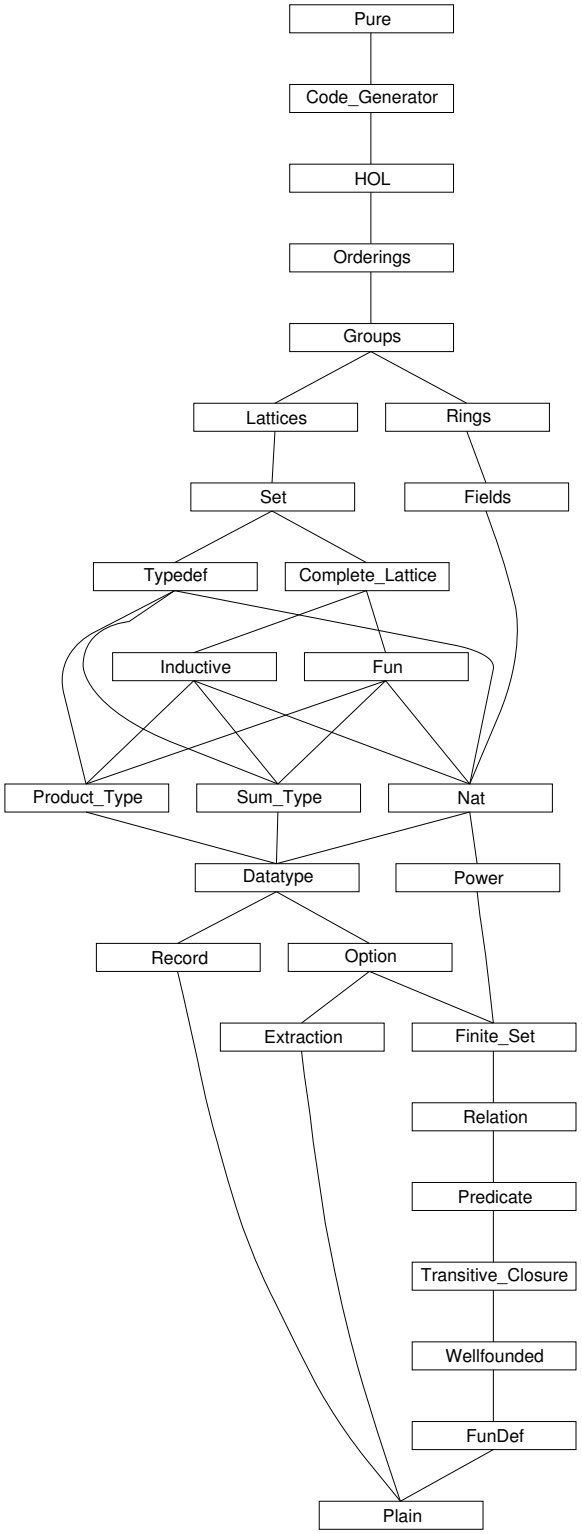
Isabelle/HOL — Higher-Order Logic

June 21, 2010

Contents


Pure
Code_Generator
HOL
Orderings
Groups
Lattices
Rings
Set
Fields
Typedef
Complete_Lattice
Inductive
Fun
Product_Type
Sum_Type
Nat
Datatype
Power
Record
Option
Extraction
Finite_Set
Relation
Predicate
Transitive_Closure
Wellfounded
FunDef
Plain

1 Code-Generator: Loading the code generator modules

theory *Code-Generator*
imports *Pure*
uses
  ~~/src/Tools/auto-solve.ML
  ~~/src/Tools/auto-counterexample.ML
  ~~/src/Tools/quickcheck.ML
  ~~/src/Tools/value.ML
  ~~/src/Tools/Code/code-preproc.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-eval.ML
  ~~/src/Tools/Code/code-haskell.ML
  ~~/src/Tools/Code/code-scala.ML
  ~~/src/Tools/nbe.ML
begin

setup ⟪
  Code-Preproc.setup
  #> *Code-ML.setup*
  #> *Code-Eval.setup*
  #> *Code-Haskell.setup*
  #> *Code-Scala.setup*
  #> *Nbe.setup*
  #> *Quickcheck.setup*
⟫

end

2 HOL: The basis of Higher-Order Logic

theory *HOL*
imports *Pure ~~/src/Tools/Code-Generator*
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/Tools/cong-tac.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)
(Tools/recfun-codegen.ML)
begin

setup ⟪ *Intuitionistic.method-setup* @{*binding iprover*} ⟫

2.1 Primitive logic

2.1.1 Core syntax

classes *type*
default-sort *type*
setup ⟪ *Object-Logic.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* (-)) [*0*, *0*, *10*] *10*)

2.1.2 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* (-)) [*0*, *10*] *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 == *CONST The* (%*x*. *P*)
-Let (*-binds b bs*) *e* == *-Let b* (*-Let bs e*)
let x = a in e == *CONST Let a* (%*x*. *e*)

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

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*)

2.1.3 Axioms and basic definitions

axioms
refl: *t* = (*t*::*'a*)
subst: *s* = *t* ⟹ *P s* ⟹ *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: $[|\ P\text{--}>Q;\ \ P\ |] ==> Q$

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

axioms
iff: $(P\text{--}>Q)\ \text{--}>\ (Q\text{--}>P)\ \text{--}>\ (P{=}Q)$
True-or-False: $(P{=}True)\ |\ (P{=}False)$

defs
Let-def [*code*]: *Let s f* $== f(s)$
if-def: *If P x y* $==$ *THE* $z{::}'a.\ (P{=}True\ \text{--}>\ z{=}x)\ \&\ (P{=}False\ \text{--}>\ z{=}y)$

finalconsts
op $=$
op $\text{--}>$
The

axiomatization
undefined $::\ 'a$

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

2.2 Fundamental rules

2.2.1 Equality

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

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

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

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

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

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

For calculational reasoning:

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

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

2.2.2 Congruence rules for application

Similar to *AP-THM* in Gordon's HOL.

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

Similar to *AP-TERM* in Gordon's HOL and FOL's *subst-context*.

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

lemma *arg-cong2*: ⟦ *a* = *b*; *c* = *d* ⟧ ⟹ *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

ML ⟪ *val cong-tac* = *Cong-Tac.cong-tac* @{*thm cong*} ⟫

2.2.3 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* ⟹ *Q* ⟹ *P*
by (*drule sym*) (*rule iffD2*)

lemma *rev-iffD1*: *Q* ⟹ *Q* = *P* ⟹ *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*])

2.2.4 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*)

2.2.5 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*])

2.2.6 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*])

2.2.7 Negation

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

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

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

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

lemma *notI2*: (*P* ⟹ ¬ *Pa*) ⟹ (*P* ⟹ *Pa*) ⟹ ¬ *P*
by (*erule notE* [*THEN notI*]) (*erule meta-mp*)

2.2.8 Implication

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

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

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

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

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

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

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

2.2.9 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

2.2.10 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*)

2.2.11 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*])

2.2.12 Classical logic

lemma *classical*:
assumes *prem*: ~*P* ==> *P*

shows P
apply (*rule True-or-False* [*THEN disjE*, *THEN eqTrueE*])
apply *assumption*
apply (*rule notI* [*THEN prem*, *THEN eqTrueI*])
apply (*erule subst*)
apply *assumption*
done

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

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

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

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

2.2.13 Unique existence

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

lemma *ex1E*:
assumes *major*: $EX!\ x.\ P(x)$
and *minor*: $!!x.\ [|\ P(x);\ \ ALL\ y.\ P(y) \longrightarrow y=x\ |] \Longrightarrow R$

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

2.2.14 THE: definite description operator

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

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

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

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

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

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

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

2.2.15 Classical intro rules for disjunction and existential quantifiers

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

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

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

lemma *case-split* [*case-names True False*]:
assumes *prem1*: *P* ==> *Q*
and *prem2*: ~*P* ==> *Q*
shows *Q*
apply (*rule excluded-middle* [*THEN disjE*])
apply (*erule prem2*)
apply (*erule prem1*)
done

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

done

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

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

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

2.2.16 Intuitionistic Reasoning

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

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

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

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

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

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

use *Tools/hologic.ML*

2.2.17 Atomizing meta-level connectives

axiomatization where
  eq-reflection: *x* = *y* ⟹ *x* ≡ *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*

2.2.18 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*
..

2.3 Package setup

2.3.1 Sledgehammer setup

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.

ML ⟪
structure No-ATPs = Named-Thms
(
  val name = no-atp
  val description = theorems that should be filtered out by Sledgehammer
)
⟫

setup ⟪ *No-ATPs.setup* ⟫

2.3.2 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 Basic-Classical: BASIC-CLASSICAL = Classical;
open Basic-Classical;

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

setup Classical.setup

setup ⟨⟨
let
  fun non-bool-eq (@{const-name op =}, Type (-, [T, -])) = T <> @{typ bool}
    | non-bool-eq - = false;
  val hyp-subst-tac' =
    SUBGOAL (fn (goal, i) =>
      if Term.exists-Const non-bool-eq goal
      then Hypsubst.hyp-subst-tac i
      else no-tac);
in
  Hypsubst.hypsubst-setup
  (*prevent substitution on bool*)
  #> Context-Rules.addSWrapper (fn tac => hyp-subst-tac' ORELSE' tac)
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*!]

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 = *Blast*
(
val thy = @{*theory*}
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*

2.3.3 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* [*code*]: $(\neg\ True) = False$
and *not-False-eq-True* [*code*]: $(\neg\ False) = True$
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**
$(ALL\ x.\ P) = P\ \ (EX\ x.\ P) = P\ \ EX\ x.\ x{=}t\ \ EX\ x.\ t{=}x$
and
$!!P.\ (EX\ x.\ x{=}t\ \&\ P(x)) = P(t)$
$!!P.\ (EX\ x.\ t{=}x\ \&\ P(x)) = P(t)$
$!!P.\ (ALL\ x.\ x{=}t \longrightarrow P(x)) = P(t)$
$!!P.\ (ALL\ x.\ t{=}x \longrightarrow P(x)) = P(t)$
by (*blast*, *blast*, *blast*, *blast*, *blast*, *iprover*+)

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

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

by *blast*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

declare *All-def* [*no-atp*]

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

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

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

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

The | congruence rule: not included by default!

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

if-then-else rules

lemma *if-True* [*code*]: $(\textit{if True then } x \textit{ else } y) = x$
by (*unfold if-def*) *blast*

lemma *if-False* [*code*]: $(\textit{if False then } x \textit{ else } y) = y$

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

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

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

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

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

lemmas *if-splits* [*no-atp*] = *split-if split-if-asm*

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

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

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

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

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

let rules for simproc

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

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

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

definition *simp-implies* :: [*prop*, *prop*] => *prop* (**infixr** =*simp*=> *1*) **where**
[*code del*]: *simp-implies* ≡ *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 ocurrence 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 = Thm.symmetric (Thm.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 [Thm.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*:
 $!!P\ Q.\ (EX\ x.\ P\ x\ \&\ Q) = ((EX\ x.\ P\ x)\ \&\ Q)$
 $!!P\ Q.\ (EX\ x.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x.\ Q\ x))$
 $!!P\ Q.\ (EX\ x.\ P\ x \mid Q) = ((EX\ x.\ P\ x) \mid Q)$
 $!!P\ Q.\ (EX\ x.\ P \mid Q\ x) = (P \mid (EX\ x.\ Q\ x))$
 $!!P\ Q.\ (EX\ x.\ P\ x \longrightarrow Q) = ((ALL\ x.\ P\ x) \longrightarrow Q)$
 $!!P\ Q.\ (EX\ x.\ P \longrightarrow Q\ x) = (P \longrightarrow (EX\ x.\ Q\ x))$
 — Miniscoping: pushing in existential quantifiers.
 by (*iprover* | *blast*)+

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

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

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

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

ML ⟨⟨ *val HOL-ss* = @{*simpset*} ⟩⟩

Simplifies x assuming c and y assuming c

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

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

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

Prevents simplification of t: much faster

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

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

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

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

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

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

2.3.4 Generic cases and induction

Rule projections:

ML ⟪
structure Project-Rule = Project-Rule
(
val conjunct1 = @{*thm conjunct1*}
val conjunct2 = @{*thm conjunct2*}
val mp = @{*thm mp*}
)
⟫

definition *induct-forall* **where**
induct-forall P == ∀ *x*. *P x*

definition *induct-implies* **where**
induct-implies A B == *A* ⟶ *B*

definition *induct-equal* **where**
induct-equal x y == *x* = *y*

definition *induct-conj* **where**
induct-conj A B == *A* ∧ *B*

definition *induct-true* **where**
induct-true == *True*

definition *induct-false* **where**
induct-false == *False*

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

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

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

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

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

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

lemma *induct-implies-conj*: $induct\text{-}implies\ C\ (induct\text{-}conj\ A\ B) =$
$induct\text{-}conj\ (induct\text{-}implies\ C\ A)\ (induct\text{-}implies\ C\ B)$
by (*unfold induct-implies-def induct-conj-def*) *iprover*

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

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

lemma *induct-trueI*: *induct-true*
by (*simp add*: *induct-true-def*)

Method setup.

ML ⟪
structure Induct = *Induct*
(
val cases-default = @{*thm case-split*}
val atomize = @{*thms induct-atomize*}
val rulify = @{*thms induct-rulify'*}
val rulify-fallback = @{*thms induct-rulify-fallback*}
val equal-def = @{*thm induct-equal-def*}
fun dest-def (*Const* (@{*const-name induct-equal*}, -) $ *t* $ *u*) = *SOME* (*t*, *u*)
| *dest-def* - = *NONE*

```
  val trivial-tac = match-tac @{thms induct-trueI}
)
⟩

setup ⟨
  Induct.setup #>
  Context.theory-map (Induct.map-simpset (fn ss => ss
    setmksimps (fn ss => Simpdata.mksimps Simpdata.mksimps-pairs ss #>
      map (Simplifier.rewrite-rule (map Thm.symmetric
        @{thms induct-rulify-fallback}))))
    addsimprocs
      [Simplifier.simproc @{theory} swap-induct-false
        [induct-false ==> PROP P ==> PROP Q]
        (fn - => fn - =>
          (fn - $ (P as - $ @{const induct-false}) $ (- $ Q $ -) =>
              if P <> Q then SOME Drule.swap-prems-eq else NONE
            | - => NONE)),
       Simplifier.simproc @{theory} induct-equal-conj-curry
        [induct-conj P Q ==> PROP R]
        (fn - => fn - =>
          (fn - $ (- $ P) $ - =>
              let
                fun is-conj (@{const induct-conj} $ P $ Q) =
                      is-conj P andalso is-conj Q
                  | is-conj (Const (@{const-name induct-equal}, -) $ - $ -) = true
                  | is-conj @{const induct-true} = true
                  | is-conj @{const induct-false} = true
                  | is-conj - = false
              in if is-conj P then SOME @{thm induct-conj-curry} else NONE end
            | - => NONE))]))
⟩
```

Pre-simplification of induction and cases rules

```
lemma [induct-simp]: (!!x. induct-equal x t ==> PROP P x) == PROP P t
  unfolding induct-equal-def
proof
  assume R: !!x. x = t ==> PROP P x
  show PROP P t by (rule R [OF refl])
next
  fix x assume PROP P t x = t
  then show PROP P x by simp
qed

lemma [induct-simp]: (!!x. induct-equal t x ==> PROP P x) == PROP P t
  unfolding induct-equal-def
proof
  assume R: !!x. t = x ==> PROP P x
  show PROP P t by (rule R [OF refl])
next
```

fix x **assume** *PROP P t t* = x
then show *PROP P x* **by** *simp*
qed

lemma [*induct-simp*]: (*induct-false* ==> *P*) == *Trueprop induct-true*
unfolding *induct-false-def induct-true-def*
by (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: (*induct-true* ==> *PROP P*) == *PROP P*
unfolding *induct-true-def*
proof
assume *R*: *True* ⟹ *PROP P*
from *TrueI* **show** *PROP P* **by** (*rule R*)
next
assume *PROP P*
then show *PROP P* **.**
qed

lemma [*induct-simp*]: (*PROP P* ==> *induct-true*) == *Trueprop induct-true*
unfolding *induct-true-def*
by (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: (!!*x*. *induct-true*) == *Trueprop induct-true*
unfolding *induct-true-def*
by (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: *induct-implies induct-true P* == *P*
by (*simp add*: *induct-implies-def induct-true-def*)

lemma [*induct-simp*]: (*x* = *x*) = *True*
by (*rule simp-thms*)

