*, *b'*) : *s*))
by (*auto simp*:*lex-prod-def*)

op <*lex*> preserves transitivity

lemma *trans-lex-prod* [*intro*!]:
[| *trans R1*; *trans R2* |] ==> *trans* (*R1* <*lex*> *R2*)
by (*unfold trans-def lex-prod-def*, *blast*)

lexicographic combinations with measure functions

definition
mlex-prod :: ('*a* ⇒ *nat*) ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* (**infixr** <*mlex*> 80)
where
f <*mlex*> *R* = *inv-image* (*less-than* <*lex*> *R*) (%*x*. (*f x*, *x*))

lemma *wf-mlex*: *wf R* ⟹ *wf* (*f* <*mlex*> *R*)
unfolding *mlex-prod-def*
by *auto*

lemma *mlex-less*: *f x* < *f y* ⟹ (*x*, *y*) ∈ *f* <*mlex*> *R*
unfolding *mlex-prod-def* **by** *simp*

lemma *mlex-leq*: *f x* ≤ *f y* ⟹ (*x*, *y*) ∈ *R* ⟹ (*x*, *y*) ∈ *f* <*mlex*> *R*
unfolding *mlex-prod-def* **by** *auto*

proper subset relation on finite sets

definition *finite-psubset* :: (*'a set* ∗ *'a set*) *set*
where *finite-psubset* == {(*A*,*B*). *A* < *B* & *finite B*}

lemma *wf-finite-psubset*[*simp*]: *wf*(*finite-psubset*)
apply (*unfold finite-psubset-def*)
apply (*rule wf-measure* [*THEN wf-subset*])
apply (*simp add*: *measure-def inv-image-def less-than-def less-eq*)
apply (*fast elim*!: *psubset-card-mono*)
done

lemma *trans-finite-psubset*: *trans finite-psubset*
by (*simp add*: *finite-psubset-def less-le trans-def*, *blast*)

lemma *in-finite-psubset*[*simp*]: (*A*, *B*) ∈ *finite-psubset* = (*A* < *B* & *finite B*)
unfolding *finite-psubset-def* **by** *auto*

max- and min-extension of order to finite sets

inductive-set *max-ext* :: (*'a* × *'a*) *set* ⇒ (*'a set* × *'a set*) *set*
for *R* :: (*'a* × *'a*) *set*
where
max-extI[*intro*]: *finite X* ⟹ *finite Y* ⟹ *Y* ≠ {} ⟹ (⋀*x*. *x* ∈ *X* ⟹ ∃ *y*∈*Y*. (*x*, *y*) ∈ *R*) ⟹ (*X*, *Y*) ∈ *max-ext R*

lemma *max-ext-wf*:
assumes *wf*: *wf r*
shows *wf* (*max-ext r*)
proof (*rule acc-wfI*, *intro allI*)
fix *M* **show** *M* ∈ *acc* (*max-ext r*) (**is** - ∈ *?W*)
proof *cases*
assume *finite M*
thus *?thesis*
proof (*induct M*)
show {} ∈ *?W*
by (*rule accI*) (*auto elim*: *max-ext.cases*)
next
fix *M a* **assume** *M* ∈ *?W finite M*
with *wf* **show** *insert a M* ∈ *?W*
proof (*induct arbitrary*: *M*)
fix *M a*
assume *M* ∈ *?W* **and** [*intro*]: *finite M*
assume *hyp*: ⋀*b M*. (*b*, *a*) ∈ *r* ⟹ *M* ∈ *?W* ⟹ *finite M* ⟹ *insert b M* ∈ *?W*
{
fix *N M* :: *'a set*
assume *finite N finite M*
then
have ⟦*M* ∈ *?W* ; (⋀*y*. *y* ∈ *N* ⟹ (*y*, *a*) ∈ *r*)⟧ ⟹ *N* ∪ *M* ∈ *?W*
by (*induct N arbitrary*: *M*) (*auto simp*: *hyp*)

```
      }
      note add-less = this

      show insert a M ∈ ?W
      proof (rule accI)
        fix N assume Nless: (N, insert a M) ∈ max-ext r
        hence asm1: ⋀x. x ∈ N ⟹ (x, a) ∈ r ∨ (∃ y ∈ M. (x, y) ∈ r)
          by (auto elim!: max-ext.cases)

        let ?N1 = { n ∈ N. (n, a) ∈ r }
        let ?N2 = { n ∈ N. (n, a) ∉ r }
        have N: ?N1 ∪ ?N2 = N by (rule set-ext) auto
        from Nless have finite N by (auto elim: max-ext.cases)
        then have finites: finite ?N1 finite ?N2 by auto

        have ?N2 ∈ ?W
        proof cases
          assume [simp]: M = {}
          have Mw: {} ∈ ?W by (rule accI) (auto elim: max-ext.cases)

          from asm1 have ?N2 = {} by auto
          with Mw show ?N2 ∈ ?W by (simp only:)
        next
          assume M ≠ {}
          have N2: (?N2, M) ∈ max-ext r
            by (rule max-extI[OF - - ‹M ≠ {}›]) (insert asm1, auto intro: finites)

          with ‹M ∈ ?W› show ?N2 ∈ ?W by (rule acc-downward)
        qed
        with finites have ?N1 ∪ ?N2 ∈ ?W
          by (rule add-less) simp
        then show N ∈ ?W by (simp only: N)
      qed
    qed
  qed
 next
  assume [simp]: ¬ finite M
  show ?thesis
    by (rule accI) (auto elim: max-ext.cases)
 qed
qed

lemma max-ext-additive:
 (A, B) ∈ max-ext R ⟹ (C, D) ∈ max-ext R ⟹
  (A ∪ C, B ∪ D) ∈ max-ext R
by (force elim!: max-ext.cases)


definition
```

min-ext :: ('*a* × '*a*) *set* ⇒ ('*a set* × '*a set*) *set*
where
[*code del*]: *min-ext r* = {(*X*, *Y*) | *X Y*. *X* ≠ {} ∧ (∀ *y* ∈ *Y*. (∃ *x* ∈ *X*. (*x*, *y*) ∈ *r*))}

lemma *min-ext-wf*:
assumes *wf r*
shows *wf* (*min-ext r*)
proof (*rule wfI-min*)
fix *Q* :: '*a set set*
fix *x*
assume *nonempty*: *x* ∈ *Q*
show ∃ *m* ∈ *Q*. (∀ *n*. (*n*, *m*) ∈ *min-ext r* ⟶ *n* ∉ *Q*)
proof *cases*
assume *Q* = {{}} **thus** *?thesis* **by** (*simp add*: *min-ext-def*)
next
assume *Q* ≠ {{}}
with *nonempty*
obtain *e x* **where** *x* ∈ *Q e* ∈ *x* **by** *force*
then have *eU*: *e* ∈ ⋃ *Q* **by** *auto*
with ‹*wf r*›
obtain *z* **where** *z*: *z* ∈ ⋃ *Q* ⋀*y*. (*y*, *z*) ∈ *r* ⟹ *y* ∉ ⋃ *Q*
by (*erule wfE-min*)
from *z* **obtain** *m* **where** *m* ∈ *Q z* ∈ *m* **by** *auto*
from ‹*m* ∈ *Q*›
show *?thesis*
proof (*rule*, *intro bexI allI impI*)
fix *n*
assume *smaller*: (*n*, *m*) ∈ *min-ext r*
with ‹*z* ∈ *m*› **obtain** *y* **where** *y*: *y* ∈ *n* (*y*, *z*) ∈ *r* **by** (*auto simp*: *min-ext-def*)
then show *n* ∉ *Q* **using** *z*(*2*) **by** *auto*
qed
qed
qed

Wellfoundedness of *same-fst*

definition
same-fst :: ('*a* => *bool*) => ('*a* => ('*b* * '*b*)*set*) => (('*a*∗'*b*)∗('*a*∗'*b*))*set*
where
same-fst P R == {((*x*',*y*'),(*x*,*y*)) . *x*'=*x* & *P x* & (*y*',*y*) : *R x*}
— For *rec-def* declarations where the first n parameters stay unchanged in the recursive call.

lemma *same-fstI* [*intro*!]:
[| *P x*; (*y*',*y*) : *R x* |] ==> ((*x*,*y*'),(*x*,*y*)) : *same-fst P R*
by (*simp add*: *same-fst-def*)

lemma *wf-same-fst*:
assumes *prem*: (!!*x*. *P x* ==> *wf*(*R x*))

shows *wf*(*same-fst P R*)
apply (*simp cong del*: *imp-cong add*: *wf-def same-fst-def*)
apply (*intro strip*)
apply (*rename-tac a b*)
apply (*case-tac wf* (*R a*))
apply (*erule-tac a* = *b* **in** *wf-induct*, *blast*)
apply (*blast intro*: *prem*)
done

21.9 Weakly decreasing sequences (w.r.t. some well-founded order) stabilize.

This material does not appear to be used any longer.

lemma *sequence-trans*: [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^* |] ==> (*f* (*i*+*k*), *f i*) : *r*^*
by (*induct k*) (*auto intro*: *rtrancl-trans*)

lemma *wf-weak-decr-stable*:
assumes *as*: *ALL i*. (*f* (*Suc i*), *f i*) : *r*^* *wf* (*r*^+)
shows *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
proof −
have *lem*: !!*x*. [| *ALL i*. (*f* (*Suc i*), *f i*) : *r*^*; *wf* (*r*^+) |]
==> *ALL m*. *f m* = *x* −−> (*EX i*. *ALL k*. *f* (*m*+*i*+*k*) = *f* (*m*+*i*))
apply (*erule wf-induct*, *clarify*)
apply (*case-tac EX j*. (*f* (*m*+*j*), *f m*) : *r*^+)
apply *clarify*
apply (*subgoal-tac EX i*. *ALL k*. *f* ((*m*+*j*) +*i*+*k*) = *f* ((*m*+*j*) +*i*))
apply *clarify*
apply (*rule-tac x* = *j*+*i* **in** *exI*)
apply (*simp add*: *add-ac*, *blast*)
apply (*rule-tac x* = *0* **in** *exI*, *clarsimp*)
apply (*drule-tac i* = *m* **and** *k* = *k* **in** *sequence-trans*)
apply (*blast elim*: *rtranclE dest*: *rtrancl-into-trancl1*)
done

from *lem*[*OF as*, *THEN spec*, *of 0*, *simplified*]
show *?thesis* **by** *auto*
qed

lemma *weak-decr-stable*:
ALL i. *f* (*Suc i*) <= ((*f i*)::*nat*) ==> *EX i*. *ALL k*. *f* (*i*+*k*) = *f i*
apply (*rule-tac r* = *pred-nat* **in** *wf-weak-decr-stable*)
apply (*simp add*: *pred-nat-trancl-eq-le*)
apply (*intro wf-trancl wf-pred-nat*)
done

21.10 size of a datatype value

use *Tools/function-package/size.ML*

setup *Size.setup*

lemma *size-bool* [*code*]:
size (*b*::*bool*) = *0* **by** (*cases b*) *auto*

lemma *nat-size* [*simp*, *code*]: *size* (*n*::*nat*) = *n*
by (*induct n*) *simp-all*

declare *prod.size* [*noatp*]

lemma [*code*]:
size (*P* :: *'a Predicate.pred*) = *0* **by** (*cases P*) *simp*

lemma [*code*]:
pred-size f P = *0* **by** (*cases P*) *simp*

end

22 FunDef: Function Definitions and Termination Proofs

theory *FunDef*
imports *Wellfounded*
uses
Tools/prop-logic.ML
Tools/sat-solver.ML
(*Tools/function-package/fundef-lib.ML*)
(*Tools/function-package/fundef-common.ML*)
(*Tools/function-package/inductive-wrap.ML*)
(*Tools/function-package/context-tree.ML*)
(*Tools/function-package/fundef-core.ML*)
(*Tools/function-package/sum-tree.ML*)
(*Tools/function-package/mutual.ML*)
(*Tools/function-package/pattern-split.ML*)
(*Tools/function-package/fundef-package.ML*)
(*Tools/function-package/auto-term.ML*)
(*Tools/function-package/measure-functions.ML*)
(*Tools/function-package/lexicographic-order.ML*)
(*Tools/function-package/fundef-datatype.ML*)
(*Tools/function-package/induction-scheme.ML*)
(*Tools/function-package/termination.ML*)
(*Tools/function-package/decompose.ML*)
(*Tools/function-package/descent.ML*)
(*Tools/function-package/scnp-solve.ML*)
(*Tools/function-package/scnp-reconstruct.ML*)
begin

22.1 Definitions with default value.

definition
THE-default :: $'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$ **where**
THE-default d P = (*if* $(\exists!x.\ P\ x)$ *then* $(THE\ x.\ P\ x)$ *else d*)

lemma *THE-defaultI'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
by (*simp add*: *theI' THE-default-def*)

lemma *THE-default1-equality*:
$[\![\exists!x.\ P\ x;\ P\ a]\!] \Longrightarrow THE\text{-}default\ d\ P = a$
by (*simp add*: *the1-equality THE-default-def*)

lemma *THE-default-none*:
$\neg(\exists!x.\ P\ x) \Longrightarrow THE\text{-}default\ d\ P = d$
by (*simp add*:*THE-default-def*)

lemma *fundef-ex1-existence*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $G\ x\ (f\ x)$
apply (*simp only*: *f-def*)
apply (*rule THE-defaultI'*)
apply (*rule ex1*)
done

lemma *fundef-ex1-uniqueness*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
assumes *elm*: $G\ x\ (h\ x)$
shows $h\ x = f\ x$
apply (*simp only*: *f-def*)
apply (*rule THE-default1-equality* [*symmetric*])
apply (*rule ex1*)
apply (*rule elm*)
done

lemma *fundef-ex1-iff*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $(G\ x\ y) = (f\ x = y)$
apply (*auto simp*:*ex1 f-def THE-default1-equality*)
apply (*rule THE-defaultI'*)
apply (*rule ex1*)
done

lemma *fundef-default-value*:
assumes *f-def*: $f == (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *graph*: $\bigwedge x\ y.\ G\ x\ y \Longrightarrow D\ x$

```
  assumes ¬ D x
  shows f x = d x
proof −
  have ¬(∃ y. G x y)
  proof
    assume ∃ y. G x y
    hence D x using graph ..
    with ‹¬ D x› show False ..
  qed
  hence ¬(∃!y. G x y) by blast

  thus ?thesis
    unfolding f-def
    by (rule THE-default-none)
qed

definition in-rel-def[simp]:
  in-rel R x y == (x, y) ∈ R

lemma wf-in-rel:
  wf R ⟹ wfP (in-rel R)
  by (simp add: wfP-def)

use Tools/function-package/fundef-lib.ML
use Tools/function-package/fundef-common.ML
use Tools/function-package/inductive-wrap.ML
use Tools/function-package/context-tree.ML
use Tools/function-package/fundef-core.ML
use Tools/function-package/sum-tree.ML
use Tools/function-package/mutual.ML
use Tools/function-package/pattern-split.ML
use Tools/function-package/auto-term.ML
use Tools/function-package/fundef-package.ML
use Tools/function-package/fundef-datatype.ML
use Tools/function-package/induction-scheme.ML

setup ⟪
  FundefPackage.setup
  #> FundefDatatype.setup
  #> InductionScheme.setup
⟫
```

22.2 Measure Functions

```
inductive is-measure :: ('a ⇒ nat) ⇒ bool
where is-measure-trivial: is-measure f

use Tools/function-package/measure-functions.ML
setup MeasureFunctions.setup
```

lemma *measure-size*[*measure-function*]: *is-measure size*
by (*rule is-measure-trivial*)

lemma *measure-fst*[*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* ($\lambda p.$ *f* (*fst p*))
by (*rule is-measure-trivial*)
lemma *measure-snd*[*measure-function*]: *is-measure f* $\Longrightarrow$ *is-measure* ($\lambda p.$ *f* (*snd p*))
by (*rule is-measure-trivial*)

use *Tools/function-package/lexicographic-order.ML*
setup *LexicographicOrder.setup*

22.3 Congruence Rules

lemma *let-cong* [*fundef-cong*]:
M = *N* $\Longrightarrow$ ($\bigwedge x.$ *x* = *N* $\Longrightarrow$ *f x* = *g x*) $\Longrightarrow$ *Let M f* = *Let N g*
unfolding *Let-def* **by** *blast*

lemmas [*fundef-cong*] =
if-cong image-cong INT-cong UN-cong
bex-cong ball-cong imp-cong

lemma *split-cong* [*fundef-cong*]:
($\bigwedge x\ y.$ (*x*, *y*) = *q* $\Longrightarrow$ *f x y* = *g x y*) $\Longrightarrow$ *p* = *q*
$\Longrightarrow$ *split f p* = *split g q*
by (*auto simp*: *split-def*)

lemma *comp-cong* [*fundef-cong*]:
f (*g x*) = *f'* (*g' x'*) $\Longrightarrow$ (*f o g*) *x* = (*f' o g'*) *x'*
unfolding *o-apply* **.**

22.4 Simp rules for termination proofs

lemma *termination-basic-simps*[*termination-simp*]:
x < (*y*::*nat*) $\Longrightarrow$ *x* < *y* + *z*
x < *z* $\Longrightarrow$ *x* < *y* + *z*
x ≤ *y* $\Longrightarrow$ *x* ≤ *y* + (*z*::*nat*)
x ≤ *z* $\Longrightarrow$ *x* ≤ *y* + (*z*::*nat*)
x < *y* $\Longrightarrow$ *x* ≤ (*y*::*nat*)
by *arith*+

declare *le-imp-less-Suc*[*termination-simp*]

lemma *prod-size-simp*[*termination-simp*]:
prod-size f g p = *f* (*fst p*) + *g* (*snd p*) + *Suc 0*
by (*induct p*) *auto*

22.5 Decomposition

lemma *less-by-empty*:
  $A = \{\} \Longrightarrow A \subseteq B$
and *union-comp-emptyL*:
  $[\![A\ O\ C = \{\};\ B\ O\ C = \{\}]\!] \Longrightarrow (A \cup B)\ O\ C = \{\}$
and *union-comp-emptyR*:
  $[\![A\ O\ B = \{\};\ A\ O\ C = \{\}]\!] \Longrightarrow A\ O\ (B \cup C) = \{\}$
and *wf-no-loop*:
  $R\ O\ R = \{\} \Longrightarrow wf\ R$
by (*auto simp add*: *wf-comp-self*[*of R*])

22.6 Reduction Pairs

definition
  reduction-pair $P = (wf\ (fst\ P) \wedge snd\ P\ O\ fst\ P \subseteq fst\ P)$

lemma *reduction-pairI*[*intro*]: $wf\ R \Longrightarrow S\ O\ R \subseteq R \Longrightarrow$ *reduction-pair* (R, S)
unfolding *reduction-pair-def* **by** *auto*

lemma *reduction-pair-lemma*:
  assumes *rp*: *reduction-pair P*
  assumes $R \subseteq fst\ P$
  assumes $S \subseteq snd\ P$
  assumes *wf S*
  shows $wf\ (R \cup S)$
proof −
  from *rp* ‹$S \subseteq snd\ P$› **have** $wf\ (fst\ P)\ S\ O\ fst\ P \subseteq fst\ P$
    unfolding *reduction-pair-def* **by** *auto*
  with ‹*wf S*› **have** $wf\ (fst\ P \cup S)$
    by (*auto intro*: *wf-union-compatible*)
  moreover from ‹$R \subseteq fst\ P$› **have** $R \cup S \subseteq fst\ P \cup S$ **by** *auto*
  ultimately show *?thesis* **by** (*rule wf-subset*)
qed

definition
  rp-inv-image $= (\lambda(R,S)\ f.\ (inv\text{-}image\ R\ f,\ inv\text{-}image\ S\ f))$

lemma *rp-inv-image-rp*:
  reduction-pair $P \Longrightarrow$ *reduction-pair* (*rp-inv-image P f*)
  unfolding *reduction-pair-def rp-inv-image-def split-def*
  by *force*

22.7 Concrete orders for SCNP termination proofs

definition *pair-less* $=$ *less-than* $<*lex*>$ *less-than*
definition [*code del*]: *pair-leq* $=$ *pair-less*$^=$
definition *max-strict* $=$ *max-ext pair-less*
definition [*code del*]: *max-weak* $=$ *max-ext pair-leq* $\cup$ $\{(\{\}, \{\})\}$
definition [*code del*]: *min-strict* $=$ *min-ext pair-less*

definition [*code del*]: *min-weak* = *min-ext pair-leq* ∪ {({}, {})}

lemma *wf-pair-less*[*simp*]: *wf pair-less*
by (*auto simp*: *pair-less-def*)

Introduction rules for *pair-less*/*pair-leq*

lemma *pair-leqI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in$ *pair-leq*
and *pair-leqI2*: $a \leq b \Longrightarrow s \leq t \Longrightarrow ((a, s), (b, t)) \in$ *pair-leq*
and *pair-lessI1*: $a < b \Longrightarrow ((a, s), (b, t)) \in$ *pair-less*
and *pair-lessI2*: $a \leq b \Longrightarrow s < t \Longrightarrow ((a, s), (b, t)) \in$ *pair-less*
unfolding *pair-leq-def pair-less-def* **by** *auto*

Introduction rules for max

lemma *smax-emptyI*:
finite $Y \Longrightarrow Y \neq \{\} \Longrightarrow (\{\}, Y) \in$ *max-strict*
and *smax-insertI*:
$[\![y \in Y; (x, y) \in$ *pair-less*; $(X, Y) \in$ *max-strict*$]\!] \Longrightarrow$ (*insert x X*, Y) ∈ *max-strict*
and *wmax-emptyI*:
finite $X \Longrightarrow (\{\}, X) \in$ *max-weak*
and *wmax-insertI*:
$[\![y \in YS; (x, y) \in$ *pair-leq*; $(XS, YS) \in$ *max-weak*$]\!] \Longrightarrow$ (*insert x XS*, YS) ∈ *max-weak*
unfolding *max-strict-def max-weak-def* **by** (*auto elim*!: *max-ext.cases*)

Introduction rules for min

lemma *smin-emptyI*:
$X \neq \{\} \Longrightarrow (X, \{\}) \in$ *min-strict*
and *smin-insertI*:
$[\![x \in XS; (x, y) \in$ *pair-less*; $(XS, YS) \in$ *min-strict*$]\!] \Longrightarrow (XS,$ *insert y* $YS) \in$ *min-strict*
and *wmin-emptyI*:
$(X, \{\}) \in$ *min-weak*
and *wmin-insertI*:
$[\![x \in XS; (x, y) \in$ *pair-leq*; $(XS, YS) \in$ *min-weak*$]\!] \Longrightarrow (XS,$ *insert y* $YS) \in$ *min-weak*
by (*auto simp*: *min-strict-def min-weak-def min-ext-def*)