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

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

2.3.5 Coherent logic

ML ⟪
structure Coherent = *Coherent*
(
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*

2.3.6 Reorienting equalities

ML ⟨⟨
```
signature REORIENT-PROC =
sig
  val add : (term -> bool) -> theory -> theory
  val proc : morphism -> simpset -> cterm -> thm option
end;

structure Reorient-Proc : REORIENT-PROC =
struct
  structure Data = Theory-Data
  (
    type T = ((term -> bool) * stamp) list;
    val empty = [];
    val extend = I;
    fun merge data : T = Library.merge (eq-snd op =) data;
  );
  fun add m = Data.map (cons (m, stamp ()));
  fun matches thy t = exists (fn (m, -) => m t) (Data.get thy);

  val meta-reorient = @{thm eq-commute [THEN eq-reflection]};
  fun proc phi ss ct =
    let
      val ctxt = Simplifier.the-context ss;
      val thy = ProofContext.theory-of ctxt;
    in
      case Thm.term-of ct of
        (- $ t $ u) => if matches thy u then NONE else SOME meta-reorient
      | - => NONE
    end;
end;
```
⟩⟩

2.4 Other simple lemmas and lemma duplicates

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

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

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

2.5 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}
⟩⟩

2.6 Code generator setup

2.6.1 SML code generator setup

use *Tools/recfun-codegen.ML*

setup ⟨⟨
Codegen.setup
#> RecfunCodegen.setup
#> Codegen.map-unfold (K HOL-basic-ss)
⟩⟩

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* (@{*const-name op* =}, *Type* (-, [*Type* (*fun*, -), -])), -) => *NONE*
| (*Const* (@{*const-name 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* (@{*const-name 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
⟫

2.6.2 Generic code generator preprocessor setup

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

2.6.3 Equality

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

lemma *eq* [*code-unfold*, *code-inline del*]: *eq* = (*op* =)
by (*rule ext eq-equals*)+

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

lemma *equals-eq*: (*op* =) ≡ *eq*
by (*rule eq-reflection*) (*rule ext*, *rule ext*, *rule sym*, *rule eq-equals*)

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

end

declare *equals-eq* [*code*]

setup ⟪
Code-Preproc.map-pre (*fn simpset* =>
simpset addsimprocs [*Simplifier.simproc-i* @{*theory*} *eq* [@{*term op* =}]
(*fn thy* => *fn* - => *fn t as Const* (-, *T*) => *case strip-type T*
of ((*T as Type* -) :: -, -) => *SOME* @{*thm equals-eq*}
| - => *NONE*)])
⟫

2.6.4 Generic code generator foundation

Datatypes

code-datatype *True False*

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

code-datatype *prop Trueprop*

Code equations

lemma [*code*]:
shows (*True* ⟹ *PROP Q*) ≡ *PROP Q*
and (*PROP Q* ⟹ *True*) ≡ *Trueprop True*
and (*P* ⟹ *R*) ≡ *Trueprop* (*P* ⟶ *R*) **by** (*auto intro*!: *equal-intr-rule*)

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

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

lemma [*code*]:

shows (*False* ⟶ *P*) ⟷ *True*
and (*True* ⟶ *P*) ⟷ *P*
and (*P* ⟶ *False*) ⟷ ¬ *P*
and (*P* ⟶ *True*) ⟷ *True* **by** *simp-all*

instantiation *itself* :: (*type*) *eq*
begin

definition *eq-itself* :: *'a itself* ⇒ *'a itself* ⇒ *bool* **where**
eq-itself x y ⟷ *x* = *y*

instance proof
qed (*fact eq-itself-def*)

end

lemma *eq-itself-code* [*code*]:
eq-class.eq TYPE('a) TYPE('a) ⟷ *True*
by (*simp add*: *eq*)

Equality

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

setup ⟪
Sign.add-const-constraint (@{*const-name eq*}, *SOME* @{*typ 'a*::*type* ⇒ *'a* ⇒ *bool*})
⟫

lemma *equals-alias-cert*: *OFCLASS('a, eq-class)* ≡ ((*op* = :: *'a* ⇒ *'a* ⇒ *bool*) ≡ *eq*) (**is** *?ofclass* ≡ *?eq*)
proof
assume *PROP ?ofclass*
show *PROP ?eq*
by (*tactic* ⟪ *ALLGOALS* (*rtac* (*Thm.unconstrainT* @{*thm equals-eq*})) ⟫)
(*fact* ‹*PROP ?ofclass*›)
next
assume *PROP ?eq*
show *PROP ?ofclass* **proof**
qed (*simp add*: ‹*PROP ?eq*›)
qed

setup ⟪
Sign.add-const-constraint (@{*const-name eq*}, *SOME* @{*typ 'a*::*eq* ⇒ *'a* ⇒ *bool*})
⟫

setup ⟪
Nbe.add-const-alias @{*thm equals-alias-cert*}
⟫

hide-const (**open**) *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*

2.6.5 Generic code generator target languages

type bool

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

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* (-)))
  (*Scala true* **and** *false* **and** *'*! -
    and infixl *3* && **and infixl** *1* ||
    and !(*if* ((-))/ (-)/ *else* (-)))

code-reserved *SML*
  bool true false not

code-reserved *OCaml*
  bool not

code-reserved *Scala*
Boolean

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*)
(*Scala* !*error*(*undefined*))

2.6.6 Evaluation and normalization by evaluation

Avoid some named infixes in evaluation environment

code-reserved *Eval oo ooo oooo upto downto orf andf*

setup ⟪
Value.add-evaluator (*SML*, *Codegen.eval-term o ProofContext.theory-of*)
⟫

ML ⟪
structure Eval-Method =
struct

val eval-ref : (*unit* −> *bool*) *option Unsynchronized.ref* = *Unsynchronized.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-Eval.eval NONE*
(*Eval-Method.eval-ref*, *Eval-Method.eval-ref*) (*K I*) *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
```

2.7 Counterexample Search Units

2.7.1 Quickcheck

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

2.7.2 Nitpick setup

```
ML ⟨⟨
structure Nitpick-Defs = Named-Thms
(
  val name = nitpick-def
  val description = alternative definitions of constants as needed by Nitpick
)
structure Nitpick-Simps = Named-Thms
(
  val name = nitpick-simp
  val description = equational specification of constants as needed by Nitpick
)
structure Nitpick-Psimps = Named-Thms
(
  val name = nitpick-psimp
  val description = partial equational specification of constants as needed by Nitpick
)
```

```
structure Nitpick-Choice-Specs = Named-Thms
(
  val name = nitpick-choice-spec
  val description = choice specification of constants as needed by Nitpick
)
⟩⟩

setup ⟨⟨
  Nitpick-Defs.setup
  #> Nitpick-Simps.setup
  #> Nitpick-Psimps.setup
  #> Nitpick-Choice-Specs.setup
⟩⟩
```

2.8 Preprocessing for the predicate compiler

```
ML ⟨⟨
structure Predicate-Compile-Alternative-Defs = Named-Thms
(
  val name = code-pred-def
  val description = alternative definitions of constants for the Predicate Compiler
)
structure Predicate-Compile-Inline-Defs = Named-Thms
(
  val name = code-pred-inline
  val description = inlining definitions for the Predicate Compiler
)
structure Predicate-Compile-Simps = Named-Thms
(
  val name = code-pred-simp
  val description = simplification rules for the optimisations in the Predicate Compiler
)
⟩⟩

setup ⟨⟨
  Predicate-Compile-Alternative-Defs.setup
  #> Predicate-Compile-Inline-Defs.setup
  #> Predicate-Compile-Simps.setup
⟩⟩
```

2.9 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 (@{const-name 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

3 Orderings: Abstract orderings

theory *Orderings*
imports *HOL*
uses
~~/*src*/*Provers*/*order.ML*
~~/*src*/*Provers*/*quasi.ML*
begin

3.1 Syntactic orders

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

3.2 Quasi orders

class *preorder* = *ord* +
  assumes *less-le-not-le*: $x < y \longleftrightarrow x \leq y \wedge \neg (y \leq x)$
  and *order-refl* [*iff*]: $x \leq x$
  and *order-trans*: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
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*:
class.preorder $(op \geq)$ $(op >)$
proof qed (*auto simp add*: *less-le-not-le intro*: *order-trans*)

end

3.3 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* ⇒ *bool*) ⇒ *'a* (**binder** *LEAST* *10*) **where**
  Least P = (*THE x*. *P x* ∧ (∀ *y*. *P y* ⟶ *x* ≤ *y*))

lemma *Least-equality*:
  assumes *P x*
    and ⋀*y*. *P y* ⟹ *x* ≤ *y*
  shows *Least P* = *x*
unfolding *Least-def* **by** (*rule the-equality*)
  (*blast intro*: *assms antisym*)+

lemma *LeastI2-order*:
  assumes *P x*
    and ⋀*y*. *P y* ⟹ *x* ≤ *y*
    and ⋀*x*. *P x* ⟹ ∀ *y*. *P y* ⟶ *x* ≤ *y* ⟹ *Q x*
  shows *Q* (*Least P*)
unfolding *Least-def* **by** (*rule theI2*)
  (*blast intro*: *assms antisym*)+

Dual order

lemma *dual-order*:
  class.order (*op* ≥) (*op* >)
by (*intro-locales*, *rule dual-preorder*) (*unfold-locales*, *rule antisym*)

end

3.4 Linear (total) orders

class *linorder* = *order* +
  assumes *linear*: *x* ≤ *y* ∨ *y* ≤ *x*
begin

lemma *less-linear*: *x* < *y* ∨ *x* = *y* ∨ *y* < *x*
unfolding *less-le* **using** *less-le linear* **by** *blast*

lemma *le-less-linear*: *x* ≤ *y* ∨ *y* < *x*
by (*simp add*: *le-less less-linear*)

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

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

lemma *not-less*: ¬ *x* < *y* ⟷ *y* ≤ *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*:
class.linorder (*op* $\geq$) (*op* $>$)
by (*rule class.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* [*no-atp*]:
$P\ (min\ i\ j) \longleftrightarrow (i \leq j \longrightarrow P\ i) \wedge (\neg\ i \leq j \longrightarrow P\ j)$
by (*simp add*: *min-def*)

lemma *split-max* [*no-atp*]:
$P\ (max\ i\ j) \longleftrightarrow (i \leq j \longrightarrow P\ j) \wedge (\neg\ i \leq j \longrightarrow P\ i)$
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*]

3.5 Reasoning tools setup

ML ⟪

```
signature ORDERS =
sig
  val print-structures: Proof.context -> unit
  val setup: theory -> theory
  val order-tac: Proof.context -> thm list -> 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 = Generic-Data
(
  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 data = AList.join struct-eq (K fst) data;
);

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) ctxt 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 ctxt prems
    | linorder => Order-Tac.linear-tac decomp thms ctxt prems
      | - => error (Unknown kind of order ' ^ s ^ ' encountered in transitivity
reasoner.)
  end

fun order-tac ctxt prems =
  FIRST' (map (fn s => CHANGED o struct-tac s ctxt 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 - =
  Outer-Syntax.improper-command print-orders
    print order structures available to transitivity reasoner Keyword.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 (fn ctxt => SIMPLE-METHOD'
(order-tac ctxt [])))
    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.the-context ss) (Simplifier.prems-of-ss
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
⟩⟩

3.6 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),
    (@{syntax-const -All-less}, @{syntax-const -All-greater})),
    ((All-binder, impl, less-eq),
    (@{syntax-const -All-less-eq}, @{syntax-const -All-greater-eq})),
    ((Ex-binder, conj, less),
    (@{syntax-const -Ex-less}, @{syntax-const -Ex-greater})),
    ((Ex-binder, conj, less-eq),
    (@{syntax-const -Ex-less-eq}, @{syntax-const -Ex-greater-eq}))];

  fun matches-bound v t =
    (case t of
      Const (@{syntax-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 (@{syntax-const -bound}, -) $ Free (v, -),
        Const (c, -) $ (Const (d, -) $ t $ u) $ P] =>
        (case ALists.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 (*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 (*less-trans*) **show** *?thesis* **.**
qed

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

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

lemma *ord-le-eq-subst*: $a <= b \Longrightarrow f\ b = c \Longrightarrow$
$(!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y) \Longrightarrow f\ a <= c$
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 (*ord-le-eq-trans*) **show** *?thesis* **.**
qed

lemma *ord-eq-le-subst*: $a = f\ b \Longrightarrow b <= c \Longrightarrow$
$(!!x\ y.\ x <= y \Longrightarrow f\ x <= f\ y) \Longrightarrow a <= f\ c$

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

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

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

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

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

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

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

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

3.8 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 ⟷ 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 ≤ f y ⟷ x ≤ y
proof
  assume x ≤ y
  with assms strict-mono-mono monoD show f x ≤ f y by auto
next
  assume f x ≤ f y
  show x ≤ y proof (rule ccontr)
    assume ¬ x ≤ y then have y < x by simp
    with assms strict-monoD have f y < f x by auto
    with ‹f x ≤ f y› show False by simp
  qed
qed

lemma strict-mono-less:
  assumes strict-mono f
  shows f x < f y ⟷ 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 ⇒ 'b::linorder
  shows mono f ⟹ min (f m) (f n) = f (min m n)
  by (auto simp: mono-def Orderings.min-def min-def intro: Orderings.antisym)
```

lemma *max-of-mono*:
fixes $f :: 'a \Rightarrow 'b$::*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*: *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*: *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-linorder* = *linorder* +
assumes *gt-ex*: $\exists y.\ x < y$
and *lt-ex*: $\exists y.\ y < x$
and *dense*: $x < y \Longrightarrow (\exists z.\ x < z \wedge z < y)$
begin

lemma *dense-le*:
fixes $y\ z :: 'a$
assumes $\bigwedge x.\ x < y \Longrightarrow x \leq z$
shows $y \leq z$
proof (*rule ccontr*)
assume $\neg$ *?thesis*
hence $z < y$ **by** *simp*
from *dense*[*OF this*]

```
  obtain x where x < y and z < x by safe
  moreover have x ≤ z using assms[OF ‹x < y›] .
  ultimately show False by auto
qed

lemma dense-le-bounded:
  fixes x y z :: 'a
  assumes x < y
  assumes *: ⋀w. ⟦ x < w ; w < y ⟧ ⟹ w ≤ z
  shows y ≤ z
proof (rule dense-le)
  fix w assume w < y
  from dense[OF ‹x < y›] obtain u where x < u u < y by safe
  from linear[of u w]
  show w ≤ z
  proof (rule disjE)
    assume u ≤ w
    from less-le-trans[OF ‹x < u› ‹u ≤ w›] ‹w < y›
    show w ≤ z by (rule *)
  next
    assume w ≤ u
    from ‹w ≤ u› *[OF ‹x < u› ‹u < y›]
    show w ≤ z by (rule order-trans)
  qed
qed

end
```

3.11 Wellorders

```
class wellorder = linorder +
  assumes less-induct [case-names less]: (⋀x. (⋀y. y < x ⟹ P y) ⟹ P x) ⟹
P a
begin

lemma wellorder-Least-lemma:
  fixes k :: 'a
  assumes P k
  shows LeastI: P (LEAST x. P x) and Least-le: (LEAST x. P x) ≤ k
proof −
  have P (LEAST x. P x) ∧ (LEAST x. P x) ≤ 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: ¬ (P (LEAST a. P a) ∧ (LEAST a. P a) ≤ x)
      have ⋀y. P y ⟹ x ≤ y
      proof (rule classical)
        fix y
        assume P y and ¬ x ≤ y
```

```
      with less have P (LEAST a. P a) and (LEAST a. P a) ≤ y
        by (auto simp add: not-le)
      with assm have x < (LEAST a. P a) and (LEAST a. P a) ≤ y
        by auto
      then show x ≤ 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) ≤ k by auto
qed