Reduction Pairs

lemma *max-ext-compat*:
assumes *S O R* ⊆ *R*
shows (*max-ext S* ∪ {({},{})}) *O max-ext R* ⊆ *max-ext R*
using *assms*
apply *auto*
apply (*elim max-ext.cases*)
apply *rule*
apply *auto*[*3*]
apply (*drule-tac x*=*xa* **in** *meta-spec*)
apply *simp*

apply (*erule bexE*)
apply (*drule-tac x=xb* **in** *meta-spec*)
by *auto*

lemma *max-rpair-set*: *reduction-pair* (*max-strict*, *max-weak*)
  unfolding *max-strict-def max-weak-def*
apply (*intro reduction-pairI max-ext-wf*)
apply *simp*
apply (*rule max-ext-compat*)
by (*auto simp*: *pair-less-def pair-leq-def*)

lemma *min-ext-compat*:
  assumes *S O R* ⊆ *R*
  shows (*min-ext S* ∪ {({},{})}) *O min-ext R* ⊆ *min-ext R*
using *assms*
apply (*auto simp*: *min-ext-def*)
apply (*drule-tac x=ya* **in** *bspec*, *assumption*)
apply (*erule bexE*)
apply (*drule-tac x=xc* **in** *bspec*)
apply *assumption*
by *auto*

lemma *min-rpair-set*: *reduction-pair* (*min-strict*, *min-weak*)
  unfolding *min-strict-def min-weak-def*
apply (*intro reduction-pairI min-ext-wf*)
apply *simp*
apply (*rule min-ext-compat*)
by (*auto simp*: *pair-less-def pair-leq-def*)

22.8 Tool setup

use *Tools/function-package/termination.ML*
use *Tools/function-package/decompose.ML*
use *Tools/function-package/descent.ML*
use *Tools/function-package/scnp-solve.ML*
use *Tools/function-package/scnp-reconstruct.ML*

setup ⟪ *ScnpReconstruct.setup* ⟫

ML-val — setup inactive
⟪
  Context.theory-map (*FundefCommon.set-termination-prover* (*ScnpReconstruct.decomp-scnp*

  [*ScnpSolve.MAX*, *ScnpSolve.MIN*, *ScnpSolve.MS*]))
⟫

end

23 Record: Extensible records with structural subtyping

theory *Record*
imports *Product-Type*
uses (*Tools/record-package.ML*)
begin

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
by *simp*

lemma *rec-UNIV-I*: $\bigwedge x.\ x{\in}UNIV \equiv True$
by *simp*

lemma *rec-True-simp*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
by *simp*

lemma *K-record-comp*: $(\lambda x.\ c) \circ f = (\lambda x.\ c)$
by (*simp add*: *comp-def*)

23.1 Concrete record syntax

nonterminals
ident field-type field-types field fields update updates
syntax
-constify :: *id* => *ident* (-)
-constify :: *longid* => *ident* (-)

-field-type :: [*ident*, *type*] => *field-type* ((*2*- ::/ -))
:: *field-type* => *field-types* (-)
-field-types :: [*field-type*, *field-types*] => *field-types* (-,/ -)
-record-type :: *field-types* => *type* ((*3'*(| - |')))
-record-type-scheme :: [*field-types*, *type*] => *type* ((*3'*(| -,/ (*2*... ::/ -) |')))

-field :: [*ident*, *'a*] => *field* ((*2*- =/ -))
:: *field* => *fields* (-)
-fields :: [*field*, *fields*] => *fields* (-,/ -)
-record :: *fields* => *'a* ((*3'*(| - |')))
-record-scheme :: [*fields*, *'a*] => *'a* ((*3'*(| -,/ (*2*... =/ -) |')))

-update-name :: *idt*
-update :: [*ident*, *'a*] => *update* ((*2*- :=/ -))
:: *update* => *updates* (-)
-updates :: [*update*, *updates*] => *updates* (-,/ -)
-record-update :: [*'a*, *updates*] => *'b* (-/(*3'*(| - |')) [*900*,*0*] *900*)

syntax (*xsymbols*)
-record-type :: *field-types* => *type* ((*3*(|-|)))
-record-type-scheme :: [*field-types*, *type*] => *type* ((*3*(|-,/ (*2*... ::/ -)|)))

```
  -record              :: fields => 'a                          ((3(|-|)))
  -record-scheme       :: [fields, 'a] => 'a                    ((3(|-,/ (2... =/ -)|)))
  -record-update       :: ['a, updates] => 'b                   (-/(3(|-|)) [900,0] 900)
```

use *Tools/record-package.ML*
setup *RecordPackage.setup*

end

24 Option: Datatype option

theory *Option*
imports *Datatype Finite-Set*
begin

datatype *'a option = None | Some 'a*

lemma *not-None-eq* [*iff*]: *(x ~= None) = (EX y. x = Some y)*
by *(induct x) auto*

lemma *not-Some-eq* [*iff*]: *(ALL y. x ~= Some y) = (x = None)*
by *(induct x) auto*

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform iff attribute.

lemma *option-caseE*:
assumes *c*: *(case x of None => P | Some y => Q y)*
obtains
(None) x = None **and** *P*
| *(Some) y* **where** *x = Some y* **and** *Q y*
using *c* **by** *(cases x) simp-all*

lemma *insert-None-conv-UNIV*: *insert None (range Some) = UNIV*
by *(rule set-ext, case-tac x) auto*

instance *option* :: *(finite) finite* **proof**
qed *(simp add*: *insert-None-conv-UNIV* [*symmetric*])

lemma *inj-Some* [*simp*]: *inj-on Some A*
by *(rule inj-onI) simp*

24.0.1 Operations

primrec *the* :: *'a option => 'a* **where**
the (Some x) = x

primrec *set* :: *'a option => 'a set* **where**

set None = {} |
set (*Some x*) = {*x*}

lemma *ospec* [*dest*]: (*ALL x*:*set A*. *P x*) ==> *A* = *Some x* ==> *P x*
by *simp*

declaration ⟪ *fn* - =>
Classical.map-cs (*fn cs* => *cs addSD2* (*ospec*, *thm ospec*))
⟫

lemma *elem-set* [*iff*]: (*x* : *set xo*) = (*xo* = *Some x*)
by (*cases xo*) *auto*

lemma *set-empty-eq* [*simp*]: (*set xo* = {}) = (*xo* = *None*)
by (*cases xo*) *auto*

definition
map :: ('*a* ⇒ '*b*) ⇒ '*a option* ⇒ '*b option*
where
[*code del*]: *map* = (%*f y*. *case y of None* => *None* | *Some x* => *Some* (*f x*))

lemma *option-map-None* [*simp*, *code*]: *map f None* = *None*
by (*simp add*: *map-def*)

lemma *option-map-Some* [*simp*, *code*]: *map f* (*Some x*) = *Some* (*f x*)
by (*simp add*: *map-def*)

lemma *option-map-is-None* [*iff*]:
(*map f opt* = *None*) = (*opt* = *None*)
by (*simp add*: *map-def split add*: *option.split*)

lemma *option-map-eq-Some* [*iff*]:
(*map f xo* = *Some y*) = (*EX z*. *xo* = *Some z* & *f z* = *y*)
by (*simp add*: *map-def split add*: *option.split*)

lemma *option-map-comp*:
map f (*map g opt*) = *map* (*f o g*) *opt*
by (*simp add*: *map-def split add*: *option.split*)

lemma *option-map-o-sum-case* [*simp*]:
map f o sum-case g h = *sum-case* (*map f o g*) (*map f o h*)
by (*rule ext*) (*simp split*: *sum.split*)

hide (**open**) *const set map*

24.0.2 Code generator setup

definition

is-none :: *'a option* ⇒ *bool* **where**
is-none-none [*code post*, *symmetric*, *code inline*]: *is-none x* ⟷ *x* = *None*

lemma *is-none-code* [*code*]:
shows *is-none None* ⟷ *True*
and *is-none* (*Some x*) ⟷ *False*
unfolding *is-none-none* [*symmetric*] **by** *simp-all*

hide (**open**) *const is-none*

code-type *option*
(*SML* - *option*)
(*OCaml* - *option*)
(*Haskell Maybe* -)

code-const *None* **and** *Some*
(*SML NONE* **and** *SOME*)
(*OCaml None* **and** *Some* -)
(*Haskell Nothing* **and** *Just*)

code-instance *option* :: *eq*
(*Haskell* −)

code-const *eq-class.eq* :: *'a*::*eq option* ⇒ *'a option* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-reserved *SML*
option NONE SOME

code-reserved *OCaml*
option None Some

end

25 Extraction: Program extraction for HOL

theory *Extraction*
imports *Option*
uses *Tools/rewrite-hol-proof.ML*
begin

25.1 Setup

setup ⟪
let
fun realizes-set-proc (*Const* (*realizes*, *Type* (*fun*, [*Type* (*Null*, []), -])) $ *r* $
(*Const* (*op* :, -) $ *x* $ *S*)) = (*case strip-comb S of*
(*Var* (*ixn*, *U*), *ts*) => *SOME* (*list-comb* (*Var* (*ixn*, *U*), *ts* @ [*x*]))

```
    | (Free (s, U), ts) => SOME (list-comb (Free (s, U), ts @ [x]))
    | - => NONE)
  | realizes-set-proc (Const (realizes, Type (fun, [T, -])) $ r $
    (Const (op :, -) $ x $ S)) = (case strip-comb S of
      (Var (ixn, U), ts) => SOME (list-comb (Var (ixn, T --> U), r :: ts @
[x]))
    | (Free (s, U), ts) => SOME (list-comb (Free (s, T --> U), r :: ts @ [x]))
    | - => NONE)
  | realizes-set-proc - = NONE;

in
  Extraction.add-types
    [(bool, ([], NONE))] #>
  Extraction.set-preprocessor (fn thy =>
    Proofterm.rewrite-proof-notypes
      ([], RewriteHOLProof.elim-cong :: ProofRewriteRules.rprocs true) o
    Proofterm.rewrite-proof thy
      (RewriteHOLProof.rews, ProofRewriteRules.rprocs true) o
    ProofRewriteRules.elim-vars (curry Const @{const-name default}))
end
⟩⟩

lemmas [extraction-expand] =
  meta-spec atomize-eq atomize-all atomize-imp atomize-conj
  allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
  notE' impE' impE iffE imp-cong simp-thms eq-True eq-False
  induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def
  induct-atomize induct-rulify induct-rulify-fallback
  True-implies-equals TrueE

datatype sumbool = Left | Right
```

25.2 Type of extracted program

extract-type

```
  typeof (Trueprop P) ≡ typeof P

  typeof P ≡ Type (TYPE(Null)) ⟹ typeof Q ≡ Type (TYPE('Q)) ⟹
    typeof (P ⟶ Q) ≡ Type (TYPE('Q))

  typeof Q ≡ Type (TYPE(Null)) ⟹ typeof (P ⟶ Q) ≡ Type (TYPE(Null))

  typeof P ≡ Type (TYPE('P)) ⟹ typeof Q ≡ Type (TYPE('Q)) ⟹
    typeof (P ⟶ Q) ≡ Type (TYPE('P ⇒ 'Q))

  (λx. typeof (P x)) ≡ (λx. Type (TYPE(Null))) ⟹
    typeof (∀ x. P x) ≡ Type (TYPE(Null))
```

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$\quad typeof\ (\forall\, x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$\quad typeof\ (\exists\, x{::}'a.\ P\ x) \equiv Type\ (TYPE('a))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$\quad typeof\ (\exists\, x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

25.3 Realizability

realizability
$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$\ (typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \longrightarrow Q)) \equiv (\forall\, x{::}'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall\, x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg P) \equiv (\forall x{::}'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P{::}bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

25.4 Computational content of basic inference rules

theorem *disjE-realizer*:
assumes *r*: $case\ x\ of\ Inl\ p \Rightarrow P\ p \mid Inr\ q \Rightarrow Q\ q$

and *r1*: $\bigwedge p.\ P\ p \Longrightarrow R\ (f\ p)$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows R (*case x of Inl p ⇒ f p | Inr q ⇒ g q*)
proof (*cases x*)
case *Inl*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Inr*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *disjE-realizer2*:
assumes *r*: *case x of None ⇒ P | Some q ⇒ Q q*
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows R (*case x of None ⇒ f | Some q ⇒ g q*)
proof (*cases x*)
case *None*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Some*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *disjE-realizer3*:
assumes *r*: *case x of Left ⇒ P | Right ⇒ Q*
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $Q \Longrightarrow R\ g$
shows R (*case x of Left ⇒ f | Right ⇒ g*)
proof (*cases x*)
case *Left*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Right*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *conjI-realizer*:
$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (fst\ (p,\ q)) \wedge Q\ (snd\ (p,\ q))$
by *simp*

theorem *exI-realizer*:
$P\ y\ x \Longrightarrow P\ (snd\ (x,\ y))\ (fst\ (x,\ y))$ **by** *simp*

theorem *exE-realizer*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q\ (f\ x\ y)) \Longrightarrow Q\ (let\ (x,\ y) = p\ in\ f\ x\ y)$
by (*cases p*) (*simp add*: *Let-def*)

theorem *exE-realizer′*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q$ **by** (*cases p*) *simp*

setup ⟪

Sign.add-const-constraint (@{*const-name default*}, *SOME* @{*typ 'a*::*type*})
⟫

realizers
impI (*P*, *Q*): λ*pq*. *pq*
Λ *P Q pq* (*h*: -). *allI* · - • (Λ *x*. *impI* · - - - • (*h* · *x*))

impI (*P*): *Null*
Λ *P Q* (*h*: -). *allI* · - • (Λ *x*. *impI* · - - - • (*h* · *x*))

impI (*Q*): λ*q*. *q* Λ *P Q q*. *impI* · - - -

impI: *Null impI*

mp (*P*, *Q*): λ*pq*. *pq*
Λ *P Q pq* (*h*: -) *p*. *mp* · - - - • (*spec* · - · *p* • *h*)

mp (*P*): *Null*
Λ *P Q* (*h*: -) *p*. *mp* · - - - • (*spec* · - · *p* • *h*)

mp (*Q*): λ*q*. *q* Λ *P Q q*. *mp* · - - -

mp: *Null mp*

allI (*P*): λ*p*. *p* Λ *P p*. *allI* · -

allI: *Null allI*

spec (*P*): λ*x p*. *p x* Λ *P x p*. *spec* · - · *x*

spec: *Null spec*

exI (*P*): λ*x p*. (*x*, *p*) Λ *P x p*. *exI-realizer* · *P* · *p* · *x*

exI: λ*x*. *x* Λ *P x* (*h*: -). *h*

exE (*P*, *Q*): λ*p pq*. *let* (*x*, *y*) = *p in pq x y*
Λ *P Q p* (*h*: -) *pq*. *exE-realizer* · *P* · *p* · *Q* · *pq* • *h*

exE (*P*): *Null*
Λ *P Q p*. *exE-realizer'* · - - - - -

exE (*Q*): λ*x pq*. *pq x*
Λ *P Q x* (*h1*: -) *pq* (*h2*: -). *h2* · *x* • *h1*

exE: *Null*
Λ *P Q x* (*h1*: -) (*h2*: -). *h2* · *x* • *h1*

conjI (*P*, *Q*): *Pair*

$\Lambda\ P\ Q\ p\ (h\colon \text{-})\ q.\ \mathit{conjI\text{-}realizer} \cdot P \cdot p \cdot Q \cdot q \bullet h$

conjI (P): $\lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjI} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjI (Q): $\lambda q.\ q$
$\Lambda\ P\ Q\ (h\colon \text{-})\ q.\ \mathit{conjI} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet h$

conjI: *Null conjI*

conjunct1 $(P,\ Q)$: *fst*
$\Lambda\ P\ Q\ pq.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct1 (P): $\lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct1 (Q): *Null*
$\Lambda\ P\ Q\ q.\ \mathit{conjunct1} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct1: *Null conjunct1*

conjunct2 $(P,\ Q)$: *snd*
$\Lambda\ P\ Q\ pq.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct2 (P): *Null*
$\Lambda\ P\ Q\ p.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct2 (Q): $\lambda p.\ p$
$\Lambda\ P\ Q\ p.\ \mathit{conjunct2} \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjunct2: *Null conjunct2*

disjI1 $(P,\ Q)$: *Inl*
$\Lambda\ P\ Q\ p.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{sum.cases\text{-}1} \cdot P \cdot \text{-} \cdot \text{-} \cdot p)$

disjI1 (P): *Some*
$\Lambda\ P\ Q\ p.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{option.cases\text{-}2} \cdot \text{-} \cdot \text{-} \cdot P \cdot p)$

disjI1 (Q): *None*
$\Lambda\ P\ Q.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{option.cases\text{-}1} \cdot \text{-} \cdot \text{-} \cdot \text{-})$

disjI1: *Left*
$\Lambda\ P\ Q.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{sumbool.cases\text{-}1} \cdot \text{-} \cdot \text{-} \cdot \text{-})$

disjI2 $(P,\ Q)$: *Inr*
$\Lambda\ Q\ P\ q.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{sum.cases\text{-}2} \cdot \text{-} \cdot \text{-} \cdot Q \cdot q)$

disjI2 (P): *None*
$\Lambda\ Q\ P.\ \mathit{iffD2} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (\mathit{option.cases\text{-}1} \cdot \text{-} \cdot \text{-} \cdot \text{-})$

disjI2 (*Q*): *Some*
  Λ *Q P q*. *iffD2* · - · - · (*option.cases-2* · - · *Q* · *q*)

disjI2: *Right*
  Λ *Q P*. *iffD2* · - · - · (*sumbool.cases-2* · - · -)

disjE (*P*, *Q*, *R*): λ*pq pr qr*.
  (*case pq of Inl p* ⇒ *pr p* | *Inr q* ⇒ *qr q*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*Q*, *R*): λ*pq pr qr*.
  (*case pq of None* ⇒ *pr* | *Some q* ⇒ *qr q*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer2 · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*P*, *R*): λ*pq pr qr*.
  (*case pq of None* ⇒ *qr* | *Some p* ⇒ *pr p*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr* (*h3*: -).
    disjE-realizer2 · - · - · *pq* · *R* · *qr* · *pr* · *h1* · *h3* · *h2*

disjE (*R*): λ*pq pr qr*.
  (*case pq of Left* ⇒ *pr* | *Right* ⇒ *qr*)
  Λ *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr*.
    disjE-realizer3 · - · - · *pq* · *R* · *pr* · *qr* · *h1* · *h2*

disjE (*P*, *Q*): *Null*
  Λ *P Q R pq*. *disjE-realizer* · - · - · *pq* · (λ*x*. *R*) · - · -

disjE (*Q*): *Null*
  Λ *P Q R pq*. *disjE-realizer2* · - · - · *pq* · (λ*x*. *R*) · - · -

disjE (*P*): *Null*
  Λ *P Q R pq* (*h1*: -) (*h2*: -) (*h3*: -).
    disjE-realizer2 · - · - · *pq* · (λ*x*. *R*) · - · - · *h1* · *h3* · *h2*

disjE: *Null*
  Λ *P Q R pq*. *disjE-realizer3* · - · - · *pq* · (λ*x*. *R*) · - · -

FalseE (*P*): *default*
  Λ *P*. *FalseE* · -

FalseE: *Null FalseE*

notI (*P*): *Null*
  Λ *P* (*h*: -). *allI* · - · (Λ *x*. *notI* · - · (*h* · *x*))

notI: *Null notI*

notE (*P*, *R*): $\lambda p.$ *default*
  Λ *P R* (*h*: -) *p*. *notE* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *p* $\cdot$ *h*)

notE (*P*): *Null*
  Λ *P R* (*h*: -) *p*. *notE* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *p* $\cdot$ *h*)

notE (*R*): *default*
  Λ *P R*. *notE* $\cdot$ - $\cdot$ -

notE: *Null notE*

subst (*P*): $\lambda s\ t\ ps.$ *ps*
  Λ *s t P* (*h*: -) *ps*. *subst* $\cdot$ *s* $\cdot$ *t* $\cdot$ *P ps* $\cdot$ *h*

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
  Λ *Q P pq* (*h*: -) *p*.
    mp $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *p* $\cdot$ (*conjunct1* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*))

iffD1 (*P*): $\lambda p.$ *p*
  Λ *Q P p* (*h*: -). *mp* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*conjunct1* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*)

iffD1 (*Q*): *Null*
  Λ *Q P q1* (*h*: -) *q2*.
    mp $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *q2* $\cdot$ (*conjunct1* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*))

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
  Λ *P Q pq* (*h*: -) *q*.
    mp $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *q* $\cdot$ (*conjunct2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*))

iffD2 (*P*): $\lambda p.$ *p*
  Λ *P Q p* (*h*: -). *mp* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*conjunct2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*)

iffD2 (*Q*): *Null*
  Λ *P Q q1* (*h*: -) *q2*.
    mp $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*spec* $\cdot$ - $\cdot$ *q2* $\cdot$ (*conjunct2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *h*))

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
  Λ *P Q pq* (*h1* : -) *qp* (*h2* : -). *conjI-realizer* $\cdot$
    ($\lambda pq.\ \forall x.\ P\ x \longrightarrow Q\ (pq\ x)$) $\cdot$ *pq* $\cdot$
    ($\lambda qp.\ \forall x.\ Q\ x \longrightarrow P\ (qp\ x)$) $\cdot$ *qp* $\cdot$
    (*allI* $\cdot$ - $\cdot$ (Λ *x*. *impI* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*h1* $\cdot$ *x*))) $\cdot$
    (*allI* $\cdot$ - $\cdot$ (Λ *x*. *impI* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*h2* $\cdot$ *x*)))

iffI (*P*): $\lambda p.\ p$
$\Lambda\ P\ Q\ (h1 : \text{-})\ p\ (h2 : \text{-}).\ conjI \cdot \text{-} \cdot \text{-} \cdot$
$(allI \cdot \text{-} \cdot (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot (h1 \cdot x))) \cdot$
$(impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot h2)$

iffI (*Q*): $\lambda q.\ q$
$\Lambda\ P\ Q\ q\ (h1 : \text{-})\ (h2 : \text{-}).\ conjI \cdot \text{-} \cdot \text{-} \cdot$
$(impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot h1) \cdot$
$(allI \cdot \text{-} \cdot (\Lambda\ x.\ impI \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot (h2 \cdot x)))$

iffI: *Null iffI*

setup ⟪
Sign.add-const-constraint (@{*const-name default*}, *SOME* @{*typ 'a*::*default*})
⟫

end

26 Divides: The division operators div and mod

theory *Divides*
imports *Nat Power Product-Type*
uses ~~/*src/Provers/Arith/cancel-div-mod.ML*
begin

26.1 Syntactic division operations

class *div* = *dvd* +
fixes *div* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *div* 70)
and *mod* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** *mod* 70)