— The following 3 lemmas are due to Brian Huffman
lemma LeastI-ex: ∃ x. P x ⟹ P (Least P)
 by (erule exE) (erule LeastI)

lemma LeastI2:
 P a ⟹ (⋀x. P x ⟹ Q x) ⟹ Q (Least P)
 by (blast intro: LeastI)

lemma LeastI2-ex:
 ∃ a. P a ⟹ (⋀x. P x ⟹ Q x) ⟹ Q (Least P)
 by (blast intro: LeastI-ex)

lemma not-less-Least: k < (LEAST x. P x) ⟹ ¬ P k
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 ≤ Q ⟷ P ⟶ Q

definition
 less-bool-def [code del]: (P::bool) < Q ⟷ ¬ P ∧ Q

definition
 top-bool-eq: top = True

definition
```

bot-bool-eq: $bot = False$

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

end

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

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

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

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

lemma *bot-boolE*: $bot \Longrightarrow P$
by (*simp add*: *bot-bool-eq*)

lemma *top-boolI*: *top*
by (*simp add*: *top-bool-eq*)

lemma [*code*]:
$False \leq b \longleftrightarrow True$
$True \leq b \longleftrightarrow b$
$False < b \longleftrightarrow b$
$True < b \longleftrightarrow False$
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 antisym intro*!: *ext*)

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

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

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

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*

3.14 Name duplicates

lemmas *order-eq-refl* = *preorder-class.eq-refl*
lemmas *order-less-irrefl* = *preorder-class.less-irrefl*
lemmas *order-less-imp-le* = *preorder-class.less-imp-le*
lemmas *order-less-not-sym* = *preorder-class.less-not-sym*
lemmas *order-less-asym* = *preorder-class.less-asym*
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 *order-less-le = order-class.less-le*
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-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 = order-class.antisym*
lemmas *order-eq-iff = order-class.eq-iff*
lemmas *order-antisym-conv = order-class.antisym-conv*

lemmas *linorder-linear = linorder-class.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*

end

4 Groups: Groups, also combined with orderings

theory *Groups*
imports *Orderings*
uses (~~/*src*/*Provers*/*Arith*/*abel-cancel.ML*)
begin

4.1 Fact collections

ML ⟪
structure Ac-Simps = Named-Thms(
  val name = ac-simps
  val description = associativity and commutativity simplification rules
)
⟫

setup *Ac-Simps.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*.

ML ⟪
structure Algebra-Simps = Named-Thms(
  val name = algebra-simps
  val description = algebra simplification rules
)
⟫

setup *Algebra-Simps.setup*

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*.

ML ⟪
structure Field-Simps = Named-Thms(
  val name = field-simps
  val description = algebra simplification rules for fields
)
⟫

setup *Field-Simps.setup*

4.2 Abstract structures

These locales provide basic structures for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

locale *semigroup* =
  fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $*$ *70*)
  assumes *assoc* [*ac-simps*]: $a * b * c = a * (b * c)$

locale *abel-semigroup* = *semigroup* +
  assumes *commute* [*ac-simps*]: $a * b = b * a$
begin

lemma *left-commute* [*ac-simps*]:
  $b * (a * c) = a * (b * c)$
proof −
  have $(b * a) * c = (a * b) * c$
    by (*simp only*: *commute*)
  then show *?thesis*
    by (*simp only*: *assoc*)
qed

end

locale *monoid* = *semigroup* +
  fixes *z* :: *'a* (*1*)
  assumes *left-neutral* [*simp*]: *1 * a = a*
  assumes *right-neutral* [*simp*]: *a * 1 = a*

locale *comm-monoid* = *abel-semigroup* +
  fixes *z* :: *'a* (*1*)
  assumes *comm-neutral*: *a * 1 = a*

sublocale *comm-monoid* < *monoid* **proof**
qed (*simp-all add*: *commute comm-neutral*)

4.3 Generic operations

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

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

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

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

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

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

setup ⟪
  Reorient-Proc.add
    (*fn Const*(@{*const-name Groups.zero*}, -) => *true*
      | *Const*(@{*const-name Groups.one*}, -) => *true*
      | - => *false*)
⟫

simproc-setup *reorient-zero* (*0 = x*) = *Reorient-Proc.proc*
simproc-setup *reorient-one* (*1 = x*) = *Reorient-Proc.proc*

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 Groups.one*}, @{*const-syntax Groups.zero*}] *end*;
⟩⟩ — show types that are presumably too general

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*)

use ~~/*src*/*Provers*/*Arith*/*abel-cancel.ML*

4.4 Semigroups and Monoids

class *semigroup-add* = *plus* +
assumes *add-assoc* [*algebra-simps*, *field-simps*]: (*a* + *b*) + *c* = *a* + (*b* + *c*)

sublocale *semigroup-add* < *add*!: *semigroup plus* **proof**
qed (*fact add-assoc*)

class *ab-semigroup-add* = *semigroup-add* +
assumes *add-commute* [*algebra-simps*, *field-simps*]: *a* + *b* = *b* + *a*

sublocale *ab-semigroup-add* < *add*!: *abel-semigroup plus* **proof**
qed (*fact add-commute*)

context *ab-semigroup-add*
begin

lemmas *add-left-commute* [*algebra-simps*, *field-simps*] = *add.left-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*, *field-simps*]: (*a* ∗ *b*) ∗ *c* = *a* ∗ (*b* ∗ *c*)

sublocale *semigroup-mult* < *mult*!: *semigroup times* **proof**
qed (*fact mult-assoc*)

class *ab-semigroup-mult* = *semigroup-mult* +
  assumes *mult-commute* [*algebra-simps*, *field-simps*]: $a * b = b * a$

sublocale *ab-semigroup-mult* < *mult*!: *abel-semigroup times* **proof**
qed (*fact mult-commute*)

context *ab-semigroup-mult*
begin

lemmas *mult-left-commute* [*algebra-simps*, *field-simps*] = *mult.left-commute*

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

end

theorems *mult-ac* = *mult-assoc mult-commute mult-left-commute*

class *monoid-add* = *zero* + *semigroup-add* +
  assumes *add-0-left*: $0 + a = a$
    and *add-0-right*: $a + 0 = a$

sublocale *monoid-add* < *add*!: *monoid plus 0* **proof**
qed (*fact add-0-left add-0-right*)+

lemma *zero-reorient*: $0 = x \longleftrightarrow x = 0$
by (*rule eq-commute*)

class *comm-monoid-add* = *zero* + *ab-semigroup-add* +
  assumes *add-0*: $0 + a = a$

sublocale *comm-monoid-add* < *add*!: *comm-monoid plus 0* **proof**
qed (*insert add-0*, *simp add*: *ac-simps*)

subclass (**in** *comm-monoid-add*) *monoid-add* **proof**
qed (*fact add.left-neutral add.right-neutral*)+

class *monoid-mult* = *one* + *semigroup-mult* +
  assumes *mult-1-left*: $1 * a = a$
    and *mult-1-right*: $a * 1 = a$

sublocale *monoid-mult* < *mult*!: *monoid times 1* **proof**
qed (*fact mult-1-left mult-1-right*)+

lemma *one-reorient*: $1 = x \longleftrightarrow x = 1$
by (*rule eq-commute*)

class *comm-monoid-mult* = *one* + *ab-semigroup-mult* +
  assumes *mult-1*: *1* ∗ *a* = *a*

sublocale *comm-monoid-mult* < *mult*!: *comm-monoid times 1* **proof**
qed (*insert mult-1*, *simp add*: *ac-simps*)

subclass (**in** *comm-monoid-mult*) *monoid-mult* **proof**
qed (*fact mult.left-neutral mult.right-neutral*)+

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*

4.5 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-unique*:
assumes $a + b = 0$ **shows** $- a = b$
proof −
have $- a = - a + (a + b)$ **using** *assms* **by** *simp*
also have $\ldots = b$ **by** (*simp add*: *add-assoc* [*symmetric*])
finally show *?thesis* **.**
qed

lemmas *equals-zero-I* = *minus-unique*

lemma *minus-zero* [*simp*]: $- 0 = 0$
proof −
have $0 + 0 = 0$ **by** (*rule add-0-right*)
thus $- 0 = 0$ **by** (*rule minus-unique*)
qed

lemma *minus-minus* [*simp*]: $- (- a) = a$
proof −
have $- a + a = 0$ **by** (*rule left-minus*)
thus $- (- a) = a$ **by** (*rule minus-unique*)
qed

lemma *right-minus* [*simp*]: $a + - a = 0$
proof −
have $a + - a = - (- a) + - a$ **by** *simp*
also have $\ldots = 0$ **by** (*rule left-minus*)
finally show *?thesis* **.**
qed

lemma *minus-add-cancel*: $- a + (a + b) = b$
by (*simp add*: *add-assoc* [*symmetric*])

lemma *add-minus-cancel*: $a + (- a + b) = b$
by (*simp add*: *add-assoc* [*symmetric*])

lemma *minus-add*: $- (a + b) = - b + - a$
proof −
have $(a + b) + (- b + - a) = 0$
by (*simp add*: *add-assoc add-minus-cancel*)
thus $- (a + b) = - b + - a$
by (*rule minus-unique*)
qed

lemma *right-minus-eq*: $a - b = 0 \longleftrightarrow a = b$
proof
assume $a - b = 0$
have $a = (a - b) + b$ **by** (*simp add*:*diff-minus add-assoc*)

also have $\ldots = b$ **using** ‹$a - b = 0$› **by** *simp*
finally show $a = b$ **.**
next
assume $a = b$ **thus** $a - b = 0$ **by** (*simp add*: *diff-minus*)
qed

lemma *diff-self* [*simp*]: $a - a = 0$
by (*simp add*: *diff-minus*)

lemma *diff-0* [*simp*]: $0 - a = - a$
by (*simp add*: *diff-minus*)

lemma *diff-0-right* [*simp*]: $a - 0 = a$
by (*simp add*: *diff-minus*)

lemma *diff-minus-eq-add* [*simp*]: $a - - b = a + b$
by (*simp add*: *diff-minus*)

lemma *neg-equal-iff-equal* [*simp*]:
$- a = - b \longleftrightarrow a = b$
proof
assume $- a = - b$
hence $- (- a) = - (- b)$ **by** *simp*
thus $a = b$ **by** *simp*
next
assume $a = b$
thus $- a = - b$ **by** *simp*
qed

lemma *neg-equal-0-iff-equal* [*simp*]:
$- a = 0 \longleftrightarrow a = 0$
by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

lemma *neg-0-equal-iff-equal* [*simp*]:
$0 = - a \longleftrightarrow 0 = a$
by (*subst neg-equal-iff-equal* [*symmetric*], *simp*)

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*:
$a = - b \longleftrightarrow b = - a$
proof −
have $- (- a) = - b \longleftrightarrow - a = b$ **by** (*rule neg-equal-iff-equal*)
thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

lemma *minus-equation-iff*:
$- a = b \longleftrightarrow - b = a$
proof −
have $- a = - (- b) \longleftrightarrow a = -b$ **by** (*rule neg-equal-iff-equal*)

thus *?thesis* **by** (*simp add*: *eq-commute*)
qed

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*, *field-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*, *field-simps*]:
$- a + b = b - a$
by (*simp add*:*diff-minus add-commute*)

lemma *minus-add-distrib* [*simp*]:
$- (a + b) = - a + - b$
by (*rule minus-unique*) (*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*, *field-simps*]: $a + (b - c) = (a + b) - c$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-add-eq*[*algebra-simps*, *field-simps*]: $(a - b) + c = (a + c) - b$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-eq-eq*[*algebra-simps*, *field-simps*]: $a - b = c \longleftrightarrow a = c + b$
by (*auto simp add*: *diff-minus add-assoc*)

lemma *eq-diff-eq*[*algebra-simps*, *field-simps*]: $a = c - b \longleftrightarrow a + b = c$
by (*auto simp add*: *diff-minus add-assoc*)

lemma *diff-diff-eq*[*algebra-simps*, *field-simps*]: $(a - b) - c = a - (b + c)$
by (*simp add*: *diff-minus add-ac*)

lemma *diff-diff-eq2*[*algebra-simps*, *field-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*, *no-atp*]: $a - b = 0 \longleftrightarrow a = b$
by (*simp add*: *algebra-simps*)

end

4.6 (Partially) Ordered Groups

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.

class *ordered-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 *ordered-cancel-ab-semigroup-add* =
  ordered-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 *ordered-ab-semigroup-add-imp-le* =
  ordered-cancel-ab-semigroup-add +

assumes *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
begin

lemma *add-less-imp-less-left*:
assumes *less*: $c + a < c + b$ **shows** $a < b$
proof −
from *less* **have** *le*: $c + a <= c + b$ **by** (*simp add*: *order-le-less*)
have $a <= b$
apply (*insert le*)
apply (*drule add-le-imp-le-left*)
by (*insert le*, *drule add-le-imp-le-left*, *assumption*)
moreover have $a \neq b$
proof (*rule ccontr*)
assume ~$(a \neq b)$
then have $a = b$ **by** *simp*
then have $c + a = c + b$ **by** *simp*
with *less* **show** *False***by** *simp*
qed
ultimately show $a < b$ **by** (*simp add*: *order-le-less*)
qed

lemma *add-less-imp-less-right*:
$a + c < b + c \Longrightarrow a < b$
apply (*rule add-less-imp-less-left* [*of c*])
apply (*simp add*: *add-commute*)
done

lemma *add-less-cancel-left* [*simp*]:
$c + a < c + b \longleftrightarrow a < b$
by (*blast intro*: *add-less-imp-less-left add-strict-left-mono*)

lemma *add-less-cancel-right* [*simp*]:
$a + c < b + c \longleftrightarrow a < b$
by (*blast intro*: *add-less-imp-less-right add-strict-right-mono*)

lemma *add-le-cancel-left* [*simp*]:
$c + a \leq c + b \longleftrightarrow a \leq b$
by (*auto*, *drule add-le-imp-le-left*, *simp-all add*: *add-left-mono*)

lemma *add-le-cancel-right* [*simp*]:
$a + c \leq b + c \longleftrightarrow a \leq b$
by (*simp add*: *add-commute* [*of a c*] *add-commute* [*of b c*])

lemma *add-le-imp-le-right*:
$a + c \leq b + c \Longrightarrow a \leq b$
by *simp*

lemma *max-add-distrib-left*:
$max\ x\ y + z = max\ (x + z)\ (y + z)$

unfolding *max-def* **by** *auto*

lemma *min-add-distrib-left*:
min x y + z = min (x + z) (y + z)
unfolding *min-def* **by** *auto*

end

4.7 Support for reasoning about signs

class *ordered-comm-monoid-add* =
ordered-cancel-ab-semigroup-add + comm-monoid-add
begin

lemma *add-pos-nonneg*:
assumes *0 < a* **and** *0 ≤ 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 ≤ 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* [*simp*]:
assumes *0 ≤ a* **and** *0 ≤ b* **shows** *0 ≤ a + b*
proof −
have *0 + 0 ≤ a + b*
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-neg-nonpos*:
assumes *a < 0* **and** *b ≤ 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 *ordered-ab-group-add* =
ab-group-add + *ordered-ab-semigroup-add*

begin

subclass *ordered-cancel-ab-semigroup-add* **..**

subclass *ordered-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 *ordered-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*, *field-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*, *field-simps*]: $a < c - b \longleftrightarrow a + b < c$
apply (*subst less-iff-diff-less-0* [*of* $a + b$])
apply (*subst less-iff-diff-less-0* [*of* a])
apply (*simp add*: *diff-minus add-ac*)
done

lemma *diff-le-eq*[*algebra-simps*, *field-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*, *field-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*)

end

class *linordered-ab-semigroup-add* =
  linorder + *ordered-ab-semigroup-add*

class *linordered-cancel-ab-semigroup-add* =
  linorder + *ordered-cancel-ab-semigroup-add*
begin

subclass *linordered-ab-semigroup-add* **..**

subclass *ordered-ab-semigroup-add-imp-le*
proof
  fix a b c :: $'a$
  assume *le*: $c + a <= c + b$
  show $a <= b$
  proof (*rule ccontr*)
    assume w: $\sim a \leq b$
    hence $b <= a$ **by** (*simp add*: *linorder-not-le*)
    hence *le2*: $c + b <= c + a$ **by** (*rule add-left-mono*)
    have $a = b$
      apply (*insert le*)
      apply (*insert le2*)
      apply (*drule antisym*, *simp-all*)
      done
    with w **show** *False*
      by (*simp add*: *linorder-not-le* [*symmetric*])
  qed
qed

end

class *linordered-ab-group-add* = *linorder* + *ordered-ab-group-add*

begin

subclass *linordered-cancel-ab-semigroup-add* **..**

lemma *neg-less-eq-nonneg* [*simp*]:
$- a \leq a \longleftrightarrow 0 \leq a$
proof
assume *A*: $- a \leq a$ **show** $0 \leq a$
proof (*rule classical*)
assume $\neg\ 0 \leq a$
then have $a < 0$ **by** *auto*
with *A* **have** $- a < 0$ **by** (*rule le-less-trans*)
then show *?thesis* **by** *auto*
qed
next
assume *A*: $0 \leq a$ **show** $- a \leq a$
proof (*rule order-trans*)
show $- a \leq 0$ **using** *A* **by** (*simp add*: *minus-le-iff*)
next
show $0 \leq a$ **using** *A* **.**
qed
qed

lemma *neg-less-nonneg* [*simp*]:
$- a < a \longleftrightarrow 0 < a$
proof
assume *A*: $- a < a$ **show** $0 < a$
proof (*rule classical*)
assume $\neg\ 0 < a$
then have $a \leq 0$ **by** *auto*
with *A* **have** $- a < 0$ **by** (*rule less-le-trans*)
then show *?thesis* **by** *auto*
qed
next
assume *A*: $0 < a$ **show** $- a < a$
proof (*rule less-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* [*simp*]:
$a \leq - a \longleftrightarrow a \leq 0$
proof
assume *A*: $a \leq - a$ **show** $a \leq 0$
proof (*rule classical*)
assume $\neg\ a \leq 0$
then have $0 < a$ **by** *auto*