26.2 Abstract division in commutative semirings.

class *semiring-div* = *comm-semiring-1-cancel* + *div* +
assumes *mod-div-equality*: $a\ div\ b * b + a\ mod\ b = a$
and *div-by-0* [*simp*]: $a\ div\ 0 = 0$
and *div-0* [*simp*]: $0\ div\ a = 0$
and *div-mult-self1* [*simp*]: $b \neq 0 \Longrightarrow (a + c * b)\ div\ b = c + a\ div\ b$
begin

op div and *op mod*

lemma *mod-div-equality2*: $b * (a\ div\ b) + a\ mod\ b = a$
unfolding *mult-commute* [*of b*]
by (*rule mod-div-equality*)

lemma *mod-div-equality'*: *a mod b + a div b * b = a*
using *mod-div-equality* [*of a b*]
by (*simp only*: *add-ac*)

lemma *div-mod-equality*: *((a div b) * b + a mod b) + c = a + c*
by (*simp add*: *mod-div-equality*)

lemma *div-mod-equality2*: *(b * (a div b) + a mod b) + c = a + c*
by (*simp add*: *mod-div-equality2*)

lemma *mod-by-0* [*simp*]: *a mod 0 = a*
using *mod-div-equality* [*of a zero*] **by** *simp*

lemma *mod-0* [*simp*]: *0 mod a = 0*
using *mod-div-equality* [*of zero a*] *div-0* **by** *simp*

lemma *div-mult-self2* [*simp*]:
assumes *b ≠ 0*
shows *(a + b * c) div b = c + a div b*
using *assms div-mult-self1* [*of b a c*] **by** (*simp add*: *mult-commute*)

lemma *mod-mult-self1* [*simp*]: *(a + c * b) mod b = a mod b*
proof (*cases b = 0*)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
have *a + c * b = (a + c * b) div b * b + (a + c * b) mod b*
by (*simp add*: *mod-div-equality*)
also from *False div-mult-self1* [*of b a c*] **have**
*... = (c + a div b) * b + (a + c * b) mod b*
by (*simp add*: *algebra-simps*)
finally have *a = a div b * b + (a + c * b) mod b*
by (*simp add*: *add-commute* [*of a*] *add-assoc left-distrib*)
then have *a div b * b + (a + c * b) mod b = a div b * b + a mod b*
by (*simp add*: *mod-div-equality*)
then show *?thesis* **by** *simp*
qed

lemma *mod-mult-self2* [*simp*]: *(a + b * c) mod b = a mod b*
by (*simp add*: *mult-commute* [*of b*])

lemma *div-mult-self1-is-id* [*simp*]: *b ≠ 0 ⟹ b * a div b = a*
using *div-mult-self2* [*of b 0 a*] **by** *simp*

lemma *div-mult-self2-is-id* [*simp*]: *b ≠ 0 ⟹ a * b div b = a*
using *div-mult-self1* [*of b 0 a*] **by** *simp*

lemma *mod-mult-self1-is-0* [*simp*]: *b * a mod b = 0*
using *mod-mult-self2* [*of 0 b a*] **by** *simp*

lemma *mod-mult-self2-is-0* [*simp*]: *a* ∗ *b mod b* = *0*
using *mod-mult-self1* [*of 0 a b*] **by** *simp*

lemma *div-by-1* [*simp*]: *a div 1* = *a*
using *div-mult-self2-is-id* [*of 1 a*] *zero-neq-one* **by** *simp*

lemma *mod-by-1* [*simp*]: *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-self* [*simp*]: *a mod a* = *0*
using *mod-mult-self2-is-0* [*of 1*] **by** *simp*

lemma *div-self* [*simp*]: *a* ≠ *0* ⟹ *a div a* = *1*
using *div-mult-self2-is-id* [*of - 1*] **by** *simp*

lemma *div-add-self1* [*simp*]:
assumes *b* ≠ *0*
shows (*b* + *a*) *div b* = *a div b* + *1*
using *assms div-mult-self1* [*of b a 1*] **by** (*simp add*: *add-commute*)

lemma *div-add-self2* [*simp*]:
assumes *b* ≠ *0*
shows (*a* + *b*) *div b* = *a div b* + *1*
using *assms div-add-self1* [*of b a*] **by** (*simp add*: *add-commute*)

lemma *mod-add-self1* [*simp*]:
(*b* + *a*) *mod b* = *a mod b*
using *mod-mult-self1* [*of a 1 b*] **by** (*simp add*: *add-commute*)

lemma *mod-add-self2* [*simp*]:
(*a* + *b*) *mod b* = *a mod b*
using *mod-mult-self1* [*of a 1 b*] **by** *simp*

lemma *mod-div-decomp*:
fixes *a b*
obtains *q r* **where** *q* = *a div b* **and** *r* = *a mod b*
and *a* = *q* ∗ *b* + *r*
proof −
from *mod-div-equality* **have** *a* = *a div b* ∗ *b* + *a mod b* **by** *simp*
moreover have *a div b* = *a div b* **..**
moreover have *a mod b* = *a mod b* **..**
note *that* **ultimately show** *thesis* **by** *blast*
qed

lemma *dvd-eq-mod-eq-0* [*code unfold*]: *a dvd b* ⟷ *b mod a = 0*
proof
  assume *b mod a = 0*
  with *mod-div-equality* [*of b a*] **have** *b div a * a = b* **by** *simp*
  then have *b = a * (b div a)* **unfolding** *mult-commute* **..**
  then have ∃ *c*. *b = a * c* **..**
  then show *a dvd b* **unfolding** *dvd-def* **.**
next
  assume *a dvd b*
  then have ∃ *c*. *b = a * c* **unfolding** *dvd-def* **.**
  then obtain *c* **where** *b = a * c* **..**
  then have *b mod a = a * c mod a* **by** *simp*
  then have *b mod a = c * a mod a* **by** (*simp add*: *mult-commute*)
  then show *b mod a = 0* **by** *simp*
qed

lemma *mod-div-trivial* [*simp*]: *a mod b div b = 0*
proof (*cases b = 0*)
  assume *b = 0*
  thus *?thesis* **by** *simp*
next
  assume *b ≠ 0*
  hence *a div b + a mod b div b = (a mod b + a div b * b) div b*
    by (*rule div-mult-self1* [*symmetric*])
  also have ... = *a div b*
    by (*simp only*: *mod-div-equality'*)
  also have ... = *a div b + 0*
    by *simp*
  finally show *?thesis*
    by (*rule add-left-imp-eq*)
qed

lemma *mod-mod-trivial* [*simp*]: *a mod b mod b = a mod b*
proof −
  have *a mod b mod b = (a mod b + a div b * b) mod b*
    by (*simp only*: *mod-mult-self1*)
  also have ... = *a mod b*
    by (*simp only*: *mod-div-equality'*)
  finally show *?thesis* **.**
qed

lemma *dvd-imp-mod-0*: *a dvd b* ⟹ *b mod a = 0*
by (*rule dvd-eq-mod-eq-0*[*THEN iffD1*])

lemma *dvd-div-mult-self*: *a dvd b* ⟹ *(b div a) * a = b*
by (*subst* (*2*) *mod-div-equality* [*of b a*, *symmetric*]) (*simp add*:*dvd-imp-mod-0*)

lemma *dvd-div-mult*: *a dvd b* ⟹ *(b div a) * c = b * c div a*
apply (*cases a = 0*)

apply *simp*
apply (*auto simp*: *dvd-def mult-assoc*)
done

lemma *div-dvd-div*[*simp*]:
  a dvd b $\Longrightarrow$ *a dvd c* $\Longrightarrow$ (*b div a dvd c div a*) = (*b dvd c*)
apply (*cases a* = *0*)
apply *simp*
apply (*unfold dvd-def*)
apply *auto*
apply(*blast intro*:*mult-assoc*[*symmetric*])
apply(*fastsimp simp add*: *mult-assoc*)
done

lemma *dvd-mod-imp-dvd*: [| *k dvd m mod n*; *k dvd n* |] ==> *k dvd m*
  apply (*subgoal-tac k dvd* (*m div n*) ∗*n* + *m mod n*)
  apply (*simp add*: *mod-div-equality*)
  apply (*simp only*: *dvd-add dvd-mult*)
  done

Addition respects modular equivalence.

lemma *mod-add-left-eq*: (*a* + *b*) *mod c* = (*a mod c* + *b*) *mod c*
proof −
  have (*a* + *b*) *mod c* = (*a div c* ∗ *c* + *a mod c* + *b*) *mod c*
    by (*simp only*: *mod-div-equality*)
  also have ... = (*a mod c* + *b* + *a div c* ∗ *c*) *mod c*
    by (*simp only*: *add-ac*)
  also have ... = (*a mod c* + *b*) *mod c*
    by (*rule mod-mult-self1*)
  finally show *?thesis* **.**
qed

lemma *mod-add-right-eq*: (*a* + *b*) *mod c* = (*a* + *b mod c*) *mod c*
proof −
  have (*a* + *b*) *mod c* = (*a* + (*b div c* ∗ *c* + *b mod c*)) *mod c*
    by (*simp only*: *mod-div-equality*)
  also have ... = (*a* + *b mod c* + *b div c* ∗ *c*) *mod c*
    by (*simp only*: *add-ac*)
  also have ... = (*a* + *b mod c*) *mod c*
    by (*rule mod-mult-self1*)
  finally show *?thesis* **.**
qed

lemma *mod-add-eq*: (*a* + *b*) *mod c* = (*a mod c* + *b mod c*) *mod c*
by (*rule trans* [*OF mod-add-left-eq mod-add-right-eq*])

lemma *mod-add-cong*:
  assumes *a mod c* = *a′ mod c*
  assumes *b mod c* = *b′ mod c*

shows $(a + b)$ *mod* $c = (a' + b')$ *mod* c
proof −
have $(a$ *mod* $c + b$ *mod* $c)$ *mod* $c = (a'$ *mod* $c + b'$ *mod* $c)$ *mod* c
unfolding *assms* **..**
thus *?thesis*
by (*simp only*: *mod-add-eq* [*symmetric*])
qed

lemma *div-add*[*simp*]: *z dvd x* ⟹ *z dvd y*
⟹ $(x + y)$ *div* $z = x$ *div* $z + y$ *div* z
by(*cases z=0*, *simp*, *unfold dvd-def*, *auto simp add*: *algebra-simps*)

Multiplication respects modular equivalence.

lemma *mod-mult-left-eq*: $(a * b)$ *mod* $c = ((a$ *mod* $c) * b)$ *mod* c
proof −
have $(a * b)$ *mod* $c = ((a$ *div* $c * c + a$ *mod* $c) * b)$ *mod* c
by (*simp only*: *mod-div-equality*)
also have ... $= (a$ *mod* $c * b + a$ *div* $c * b * c)$ *mod* c
by (*simp only*: *algebra-simps*)
also have ... $= (a$ *mod* $c * b)$ *mod* c
by (*rule mod-mult-self1*)
finally show *?thesis* **.**
qed

lemma *mod-mult-right-eq*: $(a * b)$ *mod* $c = (a * (b$ *mod* $c))$ *mod* c
proof −
have $(a * b)$ *mod* $c = (a * (b$ *div* $c * c + b$ *mod* $c))$ *mod* c
by (*simp only*: *mod-div-equality*)
also have ... $= (a * (b$ *mod* $c) + a * (b$ *div* $c) * c)$ *mod* c
by (*simp only*: *algebra-simps*)
also have ... $= (a * (b$ *mod* $c))$ *mod* c
by (*rule mod-mult-self1*)
finally show *?thesis* **.**
qed

lemma *mod-mult-eq*: $(a * b)$ *mod* $c = ((a$ *mod* $c) * (b$ *mod* $c))$ *mod* c
by (*rule trans* [*OF mod-mult-left-eq mod-mult-right-eq*])

lemma *mod-mult-cong*:
assumes a *mod* $c = a'$ *mod* c
assumes b *mod* $c = b'$ *mod* c
shows $(a * b)$ *mod* $c = (a' * b')$ *mod* c
proof −
have $(a$ *mod* $c * (b$ *mod* $c))$ *mod* $c = (a'$ *mod* $c * (b'$ *mod* $c))$ *mod* c
unfolding *assms* **..**
thus *?thesis*
by (*simp only*: *mod-mult-eq* [*symmetric*])
qed

lemma *mod-mod-cancel*:
assumes *c dvd b*
shows *a mod b mod c = a mod c*
proof −
from ‹*c dvd b*› **obtain** *k* **where** *b = c * k*
by (*rule dvdE*)
have *a mod b mod c = a mod (c * k) mod c*
by (*simp only*: ‹*b = c * k*›)
also have ... = (*a mod (c * k) + a div (c * k) * k * c*) *mod c*
by (*simp only*: *mod-mult-self1*)
also have ... = (*a div (c * k) * (c * k) + a mod (c * k*)) *mod c*
by (*simp only*: *add-ac mult-ac*)
also have ... = *a mod c*
by (*simp only*: *mod-div-equality*)
finally show *?thesis* **.**
qed

end

lemma *div-mult-div-if-dvd*: (*y*::*'a*::{*semiring-div*,*no-zero-divisors*}) *dvd x* ⟹
z dvd w ⟹ (*x div y*) * (*w div z*) = (*x * w*) *div* (*y * z*)
unfolding *dvd-def*
apply *clarify*
apply (*case-tac y = 0*)
apply *simp*
apply (*case-tac z = 0*)
apply *simp*
apply (*simp add*: *algebra-simps*)
apply (*subst mult-assoc* [*symmetric*])
apply (*simp add*: *no-zero-divisors*)
done

lemma *div-power*: (*y*::*'a*::{*semiring-div*,*no-zero-divisors*,*recpower*}) *dvd x* ⟹
(*x div y*) ^*n = x* ^*n div y* ^*n*
apply (*induct n*)
apply *simp*
apply(*simp add*: *div-mult-div-if-dvd dvd-power-same*)
done

class *ring-div = semiring-div + comm-ring-1*
begin

Negation respects modular equivalence.