```
    then have 0 < − a using A by (rule less-le-trans)
    then show ?thesis by auto
  qed
next
  assume A: a ≤ 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 [simp]:
  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 [simp]:
  − a = a ⟷ a = 0
  by (auto dest: sym)

lemma double-zero [simp]:
  a + a = 0 ⟷ a = 0
proof
  assume assm: a + a = 0
  then have a: − a = a by (rule minus-unique)
  then show a = 0 by (simp only: neg-equal-zero)
qed simp

lemma double-zero-sym [simp]:
  0 = a + a ⟷ a = 0
  by (rule, drule sym) simp-all

lemma zero-less-double-add-iff-zero-less-single-add [simp]:
  0 < a + a ⟷ 0 < a
proof
  assume 0 < a + a
```

```
  then have 0 − a < a by (simp only: diff-less-eq)
  then have − a < a by simp
  then show 0 < a by (simp only: neg-less-nonneg)
next
  assume 0 < a
  with this have 0 + 0 < a + a
    by (rule add-strict-mono)
  then show 0 < a + a by simp
qed

lemma zero-le-double-add-iff-zero-le-single-add [simp]:
  0 ≤ a + a ⟷ 0 ≤ a
  by (auto simp add: le-less)

lemma double-add-less-zero-iff-single-add-less-zero [simp]:
  a + a < 0 ⟷ a < 0
proof −
  have ¬ a + a < 0 ⟷ ¬ a < 0
    by (simp add: not-less)
  then show ?thesis by simp
qed

lemma double-add-le-zero-iff-single-add-le-zero [simp]:
  a + a ≤ 0 ⟷ a ≤ 0
proof −
  have ¬ a + a ≤ 0 ⟷ ¬ a ≤ 0
    by (simp add: not-le)
  then show ?thesis by simp
qed

lemma le-minus-self-iff:
  a ≤ − a ⟷ a ≤ 0
proof −
  from add-le-cancel-left [of − a a + a 0]
  have a ≤ − a ⟷ a + a ≤ 0
    by (simp add: add-assoc [symmetric])
  thus ?thesis by simp
qed

lemma minus-le-self-iff:
  − a ≤ a ⟷ 0 ≤ a
proof −
  from add-le-cancel-left [of − a 0 a + a]
  have − a ≤ a ⟷ 0 ≤ a + a
    by (simp add: add-assoc [symmetric])
  thus ?thesis by simp
qed

lemma minus-max-eq-min:
```

$- \ max\ x\ y = min\ (-x)\ (-y)$
by (*auto simp add*: *max-def min-def*)

lemma *minus-min-eq-max*:
$- \ min\ x\ y = max\ (-x)\ (-y)$
by (*auto simp add*: *max-def min-def*)

end

context *ordered-comm-monoid-add*
begin

lemma *add-increasing*:
$0 \leq a \Longrightarrow b \leq c \Longrightarrow b \leq a + c$
by (*insert add-mono* [*of 0 a b c*], *simp*)

lemma *add-increasing2*:
$0 \leq c \Longrightarrow b \leq a \Longrightarrow b \leq a + c$
by (*simp add*: *add-increasing add-commute* [*of a*])

lemma *add-strict-increasing*:
$0 < a \Longrightarrow b \leq c \Longrightarrow b < a + c$
by (*insert add-less-le-mono* [*of 0 a b c*], *simp*)

lemma *add-strict-increasing2*:
$0 \leq a \Longrightarrow b < c \Longrightarrow b < a + c$
by (*insert add-le-less-mono* [*of 0 a b c*], *simp*)

end

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

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

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

end

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

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*: $sgn\ x = (if\ x = 0\ then\ 0\ else\ if\ 0 < x\ then\ 1\ else\ -1)$
begin

lemma *sgn0* [*simp*]: $sgn\ 0 = 0$
by (*simp add*:*sgn-if*)

end

class *ordered-ab-group-add-abs* = *ordered-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* $-|a|$ *0* $|a|$])

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* $-a$] **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*, *no-atp*]: $0 = |a| \longleftrightarrow a = 0$
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* $- a$], *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*: $|a| - |b| \leq |a - b|$
proof −
  have $|a| = |b + (a - b)|$
    by (*simp add*: *algebra-simps add-diff-cancel*)
  then have $|a| \leq |b| + |a - b|$
    by (*simp add*: *abs-triangle-ineq*)
  then show *?thesis*
    by (*simp add*: *algebra-simps*)
qed

lemma *abs-triangle-ineq2-sym*: $|a| - |b| \leq |b - a|$
  by (*simp only*: *abs-minus-commute* [*of b*] *abs-triangle-ineq2*)

lemma *abs-triangle-ineq3*: $||a| - |b|| \leq |a - b|$
  by (*simp add*: *abs-le-iff abs-triangle-ineq2 abs-triangle-ineq2-sym*)

lemma *abs-triangle-ineq4*: $|a - b| \leq |a| + |b|$
proof −
  have $|a - b| = |a + - b|$ **by** (*subst diff-minus*, *rule refl*)
  also have $... \leq |a| + |- b|$ **by** (*rule abs-triangle-ineq*)
  finally show *?thesis* **by** *simp*
qed

lemma *abs-diff-triangle-ineq*: $|a + b - (c + d)| \leq |a - c| + |b - d|$
proof −
  have $|a + b - (c+d)| = |(a-c) + (b-d)|$ **by** (*simp add*: *diff-minus add-ac*)
  also have $... \leq |a-c| + |b-d|$ **by** (*rule abs-triangle-ineq*)
  finally show *?thesis* **.**
qed

lemma *abs-add-abs* [*simp*]:
  $||a| + |b|| = |a| + |b|$ (**is** *?L* = *?R*)
proof (*rule antisym*)
  show *?L* $\geq$ *?R* **by**(*rule abs-ge-self*)
next

have *?L* $\leq ||a|| + ||b||$ **by**(*rule abs-triangle-ineq*)
also have $\ldots = ?R$ **by** *simp*
finally show *?L* $\leq$ *?R* **.**
qed

end

Needed for abelian cancellation simprocs:

lemma *add-cancel-21*: $((x::'a::ab\text{-}group\text{-}add) + (y + z) = y + u) = (x + z = u)$
apply (*subst add-left-commute*)
apply (*subst add-left-cancel*)
apply *simp*
done

lemma *add-cancel-end*: $(x + (y + z) = y) = (x = -\ (z::'a::ab\text{-}group\text{-}add))$
apply (*subst add-cancel-21*[*of - - - 0*, *simplified*])
apply (*simp add*: *add-right-cancel*[*symmetric*, *of x* $-z$ *z*, *simplified*])
done

lemma *less-eqI*: $(x::'a::ordered\text{-}ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (x < y) = (x' < y')$
by (*simp add*: *less-iff-diff-less-0*[*of x y*] *less-iff-diff-less-0*[*of x' y'*])

lemma *le-eqI*: $(x::'a::ordered\text{-}ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (y <= x) = (y' <= x')$
apply (*simp add*: *le-iff-diff-le-0*[*of y x*] *le-iff-diff-le-0*[*of y' x'*])
apply (*simp add*: *neg-le-iff-le*[*symmetric*, *of* $y-x$ *0*] *neg-le-iff-le*[*symmetric*, *of* $y'-x'$ *0*])
done

lemma *eq-eqI*: $(x::'a::ab\text{-}group\text{-}add) - y = x' - y' \Longrightarrow (x = y) = (x' = y')$
by (*simp only*: *eq-iff-diff-eq-0*[*of x y*] *eq-iff-diff-eq-0*[*of x' y'*])

lemma *diff-def*: $(x::'a::ab\text{-}group\text{-}add) - y == x + (-y)$
by (*simp add*: *diff-minus*)

lemma *le-add-right-mono*:
assumes
$a <= b + (c::'a::ordered\text{-}ab\text{-}group\text{-}add)$
$c <= d$
shows $a <= b + d$
apply (*rule-tac order-trans*[**where** $y = b+c$])
apply (*simp-all add*: *prems*)
done

4.8 Tools setup

lemma *add-mono-thms-linordered-semiring* [*no-atp*]:
fixes *i j k* :: $'a::ordered\text{-}ab\text{-}semigroup\text{-}add$

shows $i \leq j \wedge k \leq l \Longrightarrow i + k \leq j + l$
and $i = j \wedge k \leq l \Longrightarrow i + k \leq j + l$
and $i \leq j \wedge k = l \Longrightarrow i + k \leq j + l$
and $i = j \wedge k = l \Longrightarrow i + k = j + l$
by (*rule add-mono*, *clarify*+)+

lemma *add-mono-thms-linordered-field* [*no-atp*]:
fixes $i\ j\ k :: 'a$::*ordered-cancel-ab-semigroup-add*
shows $i < j \wedge k = l \Longrightarrow i + k < j + l$
and $i = j \wedge k < l \Longrightarrow i + k < j + l$
and $i < j \wedge k \leq l \Longrightarrow i + k < j + l$
and $i \leq j \wedge k < l \Longrightarrow i + k < j + l$
and $i < j \wedge k < l \Longrightarrow i + k < j + l$
by (*auto intro*: *add-strict-right-mono add-strict-left-mono*
add-less-le-mono add-le-less-mono add-strict-mono)

Simplification of $x - y < (0::'a)$, etc.

lemmas *diff-less-0-iff-less* [*simp*, *no-atp*] = *less-iff-diff-less-0* [*symmetric*]
lemmas *diff-le-0-iff-le* [*simp*, *no-atp*] = *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 Groups.zero}, @{const-name Groups.plus},
@{const-name Groups.uminus}, @{const-name Groups.minus}]
| agrp-ord - = ~1;

fun termless-agrp (a, b) = (Term-Ord.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 Groups.plus},-) $ - $ -) $ -) =
SOME ac1
| solve-add-ac thy - (- $ x $ (Const (@{const-name Groups.plus},-) $ y $ z)) =
if termless-agrp (y, x) then SOME ac3 else NONE
| solve-add-ac thy - (- $ x $ y) =
if termless-agrp (y, x) then SOME ac2 else NONE
| solve-add-ac thy - - = NONE
in
val add-ac-proc = Simplifier.simproc @{theory}
add-ac-proc [x + y::'a::ab-semigroup-add] solve-add-ac;
end;

val 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 @{*const-name op* =} *HOLogic.boolT o HOLogic.dest-Trueprop*
o concl-of;

);
⟩⟩

ML ⟨⟨
Addsimprocs [*ab-group-add-cancel.sum-conv*, *ab-group-add-cancel.rel-conv*];
⟩⟩

code-modulename *SML*
Groups Arith

code-modulename *OCaml*
Groups Arith

code-modulename *Haskell*
Groups Arith

end

5 Lattices: Abstract lattices

theory *Lattices*
imports *Orderings Groups*
begin

5.1 Abstract semilattice

This locales provide a basic structure for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may

occur due to interpretation.

locale *semilattice* = *abel-semigroup* +
  assumes *idem* [*simp*]: *f a a* = *a*
begin

lemma *left-idem* [*simp*]:
  f a (*f a b*) = *f a b*
  by (*simp add*: *assoc* [*symmetric*])

end

5.2 Idempotent semigroup

class *ab-semigroup-idem-mult* = *ab-semigroup-mult* +
  assumes *mult-idem*: *x* ∗ *x* = *x*

sublocale *ab-semigroup-idem-mult* < *times*!: *semilattice times* **proof**
qed (*fact mult-idem*)

context *ab-semigroup-idem-mult*
begin

lemmas *mult-left-idem* = *times.left-idem*

end

5.3 Concrete lattices

notation
  less-eq (**infix** ⊑ *50*) **and**
  less (**infix** ⊏ *50*) **and**
  top (⊤) **and**
  bot (⊥)

class *semilattice-inf* = *order* +
  fixes *inf* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** ⊓ *70*)
  assumes *inf-le1* [*simp*]: *x* ⊓ *y* ⊑ *x*
  and *inf-le2* [*simp*]: *x* ⊓ *y* ⊑ *y*
  and *inf-greatest*: *x* ⊑ *y* ⟹ *x* ⊑ *z* ⟹ *x* ⊑ *y* ⊓ *z*

class *semilattice-sup* = *order* +
  fixes *sup* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** ⊔ *65*)
  assumes *sup-ge1* [*simp*]: *x* ⊑ *x* ⊔ *y*
  and *sup-ge2* [*simp*]: *y* ⊑ *x* ⊔ *y*
  and *sup-least*: *y* ⊑ *x* ⟹ *z* ⊑ *x* ⟹ *y* ⊔ *z* ⊑ *x*
begin

Dual lattice

lemma *dual-semilattice*:

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

end

class *lattice* = *semilattice-inf* + *semilattice-sup*

5.3.1 Intro and elim rules

context *semilattice-inf*
begin

lemma *le-infI1*:
$a \sqsubseteq x \Longrightarrow a \sqcap b \sqsubseteq x$
by (*rule order-trans*) *auto*

lemma *le-infI2*:
$b \sqsubseteq x \Longrightarrow a \sqcap b \sqsubseteq x$
by (*rule order-trans*) *auto*

lemma *le-infI*: $x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcap b$
by (*rule inf-greatest*)

lemma *le-infE*: $x \sqsubseteq a \sqcap b \Longrightarrow (x \sqsubseteq a \Longrightarrow x \sqsubseteq b \Longrightarrow P) \Longrightarrow P$
by (*blast intro*: *order-trans inf-le1 inf-le2*)

lemma *le-inf-iff* [*simp*]:
$x \sqsubseteq y \sqcap z \longleftrightarrow x \sqsubseteq y \wedge x \sqsubseteq z$
by (*blast intro*: *le-infI elim*: *le-infE*)

lemma *le-iff-inf*:
$x \sqsubseteq y \longleftrightarrow x \sqcap y = x$
by (*auto intro*: *le-infI1 antisym dest*: *eq-iff* [*THEN iffD1*])

lemma *inf-mono*: $a \sqsubseteq c \Longrightarrow b \leq d \Longrightarrow a \sqcap b \sqsubseteq c \sqcap d$
by (*fast intro*: *inf-greatest le-infI1 le-infI2*)

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

end

context *semilattice-sup*
begin

lemma *le-supI1*:

$x \sqsubseteq a \Longrightarrow x \sqsubseteq a \sqcup b$
by (*rule order-trans*) *auto*

lemma *le-supI2*:
$x \sqsubseteq b \Longrightarrow x \sqsubseteq a \sqcup b$
by (*rule order-trans*) *auto*

lemma *le-supI*:
$a \sqsubseteq x \Longrightarrow b \sqsubseteq x \Longrightarrow a \sqcup b \sqsubseteq x$
by (*rule sup-least*)

lemma *le-supE*:
$a \sqcup b \sqsubseteq x \Longrightarrow (a \sqsubseteq x \Longrightarrow b \sqsubseteq x \Longrightarrow P) \Longrightarrow P$
by (*blast intro*: *order-trans sup-ge1 sup-ge2*)

lemma *le-sup-iff* [*simp*]:
$x \sqcup y \sqsubseteq z \longleftrightarrow x \sqsubseteq z \wedge y \sqsubseteq z$
by (*blast intro*: *le-supI elim*: *le-supE*)

lemma *le-iff-sup*:
$x \sqsubseteq y \longleftrightarrow x \sqcup y = y$
by (*auto intro*: *le-supI2 antisym dest*: *eq-iff* [*THEN iffD1*])

lemma *sup-mono*: $a \sqsubseteq c \Longrightarrow b \leq d \Longrightarrow a \sqcup b \sqsubseteq c \sqcup d$
by (*fast intro*: *sup-least le-supI1 le-supI2*)

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

end

5.3.2 Equational laws

sublocale *semilattice-inf* $<$ *inf*!: *semilattice inf*
proof
fix *a b c*
show $(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$
by (*rule antisym*) (*auto intro*: *le-infI1 le-infI2*)
show $a \sqcap b = b \sqcap a$
by (*rule antisym*) *auto*
show $a \sqcap a = a$
by (*rule antisym*) *auto*
qed

context *semilattice-inf*
begin

lemma *inf-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
by (*fact inf.assoc*)

lemma *inf-commute*: $(x \sqcap y) = (y \sqcap x)$
by (*fact inf.commute*)

lemma *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$
by (*fact inf.left-commute*)

lemma *inf-idem*: $x \sqcap x = x$
by (*fact inf.idem*)

lemma *inf-left-idem*: $x \sqcap (x \sqcap y) = x \sqcap y$
by (*fact inf.left-idem*)

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

lemma *inf-absorb2*: $y \sqsubseteq x \Longrightarrow x \sqcap y = y$
by (*rule antisym*) *auto*

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

end

sublocale *semilattice-sup* < *sup*!: *semilattice sup*
proof
fix *a b c*
show $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
by (*rule antisym*) (*auto intro*: *le-supI1 le-supI2*)
show $a \sqcup b = b \sqcup a$
by (*rule antisym*) *auto*
show $a \sqcup a = a$
by (*rule antisym*) *auto*
qed

context *semilattice-sup*
begin

lemma *sup-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
by (*fact sup.assoc*)

lemma *sup-commute*: $(x \sqcup y) = (y \sqcup x)$
by (*fact sup.commute*)

lemma *sup-left-commute*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
by (*fact sup.left-commute*)

lemma *sup-idem*: $x \sqcup x = x$

by (*fact sup.idem*)

lemma *sup-left-idem*: $x \sqcup (x \sqcup y) = x \sqcup y$
by (*fact sup.left-idem*)

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

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

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

end

context *lattice*
begin

lemma *dual-lattice*:
class.lattice (*op* $\geq$) (*op* $>$) *sup inf*
by (*rule class.lattice.intro*, *rule dual-semilattice*, *rule class.semilattice-sup.intro*, *rule dual-order*)
(*unfold-locales*, *auto*)

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 *inf-sup-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 (*auto intro*: *le-infI1 le-infI2 le-supI1 le-supI2*)

lemma *distrib-inf-le*: $(x \sqcap y) \sqcup (x \sqcap z) \sqsubseteq x \sqcap (y \sqcup z)$
by (*auto intro*: *le-infI1 le-infI2 le-supI1 le-supI2*)

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

end

5.3.3 Strict order

context *semilattice-inf*
begin

lemma *less-infI1*:
$a \sqsubset x \Longrightarrow a \sqcap b \sqsubset x$
by (*auto simp add*: *less-le inf-absorb1 intro*: *le-infI1*)

lemma *less-infI2*:
$b \sqsubset x \Longrightarrow a \sqcap b \sqsubset x$
by (*auto simp add*: *less-le inf-absorb2 intro*: *le-infI2*)

end

context *semilattice-sup*
begin

lemma *less-supI1*:
$x \sqsubset a \Longrightarrow x \sqsubset a \sqcup b$
proof −
interpret *dual*: *semilattice-inf op* $\geq$ *op* $>$ *sup*
by (*fact dual-semilattice*)
assume $x \sqsubset a$
then show $x \sqsubset a \sqcup b$
by (*fact dual.less-infI1*)
qed

lemma *less-supI2*:
$x \sqsubset b \Longrightarrow x \sqsubset a \sqcup b$

proof −
interpret *dual*: *semilattice-inf op* ≥ *op* > *sup*
by (*fact dual-semilattice*)
assume $x \sqsubset b$
then show $x \sqsubset a \sqcup b$
by (*fact dual.less-infI2*)
qed

end

5.4 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*: *inf-sup-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*: *inf-sup-aci inf-sup-distrib1*)

lemma *dual-distrib-lattice*:
class.distrib-lattice (*op* ≥) (*op* >) *sup inf*
by (*rule class.distrib-lattice.intro*, *rule dual-lattice*)
(*unfold-locales*, *fact inf-sup-distrib1*)

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

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

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

end

5.5 Bounded lattices and boolean algebras

class *bounded-lattice-bot* = *lattice* + *bot*
begin

lemma *inf-bot-left* [*simp*]:
$\bot \sqcap x = \bot$
by (*rule inf-absorb1*) *simp*

lemma *inf-bot-right* [*simp*]:
$x \sqcap \bot = \bot$
by (*rule inf-absorb2*) *simp*

lemma *sup-bot-left* [*simp*]:
$\bot \sqcup x = x$
by (*rule sup-absorb2*) *simp*

lemma *sup-bot-right* [*simp*]:
$x \sqcup \bot = x$
by (*rule sup-absorb1*) *simp*

lemma *sup-eq-bot-iff* [*simp*]:
$x \sqcup y = \bot \longleftrightarrow x = \bot \wedge y = \bot$
by (*simp add*: *eq-iff*)

end

class *bounded-lattice-top* = *lattice* + *top*
begin

lemma *sup-top-left* [*simp*]:
$\top \sqcup x = \top$
by (*rule sup-absorb1*) *simp*

lemma *sup-top-right* [*simp*]:
$x \sqcup \top = \top$
by (*rule sup-absorb2*) *simp*

lemma *inf-top-left* [*simp*]:
$\top \sqcap x = x$
by (*rule inf-absorb2*) *simp*

lemma *inf-top-right* [*simp*]:
$x \sqcap \top = x$
by (*rule inf-absorb1*) *simp*

lemma *inf-eq-top-iff* [*simp*]:
$x \sqcap y = \top \longleftrightarrow x = \top \wedge y = \top$
by (*simp add*: *eq-iff*)

end

class *bounded-lattice* = *bounded-lattice-bot* + *bounded-lattice-top*

begin

lemma *dual-bounded-lattice*:
class.bounded-lattice (*op* ≥) (*op* >) (*op* ⊔) (*op* ⊓) ⊤ ⊥
by *unfold-locales* (*auto simp add*: *less-le-not-le*)

end

class *boolean-algebra* = *distrib-lattice* + *bounded-lattice* + *minus* + *uminus* +
assumes *inf-compl-bot*: *x* ⊓ − *x* = ⊥
and *sup-compl-top*: *x* ⊔ − *x* = ⊤
assumes *diff-eq*: *x* − *y* = *x* ⊓ − *y*
begin

lemma *dual-boolean-algebra*:
class.boolean-algebra (λ*x y*. *x* ⊔ − *y*) *uminus* (*op* ≥) (*op* >) (*op* ⊔) (*op* ⊓) ⊤ ⊥
by (*rule class.boolean-algebra.intro*, *rule dual-bounded-lattice*, *rule dual-distrib-lattice*)
(*unfold-locales*, *auto simp add*: *inf-compl-bot sup-compl-top diff-eq*)

lemma *compl-inf-bot*:
− *x* ⊓ *x* = ⊥
by (*simp add*: *inf-commute inf-compl-bot*)

lemma *compl-sup-top*:
− *x* ⊔ *x* = ⊤
by (*simp add*: *sup-commute sup-compl-top*)

lemma *compl-unique*:
assumes *x* ⊓ *y* = ⊥
and *x* ⊔ *y* = ⊤
shows − *x* = *y*
proof −
have (*x* ⊓ − *x*) ⊔ (− *x* ⊓ *y*) = (*x* ⊓ *y*) ⊔ (− *x* ⊓ *y*)
using *inf-compl-bot assms*(*1*) **by** *simp*
then have (− *x* ⊓ *x*) ⊔ (− *x* ⊓ *y*) = (*y* ⊓ *x*) ⊔ (*y* ⊓ − *x*)
by (*simp add*: *inf-commute*)
then have − *x* ⊓ (*x* ⊔ *y*) = *y* ⊓ (*x* ⊔ − *x*)
by (*simp add*: *inf-sup-distrib1*)
then have − *x* ⊓ ⊤ = *y* ⊓ ⊤
using *sup-compl-top assms*(*2*) **by** *simp*
then show − *x* = *y* **by** *simp*
qed

lemma *double-compl* [*simp*]:
− (− *x*) = *x*
using *compl-inf-bot compl-sup-top* **by** (*rule compl-unique*)

lemma *compl-eq-compl-iff* [*simp*]:
− *x* = − *y* ⟷ *x* = *y*

proof
assume $- x = - y$
then have $- (- x) = - (- y)$ **by** (*rule arg-cong*)
then show $x = y$ **by** *simp*
next
assume $x = y$
then show $- x = - y$ **by** *simp*
qed

lemma *compl-bot-eq* [*simp*]:
$- \bot = \top$
proof −
from *sup-compl-top* **have** $\bot \sqcup - \bot = \top$ **.**
then show *?thesis* **by** *simp*
qed

lemma *compl-top-eq* [*simp*]:
$- \top = \bot$
proof −
from *inf-compl-bot* **have** $\top \sqcap - \top = \bot$ **.**
then show *?thesis* **by** *simp*
qed

lemma *compl-inf* [*simp*]:
$- (x \sqcap y) = - x \sqcup - y$
proof (*rule compl-unique*)
have $(x \sqcap y) \sqcap (- x \sqcup - y) = (y \sqcap (x \sqcap - x)) \sqcup (x \sqcap (y \sqcap - y))$
by (*simp only*: *inf-sup-distrib inf-aci*)
then show $(x \sqcap y) \sqcap (- x \sqcup - y) = \bot$
by (*simp add*: *inf-compl-bot*)
next
have $(x \sqcap y) \sqcup (- x \sqcup - y) = (- y \sqcup (x \sqcup - x)) \sqcap (- x \sqcup (y \sqcup - y))$
by (*simp only*: *sup-inf-distrib sup-aci*)
then show $(x \sqcap y) \sqcup (- x \sqcup - y) = \top$
by (*simp add*: *sup-compl-top*)
qed

lemma *compl-sup* [*simp*]:
$- (x \sqcup y) = - x \sqcap - y$
proof −
interpret *boolean-algebra* $\lambda x\ y.\ x \sqcup - y$ *uminus op* $\geq$ *op* $>$ *op* $\sqcup$ *op* $\sqcap$ $\top$ $\bot$
by (*rule dual-boolean-algebra*)
then show *?thesis* **by** *simp*
qed

lemma *compl-mono*:
$x \sqsubseteq y \Longrightarrow - y \sqsubseteq - x$
proof −
assume $x \sqsubseteq y$

then have $x \sqcup y = y$ **by** (*simp only*: *le-iff-sup*)
then have $- (x \sqcup y) = - y$ **by** *simp*
then have $- x \sqcap - y = - y$ **by** *simp*
then have $- y \sqcap - x = - y$ **by** (*simp only*: *inf-commute*)
then show $- y \sqsubseteq - x$ **by** (*simp only*: *le-iff-inf*)
qed

lemma *compl-le-compl-iff*:
$- x \leq - y \longleftrightarrow y \leq x$
by (*auto dest*: *compl-mono*)

end

5.6 Uniqueness of inf and sup

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

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

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

sublocale *linorder* $<$ *min-max*!: *distrib-lattice less-eq less min max*
proof
fix *x y z*
show *max x* (*min y z*) = *min* (*max x y*) (*max x z*)
by (*auto simp add*: *min-def max-def*)
qed (*auto simp add*: *min-def max-def not-le less-imp-le*)

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

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

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

lemmas *min-ac* = *min-max.inf-assoc min-max.inf-commute*
min-max.inf.left-commute

lemmas *max-ac* = *min-max.sup-assoc min-max.sup-commute*
min-max.sup.left-commute

5.8 Bool as lattice

instantiation *bool* :: *boolean-algebra*
begin

definition
bool-Compl-def: *uminus* = *Not*

definition
bool-diff-def: *A* − *B* ⟷ *A* ∧ ¬ *B*

definition
inf-bool-eq: *P* ⊓ *Q* ⟷ *P* ∧ *Q*

definition
sup-bool-eq: *P* ⊔ *Q* ⟷ *P* ∨ *Q*

instance proof
qed (*simp-all add*: *inf-bool-eq sup-bool-eq le-bool-def*
bot-bool-eq top-bool-eq bool-Compl-def bool-diff-def, *auto*)

end

lemma *sup-boolI1*:
P ⟹ *P* ⊔ *Q*
by (*simp add*: *sup-bool-eq*)

lemma *sup-boolI2*:
Q ⟹ *P* ⊔ *Q*
by (*simp add*: *sup-bool-eq*)

lemma *sup-boolE*:
P ⊔ *Q* ⟹ (*P* ⟹ *R*) ⟹ (*Q* ⟹ *R*) ⟹ *R*
by (*auto simp add*: *sup-bool-eq*)

5.9 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 proof
qed (*simp-all add*: *le-fun-def inf-fun-eq sup-fun-eq*)

end

instance *fun* :: (*type*, *distrib-lattice*) *distrib-lattice*
proof
qed (*simp-all add*: *inf-fun-eq sup-fun-eq sup-inf-distrib1*)

instance *fun* :: (*type*, *bounded-lattice*) *bounded-lattice* **..**

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

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

instance ..

end

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

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

instance ..

end

instance *fun* :: (*type*, *boolean-algebra*) *boolean-algebra*
proof
qed (*simp-all add*: *inf-fun-eq sup-fun-eq bot-fun-eq top-fun-eq fun-Compl-def fun-diff-def*
inf-compl-bot sup-compl-top diff-eq)

no-notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**

less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
top ($\top$) **and**
bot ($\bot$)

end

6 Set: Set theory for higher-order logic

theory *Set*
imports *Lattices*
begin

6.1 Sets as predicates

global

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

consts
Collect :: *('a => bool) => 'a set* — comprehension
op : :: *'a => 'a set => bool* — membership

local

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

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

abbreviation
not-mem x A == $\sim$ (*x : A*) — non-membership

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

notation (*xsymbols*)
op : (*op* $\in$) **and**
op : ((*-/* $\in$ *-*) [*50*, *51*] *50*) **and**
not-mem (*op* $\notin$) **and**
not-mem ((*-/* $\notin$ *-*) [*50*, *51*] *50*)

notation (*HTML* **output**)

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

Set comprehensions

syntax
-*Coll* :: *pttrn* => *bool* => *'a set* ((*1*{-./ -}))
translations
{*x*. *P*} == *CONST Collect* (%*x*. *P*)

syntax
-*Collect* :: *idt* => *'a set* => *bool* => *'a set* ((*1*{- :/ -./ -}))
syntax (*xsymbols*)
-*Collect* :: *idt* => *'a set* => *bool* => *'a set* ((*1*{- ∈/ -./ -}))
translations
{*x*:*A*. *P*} => {*x*. *x*:*A* & *P*}

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

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

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

Simproc for pulling *x*=*t* in {*x*. ... & *x*=*t* & ...} to the front (and similarly for *t*=*x*):

setup ⟪
let
val Coll-perm-tac = *rtac* @{*thm Collect-cong*} *1 THEN rtac* @{*thm iffI*} *1 THEN*
ALLGOALS(*EVERY'*[*REPEAT-DETERM o* (*etac* @{*thm conjE*}),
DEPTH-SOLVE-1 o (*ares-tac* [@{*thm conjI*}])])
val defColl-regroup = *Simplifier.simproc* @{*theory*}
defined Collect [{*x*. *P x* & *Q x*}]
(*Quantifier1.rearrange-Coll Coll-perm-tac*)
in
Simplifier.map-simpset (*fn ss* => *ss addsimprocs* [*defColl-regroup*])
end
⟫

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

Set enumerations

abbreviation *empty* :: *'a set* ({}) **where**
  $\{\} \equiv bot$

definition *insert* :: *'a* ⇒ *'a set* ⇒ *'a set* **where**
  insert-compr: $insert\ a\ B = \{x.\ x = a \vee x \in B\}$

syntax
  -Finset :: *args* => *'a set* ({(-)})
translations
  $\{x, xs\} == CONST\ insert\ x\ \{xs\}$
  $\{x\} == CONST\ insert\ x\ \{\}$

6.2 Subsets and bounded quantifiers

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

abbreviation
  subset-eq :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
  $subset\text{-}eq \equiv less\text{-}eq$

notation (**output**)
  $subset\ \ (op <)$ **and**
  $subset\ \ ((-/ < -)\ [50, 51]\ 50)$ **and**
  $subset\text{-}eq\ \ (op <=)$ **and**
  $subset\text{-}eq\ \ ((-/ <= -)\ [50, 51]\ 50)$

notation (*xsymbols*)
  $subset\ \ (op \subset)$ **and**
  $subset\ \ ((-/ \subset -)\ [50, 51]\ 50)$ **and**
  $subset\text{-}eq\ \ (op \subseteq)$ **and**
  $subset\text{-}eq\ \ ((-/ \subseteq -)\ [50, 51]\ 50)$

notation (*HTML* **output**)
  $subset\ \ (op \subset)$ **and**
  $subset\ \ ((-/ \subset -)\ [50, 51]\ 50)$ **and**
  $subset\text{-}eq\ \ (op \subseteq)$ **and**
  $subset\text{-}eq\ \ ((-/ \subseteq -)\ [50, 51]\ 50)$

abbreviation (*input*)
  supset :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
  $supset \equiv greater$

abbreviation (*input*)
  supset-eq :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
  $supset\text{-}eq \equiv greater\text{-}eq$

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

global

consts
Ball :: *'a set => ('a => bool) => bool* — bounded universal quantifiers

Bex :: *'a set => ('a => bool) => bool* — bounded existential quantifiers

local

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

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

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*}; (∗ *FIXME 'a => bool* (!*?*!) ∗)
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*), @{*syntax-const -setlessAll*}),
((*All-binder*, *impl*, *sbset-eq*), @{*syntax-const -setleAll*}),