lemma *mod-minus-eq*: (− *a*) *mod b* = (− (*a mod b*)) *mod b*
proof −
have (− *a*) *mod b* = (− (*a div b * b + a mod b*)) *mod b*
by (*simp only*: *mod-div-equality*)
also have ... = (− (*a mod b*) + − (*a div b*) * *b*) *mod b*

by (*simp only*: *minus-add-distrib minus-mult-left add-ac*)
also have $\ldots = (-\ (a\ mod\ b))\ mod\ b$
by (*rule mod-mult-self1*)
finally show *?thesis* **.**
qed

lemma *mod-minus-cong*:
assumes $a\ mod\ b = a'\ mod\ b$
shows $(-\ a)\ mod\ b = (-\ a')\ mod\ b$
proof −
have $(-\ (a\ mod\ b))\ mod\ b = (-\ (a'\ mod\ b))\ mod\ b$
unfolding *assms* **..**
thus *?thesis*
by (*simp only*: *mod-minus-eq* [*symmetric*])
qed

Subtraction respects modular equivalence.

lemma *mod-diff-left-eq*: $(a - b)\ mod\ c = (a\ mod\ c - b)\ mod\ c$
unfolding *diff-minus*
by (*intro mod-add-cong mod-minus-cong*) *simp-all*

lemma *mod-diff-right-eq*: $(a - b)\ mod\ c = (a - b\ mod\ c)\ mod\ c$
unfolding *diff-minus*
by (*intro mod-add-cong mod-minus-cong*) *simp-all*

lemma *mod-diff-eq*: $(a - b)\ mod\ c = (a\ mod\ c - b\ mod\ c)\ mod\ c$
unfolding *diff-minus*
by (*intro mod-add-cong mod-minus-cong*) *simp-all*

lemma *mod-diff-cong*:
assumes $a\ mod\ c = a'\ mod\ c$
assumes $b\ mod\ c = b'\ mod\ c$
shows $(a - b)\ mod\ c = (a' - b')\ mod\ c$
unfolding *diff-minus* **using** *assms*
by (*intro mod-add-cong mod-minus-cong*)

lemma *dvd-neg-div*: $y\ dvd\ x \Longrightarrow -x\ div\ y = -\ (x\ div\ y)$
apply (*case-tac* $y = 0$) **apply** *simp*
apply (*auto simp add*: *dvd-def*)
apply (*subgoal-tac* $-(y * k) = y * -\ k$)
apply (*erule ssubst*)
apply (*erule div-mult-self1-is-id*)
apply *simp*
done

lemma *dvd-div-neg*: $y\ dvd\ x \Longrightarrow x\ div\ -y = -\ (x\ div\ y)$
apply (*case-tac* $y = 0$) **apply** *simp*
apply (*auto simp add*: *dvd-def*)
apply (*subgoal-tac* $y * k = -y * -k$)

```
 apply (erule ssubst)
 apply (rule div-mult-self1-is-id)
 apply simp
apply simp
done

end
```

26.3 Division on *nat*

We define *op div* and *op mod* on *nat* by means of a characteristic relation with two input arguments m, n and two output arguments q(uotient) and r(emainder).

```
definition divmod-rel :: nat ⇒ nat ⇒ nat ⇒ nat ⇒ bool where
  divmod-rel m n q r ⟷ m = q * n + r ∧ (if n > 0 then 0 ≤ r ∧ r < n else q
= 0)
```

divmod-rel is total:

```
lemma divmod-rel-ex:
  obtains q r where divmod-rel m n q r
proof (cases n = 0)
  case True with that show thesis
    by (auto simp add: divmod-rel-def)
next
  case False
  have ∃ q r. m = q * n + r ∧ r < n
  proof (induct m)
    case 0 with ‹n ≠ 0›
    have (0::nat) = 0 * n + 0 ∧ 0 < n by simp
    then show ?case by blast
  next
    case (Suc m) then obtain q' r'
      where m: m = q' * n + r' and n: r' < n by auto
    then show ?case proof (cases Suc r' < n)
      case True
      from m n have Suc m = q' * n + Suc r' by simp
      with True show ?thesis by blast
    next
      case False then have n ≤ Suc r' by auto
      moreover from n have Suc r' ≤ n by auto
      ultimately have n = Suc r' by auto
      with m have Suc m = Suc q' * n + 0 by simp
      with ‹n ≠ 0› show ?thesis by blast
    qed
  qed
  with that show thesis
    using ‹n ≠ 0› by (auto simp add: divmod-rel-def)
qed
```

divmod-rel is injective:

lemma *divmod-rel-unique-div*:
  assumes *divmod-rel m n q r*
    and *divmod-rel m n q′ r′*
  shows $q = q'$
proof (*cases* $n = 0$)
  case *True* **with** *assms* **show** *?thesis*
    by (*simp add*: *divmod-rel-def*)
next
  case *False*
  have *aux*: $\bigwedge q\ r\ q'\ r'.\ q' * n + r' = q * n + r \Longrightarrow r < n \Longrightarrow q' \leq (q::nat)$
  apply (*rule leI*)
  apply (*subst less-iff-Suc-add*)
  apply (*auto simp add*: *add-mult-distrib*)
  done
  from ‹$n \neq 0$› *assms* **show** *?thesis*
    by (*auto simp add*: *divmod-rel-def*
      intro: *order-antisym dest*: *aux sym*)
qed

lemma *divmod-rel-unique-mod*:
  assumes *divmod-rel m n q r*
    and *divmod-rel m n q′ r′*
  shows $r = r'$
proof −
  from *assms* **have** $q = q'$ **by** (*rule divmod-rel-unique-div*)
  with *assms* **show** *?thesis* **by** (*simp add*: *divmod-rel-def*)
qed

We instantiate divisibility on the natural numbers by means of *divmod-rel*:

instantiation *nat* :: *semiring-div*
begin

definition *divmod* :: $nat \Rightarrow nat \Rightarrow nat \times nat$ **where**
  [*code del*]: *divmod m n* = (*THE* (*q*, *r*). *divmod-rel m n q r*)

definition *div-nat* **where**
  m div n = *fst* (*divmod m n*)

definition *mod-nat* **where**
  m mod n = *snd* (*divmod m n*)

lemma *divmod-div-mod*:
  divmod m n = (*m div n*, *m mod n*)
  unfolding *div-nat-def mod-nat-def* **by** *simp*

lemma *divmod-eq*:
  assumes *divmod-rel m n q r*
  shows *divmod m n* = (*q*, *r*)

using *assms* **by** (*auto simp add*: *divmod-def*
dest: *divmod-rel-unique-div divmod-rel-unique-mod*)

lemma *div-eq*:
assumes *divmod-rel m n q r*
shows *m div n = q*
using *assms* **by** (*auto dest*: *divmod-eq simp add*: *div-nat-def*)

lemma *mod-eq*:
assumes *divmod-rel m n q r*
shows *m mod n = r*
using *assms* **by** (*auto dest*: *divmod-eq simp add*: *mod-nat-def*)

lemma *divmod-rel*: *divmod-rel m n* (*m div n*) (*m mod n*)
proof −
from *divmod-rel-ex*
obtain *q r* **where** *rel*: *divmod-rel m n q r* **.**
moreover with *div-eq mod-eq* **have** *m div n = q* **and** *m mod n = r*
by *simp-all*
ultimately show *?thesis* **by** *simp*
qed

lemma *divmod-zero*:
divmod m 0 = (*0*, *m*)
proof −
from *divmod-rel* [*of m 0*] **show** *?thesis*
unfolding *divmod-div-mod divmod-rel-def* **by** *simp*
qed

lemma *divmod-base*:
assumes *m < n*
shows *divmod m n* = (*0*, *m*)
proof −
from *divmod-rel* [*of m n*] **show** *?thesis*
unfolding *divmod-div-mod divmod-rel-def*
using *assms* **by** (*cases m div n = 0*)
(*auto simp add*: *gr0-conv-Suc* [*of m div n*])
qed

lemma *divmod-step*:
assumes *0 < n* **and** *n ≤ m*
shows *divmod m n* = (*Suc* ((*m − n*) *div n*), (*m − n*) *mod n*)
proof −
from *divmod-rel* **have** *divmod-m-n*: *divmod-rel m n* (*m div n*) (*m mod n*) **.**
with *assms* **have** *m-div-n*: *m div n ≥ 1*
by (*cases m div n*) (*auto simp add*: *divmod-rel-def*)
from *assms divmod-m-n* **have** *divmod-rel* (*m − n*) *n* (*m div n − Suc 0*) (*m mod n*)
by (*cases m div n*) (*auto simp add*: *divmod-rel-def*)

with *divmod-eq* **have** *divmod* $(m - n)$ $n = (m$ *div* $n -$ *Suc* 0, m *mod* $n)$ **by** *simp*
moreover from *divmod-div-mod* **have** *divmod* $(m - n)$ $n = ((m - n)$ *div* n, $(m - n)$ *mod* $n)$ **.**
ultimately have m *div* $n =$ *Suc* $((m - n)$ *div* $n)$
and m *mod* $n = (m - n)$ *mod* n **using** *m-div-n* **by** *simp-all*
then show *?thesis* **using** *divmod-div-mod* **by** *simp*
qed

The "recursion" equations for *op div* and *op mod*

lemma *div-less* [*simp*]:
fixes m n :: *nat*
assumes $m < n$
shows m *div* $n = 0$
using *assms divmod-base divmod-div-mod* **by** *simp*

lemma *le-div-geq*:
fixes m n :: *nat*
assumes $0 < n$ **and** $n \leq m$
shows m *div* $n =$ *Suc* $((m - n)$ *div* $n)$
using *assms divmod-step divmod-div-mod* **by** *simp*

lemma *mod-less* [*simp*]:
fixes m n :: *nat*
assumes $m < n$
shows m *mod* $n = m$
using *assms divmod-base divmod-div-mod* **by** *simp*

lemma *le-mod-geq*:
fixes m n :: *nat*
assumes $n \leq m$
shows m *mod* $n = (m - n)$ *mod* n
using *assms divmod-step divmod-div-mod* **by** (*cases* $n = 0$) *simp-all*

instance proof
fix m n :: *nat* **show** m *div* $n * n + m$ *mod* $n = m$
using *divmod-rel* [*of* m n] **by** (*simp add*: *divmod-rel-def*)
next
fix n :: *nat* **show** n *div* $0 = 0$
using *divmod-zero divmod-div-mod* [*of* n 0] **by** *simp*
next
fix n :: *nat* **show** 0 *div* $n = 0$
using *divmod-rel* [*of* 0 n] **by** (*cases* n) (*simp-all add*: *divmod-rel-def*)
next
fix m n q :: *nat* **assume** $n \neq 0$ **then show** $(q + m * n)$ *div* $n = m + q$ *div* n
by (*induct* m) (*simp-all add*: *le-div-geq*)
qed

end

Simproc for cancelling *op div* and *op mod*

ML ⟪
structure CancelDivModData =
struct

val div-name = @{const-name div};
val mod-name = @{const-name mod};
val mk-binop = HOLogic.mk-binop;
val mk-sum = Nat-Arith.mk-sum;
val dest-sum = Nat-Arith.dest-sum;