```
    ((Ex-binder, conj, sbset), @{syntax-const -setlessEx}),
    ((Ex-binder, conj, sbset-eq), @{syntax-const -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 (@{syntax-const -bound}, -) $ Free (v, Type (T, -)),
            Const (c, -) $
              (Const (d, -) $ (Const (@{syntax-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* $bvs(e)$.

syntax

```
  -Setcompr :: 'a => idts => bool => 'a set    ((1{- |/-./ -}))
```

parse-translation ⟪

```
  let
    val ex-tr = snd (mk-binder-tr (EX , @{const-syntax Ex}));

    fun nvars (Const (@{syntax-const -idts}, -) $ - $ idts) = nvars idts + 1
      | nvars - = 1;

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

  in [(@{syntax-const -Setcompr}, setcompr-tr)] end;
⟫
```

print-translation ⟪

```
 [Syntax.preserve-binder-abs2-tr' @{const-syntax Ball} @{syntax-const -Ball},
```

```
  Syntax.preserve-binder-abs2-tr' @{const-syntax Bex} @{syntax-const -Bex}]
⟩⟩ — to avoid eta-contraction of body

print-translation ⟨⟨
let
  val ex-tr' = snd (mk-binder-tr' (@{const-syntax Ex}, DUMMY));

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

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

setup ⟨⟨
let
  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;
  val defBEX-regroup = Simplifier.simproc @{theory}
    defined BEX [EX x:A. P x & Q x] rearrange-bex;
```

```
  val defBALL-regroup = Simplifier.simproc @{theory}
    defined BALL [ALL x:A. P x --> Q x] rearrange-ball;
in
 Simplifier.map-simpset (fn ss => ss addsimprocs [defBALL-regroup, defBEX-regroup])
end
```

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

lemmas *strip* = *impI allI ballI*

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

Gives better instantiation for bound:

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

ML ⟪
structure Simpdata =
struct

open Simpdata;

val mksimps-pairs = [(@{const-name Ball}, @{thms bspec})] @ mksimps-pairs;

end;

open Simpdata;
⟫

declaration ⟪ fn - =>
  Simplifier.map-ss (fn ss => ss setmksimps (mksimps mksimps-pairs))
⟫
```

lemma *ballE* [*elim*]: *ALL x*:*A*. *P x* ==> (*P x* ==> *Q*) ==> (*x* ~: *A* ==> *Q*)
==> *Q*
by (*unfold Ball-def*) *blast*

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

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

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

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

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

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

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

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

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

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

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

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

Congruence rules

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

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

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

lemma *strong-bex-cong* [*cong*]:

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

6.3 Basic operations

6.3.1 Subsets

lemma *subsetI* [*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow x \in B) \Longrightarrow A \subseteq B$
unfolding *mem-def* **by** (*rule le-funI*, *rule le-boolI*)

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

lemma *subsetD* [*elim*, *intro?*]: $A \subseteq B$ ==> $c \in A$ ==> $c \in B$
unfolding *mem-def* **by** (*erule le-funE*, *erule le-boolE*)
— Rule in Modus Ponens style.

lemma *rev-subsetD* [*no-atp*,*intro?*]: $c \in A$ ==> $A \subseteq B$ ==> $c \in B$
— The same, with reversed premises for use with *erule* – cf *rev-mp*.
by (*rule subsetD*)

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

lemma *subsetCE* [*no-atp*,*elim*]: $A \subseteq B$ ==> ($c \notin A$ ==> P) ==> ($c \in B$ ==> P) ==> P
— Classical elimination rule.
unfolding *mem-def* **by** (*blast dest*: *le-funE le-boolE*)

lemma *subset-eq* [*no-atp*]: $A \leq B = (\forall x{\in}A.\ x \in B)$ **by** *blast*

lemma *contra-subsetD* [*no-atp*]: $A \subseteq B$ ==> $c \notin B$ ==> $c \notin A$
by *blast*

lemma *subset-refl* [*simp*]: $A \subseteq A$
by (*fact order-refl*)

lemma *subset-trans*: $A \subseteq B$ ==> $B \subseteq C$ ==> $A \subseteq C$
by (*fact order-trans*)

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

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

lemma *eq-mem-trans*: a=b ==> $b \in A$ ==> $a \in A$
by *simp*

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

6.3.2 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* ⊆ *B* ==> *B* ⊆ *A* ==> *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* ==> *A* ⊆ *B*
by *simp*

lemma *equalityD2*: *A* = *B* ==> *B* ⊆ *A*
by *simp*

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

lemma *equalityE*: *A* = *B* ==> (*A* ⊆ *B* ==> *B* ⊆ *A* ==> *P*) ==> *P*
by *simp*

lemma *equalityCE* [*elim*]:
A = *B* ==> (*c* ∈ *A* ==> *c* ∈ *B* ==> *P*) ==> (*c* ∉ *A* ==> *c* ∉ *B* ==> *P*)
==> *P*
by *blast*

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

lemma *eqelem-imp-iff*: *x* = *y* ==> (*x* : *A*) = (*y* : *A*)
by *simp*

6.3.3 The universal set – UNIV

abbreviation *UNIV* :: *'a set* **where**
UNIV ≡ *top*

lemma *UNIV-def*:
UNIV = {*x*. *True*}
by (*simp add*: *top-fun-eq top-bool-eq Collect-def*)

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*

6.3.4 The empty set

lemma *empty-def*:
$\{\} = \{x.\ False\}$
by (*simp add*: *bot-fun-eq bot-bool-eq Collect-def*)

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

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

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

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

lemma *equals0D*: $A = \{\} \Longrightarrow a \notin A$
— Use for reasoning about disjointness: *A Int 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*]: $UNIV \neq \{\}$
by (*blast elim*: *equalityE*)

6.3.5 The Powerset operator – Pow

definition $Pow :: 'a\ set \Rightarrow 'a\ set\ set$ **where**
Pow-def: $Pow\ A = \{B.\ B \leq A\}$

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

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

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

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

lemma *Pow-top*: $A \in Pow\ A$
by *simp*

6.3.6 Set complement

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

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

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

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

6.3.7 Binary union – Un

abbreviation *union* :: $'a\ set \Rightarrow 'a\ set \Rightarrow 'a\ set$ (**infixl** *Un* 65) **where**
op Un ≡ *sup*

notation (*xsymbols*)
union (**infixl** ∪ 65)

notation (*HTML* **output**)
union (**infixl** $\cup$ *65*)

lemma *Un-def*:
$A \cup B = \{x.\ x \in A \vee x \in B\}$
by (*simp add*: *sup-fun-eq sup-bool-eq Collect-def mem-def*)

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*

lemma *insert-def*: $insert\ a\ B = \{x.\ x = a\} \cup B$
by (*simp add*: *Collect-def mem-def insert-compr Un-def*)

lemma *mono-Un*: $mono\ f \Longrightarrow f\ A \cup f\ B \subseteq f\ (A \cup B)$
by (*fact mono-sup*)

6.3.8 Binary intersection – Int

abbreviation *inter* :: *'a set* $\Rightarrow$ *'a set* $\Rightarrow$ *'a set* (**infixl** *Int 70*) **where**
op Int $\equiv$ *inf*

notation (*xsymbols*)
inter (**infixl** $\cap$ *70*)

notation (*HTML* **output**)
inter (**infixl** $\cap$ *70*)

lemma *Int-def*:
$A \cap B = \{x.\ x \in A \wedge x \in B\}$
by (*simp add*: *inf-fun-eq inf-bool-eq Collect-def mem-def*)

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

lemma *IntI* [*intro*!]: *c*:*A* ==> *c*:*B* ==> *c* : *A Int B*
by *simp*

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

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

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

lemma *mono-Int*: *mono f* ⟹ *f* (*A* ∩ *B*) ⊆ *f A* ∩ *f B*
by (*fact mono-inf*)

6.3.9 Set difference

lemma *Diff-iff* [*simp*]: (*c* : *A* − *B*) = (*c*:*A* & *c*~:*B*)
by (*simp add*: *mem-def fun-diff-def bool-diff-def*)

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

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

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

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

lemma *set-diff-eq*: *A* − *B* = {*x*. *x* : *A* & ~ *x* : *B*} **by** *blast*

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

6.3.10 Augmenting a set − *insert*

lemma *insert-iff* [*simp*]: (*a* : *insert b A*) = (*a* = *b* | *a*:*A*)
by (*unfold insert-def*) *blast*

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

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

lemma *insertE* [*elim*!]: *a* : *insert b A* ==> (*a* = *b* ==> *P*) ==> (*a*:*A* ==> *P*)
==> *P*

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

lemma *insertCI* [*intro*!]: (*a*~:*B* ==> *a* = *b*) ==> *a*: *insert b B*
— Classical introduction rule.
by *auto*

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

lemma *set-insert*:
assumes *x* ∈ *A*
obtains *B* **where** *A* = *insert x B* **and** *x* ∉ *B*
proof
from *assms* **show** *A* = *insert x* (*A* − {*x*}) **by** *blast*
next
show *x* ∉ *A* − {*x*} **by** *blast*
qed

lemma *insert-ident*: *x* ~: *A* ==> *x* ~: *B* ==> (*insert x A* = *insert x B*) = (*A* = *B*)
by *auto*

6.3.11 Singletons, using insert

lemma *singletonI* [*intro*!,*no-atp*]: *a* : {*a*}
— Redundant? But unlike *insertCI*, it proves the subgoal immediately!
by (*rule insertI1*)

lemma *singletonD* [*dest*!,*no-atp*]: *b* : {*a*} ==> *b* = *a*
by *blast*

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

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

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

lemma *singleton-insert-inj-eq* [*iff*,*no-atp*]:
({*b*} = *insert a A*) = (*a* = *b* & *A* ⊆ {*b*})
by *blast*

lemma *singleton-insert-inj-eq'* [*iff*,*no-atp*]:
(*insert a A* = {*b*}) = (*a* = *b* & *A* ⊆ {*b*})
by *blast*

lemma *subset-singletonD*: *A* ⊆ {*x*} ==> *A* = {} | *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*)

6.3.12 Image of a set under a function

Frequently b does not have the syntactic form of $f\ x$.

definition *image* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set$ (**infixr** ‘ *90*) **where**
image-def [*no-atp*]: $f\ `\ A = \{y.\ EX\ x{:}A.\ y = f(x)\}$

abbreviation
range :: $('a \Rightarrow 'b) \Rightarrow 'b\ set$ **where** — of function
$\mathit{range}\ f == f\ `\ UNIV$

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

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

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

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

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

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

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

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

lemma *image-subsetI*: $(!!x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f`A \subseteq B$
— Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.
by *blast*

Range of a function – just a translation for image!

lemma *range-eqI*: $b = f\ x \Longrightarrow b \in range\ f$
by *simp*

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

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

6.3.13 Some rules with *if*

Elimination of $\{x.\ \ldots\ \&\ x{=}t\ \&\ \ldots\}$.

lemma *Collect-conv-if*: $\{x.\ x{=}a\ \&\ P\ x\} = (if\ P\ a\ then\ \{a\}\ else\ \{\})$
by *auto*

lemma *Collect-conv-if2*: $\{x.\ a{=}x\ \&\ P\ x\} = (if\ P\ a\ then\ \{a\}\ else\ \{\})$
by *auto*

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*

6.4 Further operations and lemmas

6.4.1 The "proper subset" relation

lemma *psubsetI* [*intro*!,*no-atp*]: $A \subseteq B \Longrightarrow A \neq B \Longrightarrow A \subset B$
by (*unfold less-le*) *blast*

lemma *psubsetE* [*elim*!,*no-atp*]:
$[|A \subset B;\ [|A \subseteq B; \sim (B \subseteq A)|] \Longrightarrow R|] \Longrightarrow R$
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*:
$(!!x.\ x \in A \Longrightarrow P\ x) == \mathit{Trueprop}\ (\forall\, x \in A.\ P\ x)$
by (*simp only*: *Ball-def atomize-all atomize-imp*)

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

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

6.4.2 Derived rules involving subsets.

insert.

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

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

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

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
by (*fact sup-ge1*)

lemma *Un-upper2*: $B \subseteq A \cup B$
by (*fact sup-ge2*)

lemma *Un-least*: $A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow A \cup B \subseteq C$
by (*fact sup-least*)

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
by (*fact inf-le1*)

lemma *Int-lower2*: $A \cap B \subseteq B$
by (*fact inf-le2*)

lemma *Int-greatest*: $C \subseteq A \Longrightarrow C \subseteq B \Longrightarrow C \subseteq A \cap B$
by (*fact inf-greatest*)

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*

6.4.3 Equalities involving union, intersection, inclusion, etc.

{}.

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

by *auto*

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

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

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

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

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

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

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

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

insert.

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

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

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

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

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

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

lemma *insert-subset* [*simp*]: (*insert x A* $\subseteq$ *B*) = ($x \in B$ & $A \subseteq B$)

by *blast*

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

lemma *insert-Collect*: $insert\ a\ (Collect\ P) = \{u.\ u \neq a\ --> P\ u\}$
by *auto*

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

lemma *insert-disjoint* [*simp*,*no-atp*]:
$(insert\ a\ A \cap B = \{\}) = (a \notin B \wedge A \cap B = \{\})$
$(\{\} = insert\ a\ A \cap B) = (a \notin B \wedge \{\} = A \cap B)$
by *auto*

lemma *disjoint-insert* [*simp*,*no-atp*]:
$(B \cap insert\ a\ A = \{\}) = (a \notin B \wedge B \cap A = \{\})$
$(\{\} = A \cap insert\ b\ B) = (b \notin A \wedge \{\} = A \cap B)$
by *auto*

image.

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

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

lemma *image-constant*: $x \in A ==> (\lambda x.\ c)\ `\ A = \{c\}$
by *auto*

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

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

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

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

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

lemma *image-Collect* [*no-atp*]: 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* [*no-atp*]: $\{u.\ \exists x.\ u = f\ x\} = range\ f$
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 (*fact inf-idem*)

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

lemma *Int-commute*: $A \cap B = B \cap A$
by (*fact inf-commute*)

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

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

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 (*fact inf-absorb2*)

lemma *Int-absorb2*: $A \subseteq B \Longrightarrow A \cap B = A$
by (*fact inf-absorb1*)

lemma *Int-empty-left* [*simp*]: $\{\} \cap B = \{\}$
by (*fact inf-bot-left*)

lemma *Int-empty-right* [*simp*]: $A \cap \{\} = \{\}$
by (*fact inf-bot-right*)

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 (*fact inf-top-left*)

lemma *Int-UNIV-right* [*simp*]: $A \cap UNIV = A$
by (*fact inf-top-right*)

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by (*fact inf-sup-distrib1*)

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

lemma *Int-UNIV* [*simp*,*no-atp*]: $(A \cap B = UNIV) = (A = UNIV \ \&\ B = UNIV)$
by (*fact inf-eq-top-iff*)

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

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 (*fact sup-idem*)

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

lemma *Un-commute*: $A \cup B = B \cup A$
by (*fact sup-commute*)

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

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

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 (*fact sup-absorb2*)

lemma *Un-absorb2*: $B \subseteq A \Longrightarrow A \cup B = A$
by (*fact sup-absorb1*)

lemma *Un-empty-left* [*simp*]: $\{\} \cup B = B$
by (*fact sup-bot-left*)

lemma *Un-empty-right* [*simp*]: $A \cup \{\} = A$
by (*fact sup-bot-right*)

lemma *Un-UNIV-left* [*simp*]: $UNIV \cup B = UNIV$
by (*fact sup-top-left*)

lemma *Un-UNIV-right* [*simp*]: $A \cup UNIV = UNIV$
by (*fact sup-top-right*)

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-left-if0*[*simp*]:
$a \notin C \Longrightarrow (insert\ a\ B)\ Int\ C = B \cap C$
by *auto*

lemma *Int-insert-left-if1*[*simp*]:
$a \in C \Longrightarrow (insert\ a\ B)\ Int\ C = insert\ a\ (B\ Int\ C)$
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 *Int-insert-right-if0*[*simp*]:
$a \notin A \Longrightarrow A\ Int\ (insert\ a\ B) = A\ Int\ B$
by *auto*

lemma *Int-insert-right-if1*[*simp*]:
$a \in A \Longrightarrow A\ Int\ (insert\ a\ B) = insert\ a\ (A\ Int\ B)$
by *auto*

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
by (*fact sup-inf-distrib1*)

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

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 (*fact le-iff-sup*)

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

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

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 (*fact inf-compl-bot*)

lemma *Compl-disjoint2* [*simp*]: $-A \cap A = \{\}$
by (*fact compl-inf-bot*)

lemma *Compl-partition*: $A \cup -A = UNIV$
by (*fact sup-compl-top*)

lemma *Compl-partition2*: $-A \cup A = UNIV$
by (*fact compl-sup-top*)

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

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

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

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 (*fact compl-top-eq*)

lemma *Compl-empty-eq* [*simp*]: $-\{\} = UNIV$
by (*fact compl-bot-eq*)

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

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

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

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

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

Set difference.

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

lemma *Diff-eq-empty-iff* [*simp*,*no-atp*]: $(A - B = \{\}) = (A \subseteq B)$
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*,*no-atp*]: $x \notin A \Longrightarrow A - insert\ x\ B = A - B$
by *blast*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

by *blast*

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

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

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

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

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

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

Quantification over type *bool*.

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

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

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

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

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

6.4.4 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 *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 (*fact sup-mono*)

lemma *Int-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cap B \subseteq C \cap D$
by (*fact inf-mono*)

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

lemma *Compl-anti-mono*: $A \subseteq B \Longrightarrow -B \subseteq -A$
by (*fact compl-mono*)

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B \Longrightarrow x \in A \longrightarrow x \in B$

apply (*rule impI*)
apply (*erule subsetD*, *assumption*)
done

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

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

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

lemma *imp-refl*: *P* −−> *P* **..**

lemma *not-mono*: *Q* −−> *P* ==> ~ *P* −−> ~ *Q*
by *iprover*

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* ⊆ *Collect Q*
by *blast*

lemma *Int-Collect-mono*:
A ⊆ *B* ==> (!!*x*. *x* ∈ *A* ==> *P x* −−> *Q x*) ==> *A* ∩ *Collect P* ⊆ *B* ∩ *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*

6.4.5 Inverse image of a function

definition *vimage* :: ('*a* => '*b*) => '*b set* => '*a set* (**infixr** −‘ *90*) **where**
[*code del*]: *f* −‘ *B* == {*x*. *f x* : *B*}

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*

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

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

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

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

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

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

lemma *vimage-image-eq* [*no-atp*]: $f -` (f ` A) = \{y.\ EX\ x{:}A.\ f\ x = f\ y\}$
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 *vimage-const* [*simp*]: $((\lambda x.\ c) -` A) = (if\ c \in A\ then\ UNIV\ else\ \{\})$
by *auto*

lemma *vimage-if* [*simp*]: $((\lambda x.\ if\ x \in B\ then\ c\ else\ d) -` A) =$
$(if\ c \in A\ then\ (if\ d \in A\ then\ UNIV\ else\ B)$
$else\ if\ d \in A\ then\ -B\ else\ \{\})$
by (*auto simp add*: *vimage-def*)

lemma *vimage-inter-cong*:
$(\bigwedge w.\ w \in S \Longrightarrow f\ w = g\ w) \Longrightarrow f -` y \cap S = g -` y \cap S$
by *auto*

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*

6.4.6 Getting the Contents of a Singleton Set

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

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

6.4.7 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

6.5 Misc

Rudimentary code generation

lemma *insert-code* [*code*]: *insert y A x* $\longleftrightarrow$ *y = x* $\vee$ *A x*
by (*auto simp add*: *insert-compr Collect-def mem-def*)

lemma *vimage-code* [*code*]: (*f* $-`$ *A*) *x* = *A* (*f x*)
by (*simp add*: *vimage-def Collect-def mem-def*)

Misc theorem and ML bindings

lemmas *equalityI* = *subset-antisym*

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 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

7 Typedef: HOL type definitions

theory *Typedef*
imports *Set*
uses
(*Tools/typedef.ML*)
(*Tools/typecopy.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 ∈ A ==> P (Abs y)
  shows P x
proof −
  have Rep x ∈ A by (rule Rep)
  then have P (Abs (Rep x)) by (rule r)
  moreover have Abs (Rep x) = x by (rule Rep-inverse)
  ultimately show P x by simp
qed

lemma Rep-range: 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.ML setup Typedef.setup
use Tools/typecopy.ML setup Typecopy.setup
use Tools/typedef-codegen.ML setup TypedefCodegen.setup

end
```