(∗logic∗)

val div-mod-eqs = map mk-meta-eq [@{thm div-mod-equality}, @{thm div-mod-equality2}]

val trans = trans

val prove-eq-sums =
let val simps = @{thm add-0} :: @{thm add-0-right} :: @{thms add-ac}
in Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac simps) end;

end;

structure CancelDivMod = CancelDivModFun(CancelDivModData);

val cancel-div-mod-proc = Simplifier.simproc (the-context ())
cancel-div-mod [(m::nat) + n] (K CancelDivMod.proc);

Addsimprocs[cancel-div-mod-proc];
⟫

code generator setup

lemma *divmod-if* [*code*]: *divmod m n = (if n = 0 ∨ m < n then (0, m) else*
let (q, r) = divmod (m − n) n in (Suc q, r))
by (*simp add*: *divmod-zero divmod-base divmod-step*)
(*simp add*: *divmod-div-mod*)

code-modulename *SML*
Divides Nat

code-modulename *OCaml*
Divides Nat

code-modulename *Haskell*
Divides Nat

26.3.1 Quotient

lemma *div-geq*: *0 < n ⟹ ¬ m < n ⟹ m div n = Suc ((m − n) div n)*

by (*simp add*: *le-div-geq linorder-not-less*)

lemma *div-if*: $0 < n \Longrightarrow m\ div\ n = (if\ m < n\ then\ 0\ else\ Suc\ ((m - n)\ div\ n))$
by (*simp add*: *div-geq*)

lemma *div-mult-self-is-m* [*simp*]: $0<n ==> (m*n)\ div\ n = (m::nat)$
by *simp*

lemma *div-mult-self1-is-m* [*simp*]: $0<n ==> (n*m)\ div\ n = (m::nat)$
by *simp*

26.3.2 Remainder

lemma *mod-less-divisor* [*simp*]:
fixes $m\ n :: nat$
assumes $n > 0$
shows $m\ mod\ n < (n::nat)$
using *assms divmod-rel* **unfolding** *divmod-rel-def* **by** *auto*

lemma *mod-less-eq-dividend* [*simp*]:
fixes $m\ n :: nat$
shows $m\ mod\ n \leq m$
proof (*rule add-leD2*)
from *mod-div-equality* **have** $m\ div\ n * n + m\ mod\ n = m$ **.**
then show $m\ div\ n * n + m\ mod\ n \leq m$ **by** *auto*
qed

lemma *mod-geq*: $\neg\ m < (n::nat) \Longrightarrow m\ mod\ n = (m - n)\ mod\ n$
by (*simp add*: *le-mod-geq linorder-not-less*)

lemma *mod-if*: $m\ mod\ (n::nat) = (if\ m < n\ then\ m\ else\ (m - n)\ mod\ n)$
by (*simp add*: *le-mod-geq*)

lemma *mod-1* [*simp*]: $m\ mod\ Suc\ 0 = 0$
by (*induct m*) (*simp-all add*: *mod-geq*)

lemma *mod-mult-distrib*: $(m\ mod\ n) * (k::nat) = (m * k)\ mod\ (n * k)$
apply (*cases* $n = 0$, *simp*)
apply (*cases* $k = 0$, *simp*)
apply (*induct m rule*: *nat-less-induct*)
apply (*subst mod-if*, *simp*)
apply (*simp add*: *mod-geq diff-mult-distrib*)
done

lemma *mod-mult-distrib2*: $(k::nat) * (m\ mod\ n) = (k*m)\ mod\ (k*n)$
by (*simp add*: *mult-commute* [*of k*] *mod-mult-distrib*)

lemma *mult-div-cancel*: $(n::nat) * (m\ div\ n) = m - (m\ mod\ n)$

by (*cut-tac* $a = m$ **and** $b = n$ **in** *mod-div-equality2*, *arith*)

lemma *mod-le-divisor*[*simp*]: $0 < n \Longrightarrow m \bmod n \leq (n::nat)$
apply (*drule mod-less-divisor* [**where** $m = m$])
apply *simp*
done

26.3.3 Quotient and Remainder

lemma *divmod-rel-mult1-eq*:
[| *divmod-rel b c q r*; $c > 0$ |]
==> *divmod-rel* $(a*b)$ c $(a*q + a*r \text{ div } c)$ $(a*r \bmod c)$
by (*auto simp add*: *split-ifs divmod-rel-def algebra-simps*)

lemma *div-mult1-eq*: $(a*b) \text{ div } c = a*(b \text{ div } c) + a*(b \bmod c) \text{ div } (c::nat)$
apply (*cases* $c = 0$, *simp*)
apply (*blast intro*: *divmod-rel* [*THEN divmod-rel-mult1-eq*, *THEN div-eq*])
done

lemma *divmod-rel-add1-eq*:
[| *divmod-rel a c aq ar*; *divmod-rel b c bq br*; $c > 0$ |]
==> *divmod-rel* $(a + b)$ c $(aq + bq + (ar+br)) \text{ div } c)$ $((ar + br) \bmod c)$
by (*auto simp add*: *split-ifs divmod-rel-def algebra-simps*)

lemma *div-add1-eq*:
$(a+b) \text{ div } (c::nat) = a \text{ div } c + b \text{ div } c + ((a \bmod c + b \bmod c) \text{ div } c)$
apply (*cases* $c = 0$, *simp*)
apply (*blast intro*: *divmod-rel-add1-eq* [*THEN div-eq*] *divmod-rel*)
done

lemma *mod-lemma*: [| $(0::nat) < c$; $r < b$ |] ==> $b * (q \bmod c) + r < b * c$
apply (*cut-tac* $m = q$ **and** $n = c$ **in** *mod-less-divisor*)
apply (*drule-tac* [2] $m = q \bmod c$ **in** *less-imp-Suc-add*, *auto*)
apply (*erule-tac* $P = \%x.\ ?lhs < ?rhs\ x$ **in** *ssubst*)
apply (*simp add*: *add-mult-distrib2*)
done

lemma *divmod-rel-mult2-eq*: [| *divmod-rel a b q r*; $0 < b$; $0 < c$ |]
==> *divmod-rel* a $(b*c)$ $(q \text{ div } c)$ $(b*(q \bmod c) + r)$
by (*auto simp add*: *mult-ac divmod-rel-def add-mult-distrib2* [*symmetric*] *mod-lemma*)

lemma *div-mult2-eq*: $a \text{ div } (b*c) = (a \text{ div } b) \text{ div } (c::nat)$
apply (*cases* $b = 0$, *simp*)
apply (*cases* $c = 0$, *simp*)
apply (*force simp add*: *divmod-rel* [*THEN divmod-rel-mult2-eq*, *THEN div-eq*])
done

lemma *mod-mult2-eq*: $a \bmod (b*c) = b*(a \text{ div } b \bmod c) + a \bmod (b::nat)$

apply (*cases b = 0*, *simp*)
apply (*cases c = 0*, *simp*)
apply (*auto simp add*: *mult-commute divmod-rel* [*THEN divmod-rel-mult2-eq*, *THEN mod-eq*])
done

26.3.4 Cancellation of Common Factors in Division

lemma *div-mult-mult-lemma*:
[| (*0*::*nat*) < *b*; *0* < *c* |] ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
by (*auto simp add*: *div-mult2-eq*)

lemma *div-mult-mult1* [*simp*]: (*0*::*nat*) < *c* ==> (*c*∗*a*) *div* (*c*∗*b*) = *a div b*
apply (*cases b = 0*)
apply (*auto simp add*: *linorder-neq-iff* [*of b*] *div-mult-mult-lemma*)
done

lemma *div-mult-mult2* [*simp*]: (*0*::*nat*) < *c* ==> (*a*∗*c*) *div* (*b*∗*c*) = *a div b*
apply (*drule div-mult-mult1*)
apply (*auto simp add*: *mult-commute*)
done

26.3.5 Further Facts about Quotient and Remainder

lemma *div-1* [*simp*]: *m div Suc 0* = *m*
by (*induct m*) (*simp-all add*: *div-geq*)

lemma *div-le-mono* [*rule-format*]:
∀ *m*::*nat*. *m* ≤ *n* −−> (*m div k*) ≤ (*n div k*)
apply (*case-tac k=0*, *simp*)
apply (*induct n rule*: *nat-less-induct*, *clarify*)
apply (*case-tac n<k*)

apply *simp*

apply (*case-tac m<k*)

apply *simp*

apply (*simp add*: *div-geq diff-le-mono*)
done

lemma *div-le-mono2*: !!*m*::*nat*. [| *0*<*m*; *m*≤*n* |] ==> (*k div n*) ≤ (*k div m*)
apply (*subgoal-tac 0*<*n*)
prefer *2* **apply** *simp*
apply (*induct-tac k rule*: *nat-less-induct*)
apply (*rename-tac k*)

apply (*case-tac k<n*, *simp*)
apply (*subgoal-tac* ~ (*k<m*))
prefer *2* **apply** *simp*
apply (*simp add*: *div-geq*)
apply (*subgoal-tac* (*k−n*) *div n* ≤ (*k−m*) *div n*)
prefer *2*
apply (*blast intro*: *div-le-mono diff-le-mono2*)
apply (*rule le-trans*, *simp*)
apply (*simp*)
done

lemma *div-le-dividend* [*simp*]: *m div n* ≤ (*m*::*nat*)
apply (*case-tac n=0*, *simp*)
apply (*subgoal-tac m div n* ≤ *m div 1*, *simp*)
apply (*rule div-le-mono2*)
apply (*simp-all* (*no-asm-simp*))
done

lemma *div-less-dividend* [*rule-format*]:
!!*n*::*nat*. *1<n* ==> *0* < *m* −−> *m div n* < *m*
apply (*induct-tac m rule*: *nat-less-induct*)
apply (*rename-tac m*)
apply (*case-tac m<n*, *simp*)
apply (*subgoal-tac 0<n*)
prefer *2* **apply** *simp*
apply (*simp add*: *div-geq*)
apply (*case-tac n<m*)
apply (*subgoal-tac* (*m−n*) *div n* < (*m−n*))
apply (*rule impI less-trans-Suc*)+
apply *assumption*
apply (*simp-all*)
done

lemma *nat-div-eq-0* [*simp*]: (*n*::*nat*) > *0* ==> ((*m div n*) = *0*) = (*m* < *n*)
by(*auto*, *subst mod-div-equality* [*of m n*, *symmetric*], *auto*)

lemma *nat-div-gt-0* [*simp*]: (*n*::*nat*) > *0* ==> ((*m div n*) > *0*) = (*m* >= *n*)
by (*subst neq0-conv* [*symmetric*], *auto*)

declare *div-less-dividend* [*simp*]

A fact for the mutilated chess board

lemma *mod-Suc*: *Suc*(*m*) *mod n* = (*if Suc*(*m mod n*) = *n then 0 else Suc*(*m mod n*))
apply (*case-tac n=0*, *simp*)
apply (*induct m rule*: *nat-less-induct*)
apply (*case-tac Suc* (*na*) <*n*)

apply (*frule lessI* [*THEN less-trans*], *simp add*: *less-not-refl3*)

apply (*simp add*: *linorder-not-less le-Suc-eq mod-geq*)
apply (*auto simp add*: *Suc-diff-le le-mod-geq*)
done