8 Complete-Lattice: Complete lattices, with special focus on sets

```
theory Complete-Lattice
imports Set
begin
```

notation
less-eq (**infix** $\sqsubseteq$ *50*) **and**
less (**infix** $\sqsubset$ *50*) **and**
inf (**infixl** $\sqcap$ *70*) **and**
sup (**infixl** $\sqcup$ *65*) **and**
top ($\top$) **and**
bot ($\bot$)

8.1 Syntactic infimum and supremum operations

class *Inf* =
fixes *Inf* :: *'a set* $\Rightarrow$ *'a* ($\bigsqcap$- [*900*] *900*)

class *Sup* =
fixes *Sup* :: *'a set* $\Rightarrow$ *'a* ($\bigsqcup$- [*900*] *900*)

8.2 Abstract complete lattices

class *complete-lattice* = *bounded-lattice* + *Inf* + *Sup* +
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 *dual-complete-lattice*:
class.complete-lattice Sup Inf (*op* $\geq$) (*op* $>$) (*op* $\sqcup$) (*op* $\sqcap$) $\top$ $\bot$
by (*auto intro*!: *class.complete-lattice.intro dual-bounded-lattice*)
(*unfold-locales*, (*fact bot-least top-greatest*
Sup-upper Sup-least Inf-lower Inf-greatest)+)

lemma *Inf-Sup*: $\bigsqcap A = \bigsqcup \{b.\ \forall a \in A.\ b \sqsubseteq a\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Sup-Inf*: $\bigsqcup A = \bigsqcap \{b.\ \forall a \in A.\ a \sqsubseteq b\}$
by (*auto intro*: *antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Inf-empty*:
$\bigsqcap \{\} = \top$
by (*auto intro*: *antisym Inf-greatest*)

lemma *Sup-empty*:
$\bigsqcup \{\} = \bot$
by (*auto intro*: *antisym Sup-least*)

lemma *Inf-insert*: $\bigsqcap$ *insert a A* $= a \sqcap \bigsqcap A$
by (*auto intro*: *le-infI le-infI1 le-infI2 antisym Inf-greatest Inf-lower*)

lemma *Sup-insert*: $\bigsqcup$ *insert a A* $= a \sqcup \bigsqcup A$

by (*auto intro*: *le-supI le-supI1 le-supI2 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-binary*:
$\bigsqcap\{a, b\} = a \sqcap b$
by (*simp add*: *Inf-empty Inf-insert*)

lemma *Sup-binary*:
$\bigsqcup\{a, b\} = a \sqcup b$
by (*simp add*: *Sup-empty Sup-insert*)

lemma *Inf-UNIV*:
$\bigsqcap UNIV = bot$
by (*simp add*: *Sup-Inf Sup-empty* [*symmetric*])

lemma *Sup-UNIV*:
$\bigsqcup UNIV = top$
by (*simp add*: *Inf-Sup Inf-empty* [*symmetric*])

lemma *Sup-le-iff*: $Sup\ A \sqsubseteq b \longleftrightarrow (\forall\, a{\in}A.\ a \sqsubseteq b)$
by (*auto intro*: *Sup-least dest*: *Sup-upper*)

lemma *le-Inf-iff*: $b \sqsubseteq Inf\ A \longleftrightarrow (\forall\, a{\in}A.\ b \sqsubseteq a)$
by (*auto intro*: *Inf-greatest dest*: *Inf-lower*)

definition $SUPR :: 'b\ set \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$ **where**
$SUPR\ A\ f = \bigsqcup\ (f\ `\ A)$

definition $INFI :: 'b\ set \Rightarrow ('b \Rightarrow 'a) \Rightarrow 'a$ **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, 0, 10] 10*)
-INF1 :: *pttrns* => *'b* => *'b* (*(3INF -./ -) [0, 10] 10*)
-INF :: *pttrn* => *'a set* => *'b* => *'b* (*(3INF -:-./ -) [0, 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 ⟪
[*Syntax.preserve-binder-abs2-tr'* @{*const-syntax SUPR*} @{*syntax-const -SUP*},
Syntax.preserve-binder-abs2-tr' @{*const-syntax INFI*} @{*syntax-const -INF*}]
⟫ — to avoid eta-contraction of body

context *complete-lattice*
begin

lemma *le-SUPI*: $i : A \Longrightarrow M\ i \sqsubseteq (SUP\ i{:}A.\ M\ i)$
by (*auto simp add*: *SUPR-def intro*: *Sup-upper*)

lemma *SUP-leI*: $(\bigwedge i.\ i : A \Longrightarrow M\ i \sqsubseteq u) \Longrightarrow (SUP\ i{:}A.\ M\ i) \sqsubseteq u$
by (*auto simp add*: *SUPR-def intro*: *Sup-least*)

lemma *INF-leI*: $i : A \Longrightarrow (INF\ i{:}A.\ M\ i) \sqsubseteq M\ i$
by (*auto simp add*: *INFI-def intro*: *Inf-lower*)

lemma *le-INFI*: $(\bigwedge i.\ i : A \Longrightarrow u \sqsubseteq M\ i) \Longrightarrow u \sqsubseteq (INF\ i{:}A.\ M\ i)$
by (*auto simp add*: *INFI-def intro*: *Inf-greatest*)

lemma *SUP-le-iff*: $(SUP\ i{:}A.\ M\ i) \sqsubseteq u \longleftrightarrow (\forall\, i \in A.\ M\ i \sqsubseteq u)$
unfolding *SUPR-def* **by** (*auto simp add*: *Sup-le-iff*)

lemma *le-INF-iff*: $u \sqsubseteq (INF\ i{:}A.\ M\ i) \longleftrightarrow (\forall\, i \in A.\ u \sqsubseteq M\ i)$
unfolding *INFI-def* **by** (*auto simp add*: *le-Inf-iff*)

lemma *SUP-const*[*simp*]: $A \neq \{\} \Longrightarrow (SUP\ i{:}A.\ M) = M$
by (*auto intro*: *antisym SUP-leI le-SUPI*)

lemma *INF-const*[*simp*]: $A \neq \{\} \Longrightarrow (INF\ i{:}A.\ M) = M$
by (*auto intro*: *antisym INF-leI le-INFI*)

end

8.3 *bool* and - ⇒ - as complete lattice

instantiation *bool* :: *complete-lattice*
begin

definition
Inf-bool-def: $\bigsqcap A \longleftrightarrow (\forall\, x{\in}A.\ x)$

definition
Sup-bool-def: $\bigsqcup A \longleftrightarrow (\exists\, x{\in}A.\ x)$

instance proof
qed (*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*

lemma *INFI-bool-eq*:
INFI = *Ball*
proof (*rule ext*)+
fix *A* :: *'a set*
fix *P* :: *'a* ⇒ *bool*
show (*INF x*:*A*. *P x*) ⟷ (∀ *x* ∈ *A*. *P x*)
by (*auto simp add*: *Ball-def INFI-def Inf-bool-def*)
qed

lemma *SUPR-bool-eq*:
SUPR = *Bex*
proof (*rule ext*)+
fix *A* :: *'a set*
fix *P* :: *'a* ⇒ *bool*
show (*SUP x*:*A*. *P x*) ⟷ (∃ *x* ∈ *A*. *P x*)
by (*auto simp add*: *Bex-def SUPR-def Sup-bool-def*)
qed

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 proof
qed (*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 (*simp add*: *Inf-fun-def*)

lemma *Sup-empty-fun*:
$\bigsqcup\{\} = (\lambda\text{-}.\ \bigsqcup\{\})$
by (*simp add*: *Sup-fun-def*)

8.4 Union

abbreviation *Union* :: *'a set set* ⇒ *'a set* **where**
Union S ≡ $\bigsqcup S$

notation (*xsymbols*)
Union ($\bigcup$- [*90*] *90*)

lemma *Union-eq*:
$\bigcup A = \{x.\ \exists B \in A.\ x \in B\}$
proof (*rule set-ext*)
fix *x*
have ($\exists Q{\in}\{P.\ \exists B{\in}A.\ P \longleftrightarrow x \in B\}.\ Q$) $\longleftrightarrow$ ($\exists B{\in}A.\ x \in B$)
by *auto*
then show $x \in \bigcup A \longleftrightarrow x \in \{x.\ \exists B{\in}A.\ x \in B\}$
by (*simp add*: *Sup-fun-def Sup-bool-def*) (*simp add*: *mem-def*)
qed

lemma *Union-iff* [*simp*, *no-atp*]:
$A \in \bigcup C \longleftrightarrow (\exists X{\in}C.\ A{\in}X)$
by (*unfold Union-eq*) *blast*

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

lemma *UnionE* [*elim*!]:
$A \in \bigcup C \Longrightarrow (\bigwedge X.\ A{\in}X \Longrightarrow X{\in}C \Longrightarrow R) \Longrightarrow R$
by *auto*

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

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

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

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 ∪ ⋃ B*
by *blast*

lemma *Union-Un-distrib* [*simp*]: *⋃(A Un B) = ⋃ A ∪ ⋃ B*
by *blast*

lemma *Union-Int-subset*: *⋃(A ∩ B) ⊆ ⋃ A ∩ ⋃ B*
by *blast*

lemma *Union-empty-conv* [*simp*,*no-atp*]: *(⋃ A = {}) = (∀ x∈A. x = {})*
by *blast*

lemma *empty-Union-conv* [*simp*,*no-atp*]: *({} = ⋃ A) = (∀ x∈A. x = {})*
by *blast*

lemma *Union-disjoint*: *(⋃ C ∩ A = {}) = (∀ B∈C. B ∩ A = {})*
by *blast*

lemma *subset-Pow-Union*: *A ⊆ Pow (⋃ A)*
by *blast*

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

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

8.5 Unions of families

abbreviation *UNION* :: *'a set ⇒ ('a ⇒ 'b set) ⇒ 'b set* **where**
UNION ≡ SUPR

syntax
-UNION1 :: *pttrns => 'b set => 'b set* *((3UN -./ -) [0, 10] 10)*
-UNION :: *pttrn => 'a set => 'b set => 'b set* *((3UN -:-./ -) [0, 0, 10] 10)*

syntax (*xsymbols*)
-UNION1 :: *pttrns => 'b set => 'b set* *((3⋃ -./ -) [0, 10] 10)*
-UNION :: *pttrn => 'a set => 'b set => 'b set* *((3⋃ -∈-./ -) [0, 0, 10] 10)*

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

10)

translations
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.