26.3.6 The Divides Relation

lemma *dvd-1-left* [*iff*]: *Suc 0 dvd k*
unfolding *dvd-def* **by** *simp*

lemma *dvd-1-iff-1* [*simp*]: (*m dvd Suc 0*) = (*m = Suc 0*)
by (*simp add*: *dvd-def*)

lemma *nat-dvd-1-iff-1* [*simp*]: *m dvd* (*1*::*nat*) ⟷ *m = 1*
by (*simp add*: *dvd-def*)

lemma *dvd-anti-sym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n*::*nat*)
unfolding *dvd-def*
by (*force dest*: *mult-eq-self-implies-10 simp add*: *mult-assoc mult-eq-1-iff*)

op dvd is a partial order

interpretation *dvd*: *order op dvd* λ*n m* :: *nat*. *n dvd m* ∧ ¬ *m dvd n*
proof qed (*auto intro*: *dvd-refl dvd-trans dvd-anti-sym*)

lemma *nat-dvd-diff* [*simp*]: [| *k dvd m*; *k dvd n* |] ==> *k dvd* (*m*−*n* :: *nat*)
unfolding *dvd-def*
by (*blast intro*: *diff-mult-distrib2* [*symmetric*])

lemma *dvd-diffD*: [| *k dvd m*−*n*; *k dvd n*; *n*≤*m* |] ==> *k dvd* (*m*::*nat*)
apply (*erule linorder-not-less* [*THEN iffD2*, *THEN add-diff-inverse*, *THEN subst*])
apply (*blast intro*: *dvd-add*)
done

lemma *dvd-diffD1*: [| *k dvd m*−*n*; *k dvd m*; *n*≤*m* |] ==> *k dvd* (*n*::*nat*)
by (*drule-tac m* = *m* **in** *nat-dvd-diff*, *auto*)

lemma *dvd-reduce*: (*k dvd n* + *k*) = (*k dvd* (*n*::*nat*))
apply (*rule iffI*)
apply (*erule-tac* [*2*] *dvd-add*)
apply (*rule-tac* [*2*] *dvd-refl*)
apply (*subgoal-tac n* = (*n*+*k*) −*k*)
prefer *2* **apply** *simp*
apply (*erule ssubst*)
apply (*erule nat-dvd-diff*)
apply (*rule dvd-refl*)
done

lemma *dvd-mod*: !!*n*::*nat*. [| *f dvd m*; *f dvd n* |] ==> *f dvd m mod n*
unfolding *dvd-def*
apply (*case-tac n = 0*, *auto*)
apply (*blast intro*: *mod-mult-distrib2* [*symmetric*])
done

lemma *dvd-mod-iff*: *k dvd n* ==> ((*k*::*nat*) *dvd m mod n*) = (*k dvd m*)
by (*blast intro*: *dvd-mod-imp-dvd dvd-mod*)

lemma *dvd-mult-cancel*: !!*k*::*nat*. [| *k*∗*m dvd k*∗*n*; *0*<*k* |] ==> *m dvd n*
unfolding *dvd-def*
apply (*erule exE*)
apply (*simp add*: *mult-ac*)
done

lemma *dvd-mult-cancel1*: *0*<*m* ==> (*m*∗*n dvd m*) = (*n* = (*1*::*nat*))
apply *auto*
apply (*subgoal-tac m*∗*n dvd m*∗*1*)
apply (*drule dvd-mult-cancel*, *auto*)
done

lemma *dvd-mult-cancel2*: *0*<*m* ==> (*n*∗*m dvd m*) = (*n* = (*1*::*nat*))
apply (*subst mult-commute*)
apply (*erule dvd-mult-cancel1*)
done

lemma *dvd-imp-le*: [| *k dvd n*; *0* < *n* |] ==> *k* ≤ (*n*::*nat*)
by (*auto elim*!: *dvdE*) (*auto simp add*: *gr0-conv-Suc*)

lemma *nat-dvd-not-less*: (*0*::*nat*) < *m* ⟹ *m* < *n* ⟹ ¬ *n dvd m*
by (*auto elim*!: *dvdE*) (*auto simp add*: *gr0-conv-Suc*)

lemma *dvd-mult-div-cancel*: *n dvd m* ==> *n* ∗ (*m div n*) = (*m*::*nat*)
apply (*subgoal-tac m mod n = 0*)
apply (*simp add*: *mult-div-cancel*)
apply (*simp only*: *dvd-eq-mod-eq-0*)
done

lemma *nat-zero-less-power-iff* [*simp*]: (*x*^*n* > *0*) = (*x* > (*0*::*nat*) | *n*=*0*)
by (*induct n*) *auto*

lemma *power-dvd-imp-le*: [|*i*^*m dvd i*^*n*; (*1*::*nat*) < *i*|] ==> *m* ≤ *n*
apply (*rule power-le-imp-le-exp*, *assumption*)
apply (*erule dvd-imp-le*, *simp*)
done

lemma *mod-eq-0-iff*: (*m mod d = 0*) = (∃ *q*::*nat*. *m* = *d*∗*q*)
by (*auto simp add*: *dvd-eq-mod-eq-0* [*symmetric*] *dvd-def*)

lemmas *mod-eq-0D* [*dest*!] = *mod-eq-0-iff* [*THEN iffD1*]

lemma *mod-eqD*: (*m mod d* = *r*) ==> ∃ *q*::*nat*. *m* = *r* + *q*∗*d*
apply (*cut-tac a* = *m* **in** *mod-div-equality*)
apply (*simp only*: *add-ac*)
apply (*blast intro*: *sym*)
done

lemma *split-div*:
P(*n div k* :: *nat*) =
((*k* = *0* ⟶ *P 0*) ∧ (*k* ≠ *0* ⟶ (!*i*. !*j*<*k*. *n* = *k*∗*i* + *j* ⟶ *P i*)))
(**is** *?P* = *?Q* **is** - = (- ∧ (- ⟶ *?R*)))
proof
assume *P*: *?P*
show *?Q*
proof (*cases*)
assume *k* = *0*
with *P* **show** *?Q* **by** *simp*
next
assume *not0*: *k* ≠ *0*
thus *?Q*
proof (*simp*, *intro allI impI*)
fix *i j*
assume *n*: *n* = *k*∗*i* + *j* **and** *j*: *j* < *k*
show *P i*
proof (*cases*)
assume *i* = *0*
with *n j P* **show** *P i* **by** *simp*
next
assume *i* ≠ *0*
with *not0 n j P* **show** *P i* **by**(*simp add*:*add-ac*)
qed
qed
qed
next
assume *Q*: *?Q*
show *?P*
proof (*cases*)
assume *k* = *0*
with *Q* **show** *?P* **by** *simp*
next
assume *not0*: *k* ≠ *0*
with *Q* **have** *R*: *?R* **by** *simp*
from *not0 R*[*THEN spec*,*of n div k*,*THEN spec*, *of n mod k*]
show *?P* **by** *simp*
qed
qed

lemma *split-div-lemma*:
assumes $0 < n$
shows $n * q \leq m \wedge m < n * Suc\ q \longleftrightarrow q = ((m::nat)\ div\ n)$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?rhs*
with *mult-div-cancel* **have** *nq*: $n * q = m - (m\ mod\ n)$ **by** *simp*
then have *A*: $n * q \leq m$ **by** *simp*
have $n - (m\ mod\ n) > 0$ **using** *mod-less-divisor assms* **by** *auto*
then have $m < m + (n - (m\ mod\ n))$ **by** *simp*
then have $m < n + (m - (m\ mod\ n))$ **by** *simp*
with *nq* **have** $m < n + n * q$ **by** *simp*
then have *B*: $m < n * Suc\ q$ **by** *simp*
from *A B* **show** *?lhs* **..**
next
assume *P*: *?lhs*
then have *divmod-rel m n q* $(m - n * q)$
unfolding *divmod-rel-def* **by** (*auto simp add*: *mult-ac*)
then show *?rhs* **using** *divmod-rel* **by** (*rule divmod-rel-unique-div*)
qed

theorem *split-div'*:
$P\ ((m::nat)\ div\ n) = ((n = 0 \wedge P\ 0) \vee$
$(\exists q.\ (n * q \leq m \wedge m < n * (Suc\ q)) \wedge P\ q))$
apply (*case-tac* $0 < n$)
apply (*simp only*: *add*: *split-div-lemma*)
apply *simp-all*
done

lemma *split-mod*:
$P(n\ mod\ k :: nat) =$
$((k = 0 \longrightarrow P\ n) \wedge (k \neq 0 \longrightarrow (!i.\ !j{<}k.\ n = k{*}i + j \longrightarrow P\ j)))$
(**is** *?P* = *?Q* **is** - = (- $\wedge$ (- $\longrightarrow$ *?R*)))
proof
assume *P*: *?P*
show *?Q*
proof (*cases*)
assume $k = 0$
with *P* **show** *?Q* **by** *simp*
next
assume *not0*: $k \neq 0$
thus *?Q*
proof (*simp*, *intro allI impI*)
fix *i j*
assume $n = k{*}i + j\ j < k$
thus *P j* **using** *not0 P* **by**(*simp add*:*add-ac mult-ac*)
qed
qed
next
assume *Q*: *?Q*

```
  show ?P
  proof (cases)
    assume k = 0
    with Q show ?P by simp
  next
    assume not0: k ≠ 0
    with Q have R: ?R by simp
    from not0 R[THEN spec,of n div k,THEN spec, of n mod k]
    show ?P by simp
  qed
qed

theorem mod-div-equality': (m::nat) mod n = m - (m div n) * n
  apply (rule-tac P=%x. m mod n = x - (m div n) * n in
    subst [OF mod-div-equality [of - n]])
  apply arith
  done

lemma div-mod-equality':
  fixes m n :: nat
  shows m div n * n = m - m mod n
proof -
  have m mod n ≤ m mod n ..
  from div-mod-equality have
    m div n * n + m mod n - m mod n = m - m mod n by simp
  with diff-add-assoc [OF ‹m mod n ≤ m mod n›, of m div n * n] have
    m div n * n + (m mod n - m mod n) = m - m mod n
    by simp
  then show ?thesis by simp
qed
```

26.3.7 An "induction" law for modulus arithmetic.

```
lemma mod-induct-0:
  assumes step: ∀ i<p. P i ⟶ P ((Suc i) mod p)
  and base: P i and i: i<p
  shows P 0
proof (rule ccontr)
  assume contra: ¬(P 0)
  from i have p: 0<p by simp
  have ∀ k. 0<k ⟶ ¬ P (p-k) (is ∀ k. ?A k)
  proof
    fix k
    show ?A k
    proof (induct k)
      show ?A 0 by simp  — by contradiction
    next
      fix n
      assume ih: ?A n
```

```
      show ?A (Suc n)
      proof (clarsimp)
        assume y: P (p − Suc n)
        have n: Suc n < p
        proof (rule ccontr)
          assume ¬(Suc n < p)
          hence p − Suc n = 0
            by simp
          with y contra show False
            by simp
        qed
        hence n2: Suc (p − Suc n) = p−n by arith
        from p have p − Suc n < p by arith
        with y step have z: P ((Suc (p − Suc n)) mod p)
          by blast
        show False
        proof (cases n=0)
          case True
          with z n2 contra show ?thesis by simp
        next
          case False
          with p have p−n < p by arith
          with z n2 False ih show ?thesis by simp
        qed
      qed
    qed
  qed
  moreover
  from i obtain k where 0<k ∧ i+k=p
    by (blast dest: less-imp-add-positive)
  hence 0<k ∧ i=p−k by auto
  moreover
  note base
  ultimately
  show False by blast
qed

lemma mod-induct:
  assumes step: ∀ i<p. P i ⟶ P ((Suc i) mod p)
  and base: P i and i: i<p and j: j<p
  shows P j
proof −
  have ∀ j<p. P j
  proof
    fix j
    show j<p ⟶ P j (is ?A j)
    proof (induct j)
      from step base i show ?A 0
        by (auto elim: mod-induct-0)
```

```
    next
      fix k
      assume ih: ?A k
      show ?A (Suc k)
      proof
        assume suc: Suc k < p
        hence k: k<p by simp
        with ih have P k ..
        with step k have P (Suc k mod p)
          by blast
        moreover
        from suc have Suc k mod p = Suc k
          by simp
        ultimately
        show P (Suc k) by simp
      qed
    qed
  qed
  with j show ?thesis by blast
qed

end
```

27 Plain: Plain HOL

theory *Plain*
imports *Datatype FunDef Record Extraction Divides*
begin

Plain bootstrap of fundamental HOL tools and packages; does not include *Hilbert-Choice*.

ML ⟪ *path-add* ~~/*src*/*HOL*/*Library* ⟫

end

References