print-translation ⟪
[*Syntax.preserve-binder-abs2-tr'* @{*const-syntax UNION*} @{*syntax-const -UNION*}]
⟫ — to avoid eta-contraction of body

lemma *UNION-eq-Union-image*:
$(\bigcup x{\in}A.\ B\ x) = \bigcup (B`A)$
by (*fact SUPR-def*)

lemma *Union-def*:
$\bigcup S = (\bigcup x{\in}S.\ x)$
by (*simp add*: *UNION-eq-Union-image image-def*)

lemma *UNION-def* [*no-atp*]:
$(\bigcup x{\in}A.\ B\ x) = \{y.\ \exists x{\in}A.\ y \in B\ x\}$
by (*auto simp add*: *UNION-eq-Union-image Union-eq*)

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

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

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

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

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

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

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

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

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

lemma *Collect-bex-eq* [*no-atp*]: $\{x.\ \exists y{\in}A.\ P\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ P\ x\ y\})$
by *blast*

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

lemma *UN-empty* [*simp*,*no-atp*]: $(\bigcup x{\in}\{\}.\ B\ x) = \{\}$
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 *UN-insert* [*simp*]: $(\bigcup x{\in}insert\ a\ A.\ B\ x) = B\ a \cup UNION\ A\ B$
by *blast*

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

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

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

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

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

by *auto*

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

lemma *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 *Collect-ex-eq* [*no-atp*]: $\{x.\ \exists y.\ P\ x\ y\} = (\bigcup y.\ \{x.\ P\ x\ y\})$
by *blast*

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

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

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 *UN-Pow-subset*: $(\bigcup x{\in}A.\ Pow\ (B\ x)) \subseteq Pow\ (\bigcup x{\in}A.\ B\ x)$
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 *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-eq-UN*: $f-`B = (UN\ y{:}\ B.\ f-`\{y\})$
— NOT suitable for rewriting
by *blast*

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

8.6 Inter

abbreviation *Inter* :: *'a set set* ⇒ *'a set* **where**

Inter S ≡ ⋂ *S*

notation (*xsymbols*)
Inter (⋂ - [*90*] *90*)

lemma *Inter-eq* [*code del*]:
⋂ *A* = {*x*. ∀ *B* ∈ *A*. *x* ∈ *B*}
proof (*rule set-ext*)
fix *x*
have (∀ *Q*∈{*P*. ∃ *B*∈*A*. *P* ⟷ *x* ∈ *B*}. *Q*) ⟷ (∀ *B*∈*A*. *x* ∈ *B*)
by *auto*
then show *x* ∈ ⋂ *A* ⟷ *x* ∈ {*x*. ∀ *B* ∈ *A*. *x* ∈ *B*}
by (*simp add*: *Inf-fun-def Inf-bool-def*) (*simp add*: *mem-def*)
qed

lemma *Inter-iff* [*simp*,*no-atp*]: (*A* : *Inter C*) = (*ALL X*:*C*. *A*:*X*)
by (*unfold Inter-eq*) *blast*

lemma *InterI* [*intro*!]: (!!*X*. *X*:*C* ==> *A*:*X*) ==> *A* : *Inter C*
by (*simp add*: *Inter-eq*)

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

lemma *InterD* [*elim*]: *A* : *Inter C* ==> *X*:*C* ==> *A*:*X*
by *auto*

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

lemma *Inter-lower*: *B* ∈ *A* ==> *Inter A* ⊆ *B*
by *blast*

lemma *Inter-subset*:
[| !!*X*. *X* ∈ *A* ==> *X* ⊆ *B*; *A* ~= {} |] ==> ⋂ *A* ⊆ *B*
by *blast*

lemma *Inter-greatest*: (!!*X*. *X* ∈ *A* ==> *C* ⊆ *X*) ==> *C* ⊆ *Inter A*
by (*iprover intro*: *InterI subsetI dest*: *subsetD*)

lemma *Int-eq-Inter*: *A* ∩ *B* = ⋂ {*A*, *B*}
by *blast*

lemma *Inter-empty* [*simp*]: ⋂ {} = *UNIV*
by *blast*

lemma *Inter-UNIV* [*simp*]: ⋂ *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*,*no-atp*]:
$(\bigcap A = \mathit{UNIV}) = (\forall x{\in}A.\ x = \mathit{UNIV})$
$(\mathit{UNIV} = \bigcap A) = (\forall x{\in}A.\ x = \mathit{UNIV})$
by *blast+*

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

8.7 Intersections of families

abbreviation *INTER* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ *'b set* **where**
INTER ≡ *INFI*

syntax
-INTER1 :: *pttrns => 'b set => 'b set* (*(3INT -./ -) [0, 10] 10*)
-INTER :: *pttrn => 'a set => 'b set => 'b set* (*(3INT -:-./ -) [0, 0, 10] 10*)

syntax (*xsymbols*)
-INTER1 :: *pttrns => 'b set => 'b set* (*(3⋂-./ -) [0, 10] 10*)
-INTER :: *pttrn => 'a set => 'b set => 'b set* (*(3⋂-∈-./ -) [0, 0, 10] 10*)

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

translations
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*)

print-translation ⟪
[*Syntax.preserve-binder-abs2-tr'* @{*const-syntax INTER*} @{*syntax-const -INTER*}]
⟫ — to avoid eta-contraction of body

lemma *INTER-eq-Inter-image*:
$(\bigcap x{\in}A.\ B\ x) = \bigcap(B`A)$

by (*fact INFI-def*)

lemma *Inter-def*:
$\bigcap S = (\bigcap x{\in}S.\ x)$
by (*simp add*: *INTER-eq-Inter-image image-def*)

lemma *INTER-def*:
$(\bigcap x{\in}A.\ B\ x) = \{y.\ \forall x{\in}A.\ y \in B\ x\}$
by (*auto simp add*: *INTER-eq-Inter-image Inter-eq*)

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

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*)

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

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

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

lemma *INT-greatest*: $(!!x.\ x \in A \Longrightarrow C \subseteq B\ x) \Longrightarrow C \subseteq (\bigcap x{\in}A.\ B\ x)$
by (*fact le-INFI*)

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

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

by *blast*

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

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

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

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

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

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

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

8.8 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*

8.9 Complement

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*

8.10 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

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

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

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

lemma *ball-simps* [*simp*,*no-atp*]:
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ |\ Q) = ((ALL\ x{:}A.\ P\ x)\ |\ Q)$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ |\ Q\ x) = (P\ |\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ --{>}\ Q\ x) = (P\ --{>}\ (ALL\ x{:}A.\ Q\ x))$
!!$A\ P\ Q.\ (ALL\ x{:}A.\ P\ x\ --{>}\ Q) = ((EX\ x{:}A.\ P\ x)\ --{>}\ Q)$
!!$P.\ (ALL\ x{:}\{\}.\ P\ x) = True$
!!$P.\ (ALL\ x{:}UNIV.\ P\ x) = (ALL\ x.\ P\ x)$
!!$a\ B\ P.\ (ALL\ x{:}insert\ a\ B.\ P\ x) = (P\ a\ \&\ (ALL\ x{:}B.\ P\ x))$
!!$A\ P.\ (ALL\ x{:}Union\ A.\ P\ x) = (ALL\ y{:}A.\ ALL\ x{:}y.\ P\ x)$
!!$A\ B\ P.\ (ALL\ x{:}\ UNION\ A\ B.\ P\ x) = (ALL\ a{:}A.\ ALL\ x{:}\ B\ a.\ P\ x)$
!!$P\ Q.\ (ALL\ x{:}Collect\ Q.\ P\ x) = (ALL\ x.\ Q\ x\ --{>}\ P\ x)$
!!$A\ P\ f.\ (ALL\ x{:}f`A.\ P\ x) = (ALL\ x{:}A.\ P\ (f\ x))$
!!$A\ P.\ (\sim(ALL\ x{:}A.\ P\ x)) = (EX\ x{:}A.\ \sim P\ x)$
by *auto*

lemma *bex-simps* [*simp*,*no-atp*]:
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ x\ \&\ Q) = ((EX\ x{:}A.\ P\ x)\ \&\ Q)$
!!$A\ P\ Q.\ (EX\ x{:}A.\ P\ \&\ Q\ x) = (P\ \&\ (EX\ x{:}A.\ Q\ x))$
!!$P.\ (EX\ x{:}\{\}.\ P\ x) = False$

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

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

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*) **and**
top ($\top$) **and**
bot ($\bot$)

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.

end

9 Inductive: Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*
imports *Complete-Lattice*
uses
(*Tools/inductive.ML*)
Tools/dseq.ML
(*Tools/inductive-codegen.ML*)
Tools/Datatype/datatype-aux.ML
Tools/Datatype/datatype-prop.ML
Tools/Datatype/datatype-case.ML
(*Tools/Datatype/datatype-abs-proofs.ML*)
(*Tools/Datatype/datatype-data.ML*)
(*Tools/old-primrec.ML*)
(*Tools/primrec.ML*)
(*Tools/Datatype/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*: $f\ A \leq A \Longrightarrow$ *lfp f* $\leq A$
by (*auto simp add*: *lfp-def intro*: *Inf-lower*)

lemma *lfp-greatest*: $(!!u.\ f\ u \leq u \Longrightarrow A \leq u) \Longrightarrow A \leq$ *lfp f*
by (*auto simp add*: *lfp-def intro*: *Inf-greatest*)

end

lemma *lfp-lemma2*: *mono f* $\Longrightarrow$ *f* (*lfp f*) $\leq$ *lfp f*
by (*iprover intro*: *lfp-greatest order-trans monoD lfp-lowerbound*)

lemma *lfp-lemma3*: *mono f* $\Longrightarrow$ *lfp f* $\leq$ *f* (*lfp f*)
by (*iprover intro*: *lfp-lemma2 monoD lfp-lowerbound*)

lemma *lfp-unfold*: *mono f* $\Longrightarrow$ *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*: *intro*: *indhyp*)

lemma *lfp-ordinal-induct*:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'a*

assumes *mono*: *mono f*
and *P-f*: $\bigwedge S.\ P\ S \Longrightarrow P\ (f\ S)$
and *P-Union*: $\bigwedge M.\ \forall S{\in}M.\ P\ S \Longrightarrow P\ (Sup\ M)$
shows *P* (*lfp f*)
proof −
let *?M* = $\{S.\ S \leq lfp\ f \wedge P\ S\}$
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* **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 $\{u.\ u \leq f\ u\}$

lemma *gfp-upperbound*: $X \leq f\ X \Longrightarrow X \leq gfp\ f$
by (*auto simp add*: *gfp-def intro*: *Sup-upper*)

lemma *gfp-least*: $(!!u.\ u \leq f\ u \Longrightarrow u \leq X) \Longrightarrow gfp\ f \leq X$
by (*auto simp add*: *gfp-def intro*: *Sup-least*)

lemma *gfp-lemma2*: $mono\ f \Longrightarrow gfp\ f \leq f\ (gfp\ f)$
by (*iprover intro*: *gfp-least order-trans monoD gfp-upperbound*)

lemma *gfp-lemma3*: $mono\ f \Longrightarrow f\ (gfp\ f) \leq gfp\ f$
by (*iprover intro*: *gfp-lemma2 monoD gfp-upperbound*)

lemma *gfp-unfold*: $mono\ f \Longrightarrow gfp\ f = f\ (gfp\ f)$
by (*iprover intro*: *order-antisym gfp-lemma2 gfp-lemma3*)

9.5 Coinduction rules for greatest fixed points

weak version

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

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

lemma *coinduct-lemma*:
$[|\ X \leq f\ (sup\ X\ (gfp\ f));\ \ mono\ f\ |] \Longrightarrow sup\ X\ (gfp\ f) \leq f\ (sup\ X\ (gfp\ f))$
apply (*frule gfp-lemma2*)
apply (*drule mono-sup*)
apply (*rule le-supI*)
apply *assumption*
apply (*rule order-trans*)
apply (*rule order-trans*)
apply *assumption*
apply (*rule sup-ge2*)
apply *assumption*
done

strong version, thanks to Coen and Frost

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

lemma *coinduct*: $[|\ mono(f);\ X \leq f\ (sup\ X\ (gfp\ f))\ |] \Longrightarrow X \leq gfp(f)$
apply (*rule order-trans*)
apply (*rule sup-ge1*)
apply (*erule gfp-upperbound* [*OF coinduct-lemma*])
apply *assumption*
done

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

9.6 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition $X \subseteq f\ X$ to one expressed using both *lfp* and *gfp*

lemma *coinduct3-mono-lemma*: *mono*(*f*) ==> *mono*(%*x*. *f*(*x*) *Un X Un B*)
by (*iprover intro*: *subset-refl monoI Un-mono monoD*)

lemma *coinduct3-lemma*:
[| *X* ⊆ *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; $X \subseteq f$(*lfp*(%x. $f(x)$ *Un* X *Un* A)) |] ==> a: A
by (*auto intro*!: *coinduct3*)

Monotonicity of *gfp*!

lemma *gfp-mono*: (!!Z. f $Z \leq g$ Z) ==> *gfp* $f \leq$ *gfp* g
by (*rule gfp-upperbound* [*THEN gfp-least*], *blast intro*: *order-trans*)

9.7 Inductive predicates and sets

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

use *Tools/inductive.ML*
setup *Inductive.setup*

theorems [*mono*] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-mono not-mono
Ball-def Bex-def
induct-rulify-fallback

9.8 Inductive datatypes and primitive recursion

Package setup.

use *Tools/Datatype/datatype-abs-proofs.ML*
use *Tools/Datatype/datatype-data.ML*
setup *Datatype-Data.setup*

use *Tools/Datatype/datatype-codegen.ML*
setup *Datatype-Codegen.setup*

use *Tools/old-primrec.ML*
use *Tools/primrec.ML*

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

Lambda-abstractions with pattern matching:

syntax
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((%-) *10*)
syntax (*xsymbols*)
-lam-pats-syntax :: *cases-syn* => *'a* => *'b* ((λ-) *10*)

parse-translation (**advanced**) ⟪
let
fun fun-tr ctxt [*cs*] =

```
    let
      val x = Free (Name.variant (Term.add-free-names cs []) x, dummyT);
      val ft = Datatype-Case.case-tr true Datatype-Data.info-of-constr ctxt [x, cs];
    in lambda x ft end
in [(@{syntax-const -lam-pats-syntax}, fun-tr)] end
⟩⟩
```

end

10 Fun: Notions about functions

theory *Fun*
imports *Complete-Lattice*
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*

10.1 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*)

10.2 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 *o-eq-dest*:
$a\ o\ b = c\ o\ d \Longrightarrow a\ (b\ v) = c\ (d\ v)$
by (*simp only*: *o-def*) (*fact fun-cong*)

lemma *o-eq-elim*:
$a\ o\ b = c\ o\ d \Longrightarrow ((\bigwedge v.\ a\ (b\ v) = c\ (d\ v)) \Longrightarrow R) \Longrightarrow R$
by (*erule meta-mp*) (*fact o-eq-dest*)

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

lemma *vimage-compose*: $(g \circ f) -` x = f -` (g -` x)$
by *auto*

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

10.3 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*)

code-const *fcomp*
(*Eval* **infixl** *1 #>*)

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

10.4 Injectivity and Surjectivity

definition
inj-on :: [$'a$ => $'b$, $'a$ *set*] => *bool* **where**
— injective
inj-on f A == ! *x*:*A*. ! *y*:*A*. *f*(*x*)=*f*(*y*) −−> *x*=*y*

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

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

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

definition
surj :: ($'a$ => $'b$) => *bool* **where**
— surjective
surj f == ! *y*. ? *x*. *y*=*f*(*x*)

definition
bij :: ($'a$ => $'b$) => *bool* **where**
— bijective
bij f == *inj f* & *surj f*

lemma *injI*:
assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
shows *inj f*

using *assms* **unfolding** *inj-on-def* **by** *auto*

For Proofs in *Tools/Datatype/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-on-eq-iff*: *inj-on f A* ==> *x*:*A* ==> *y*:*A* ==> (*f*(*x*) = *f*(*y*)) = (*x*=*y*)
by (*force simp add*: *inj-on-def*)

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

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*)

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-comp*: *bij f* ⟹ *bij g* ⟹ *bij* (*g o f*)
by(*fastsimp intro*: *comp-inj-on comp-surj simp*: *bij-def surj-range*)

lemma *bij-betw-trans*:
bij-betw f A B ⟹ *bij-betw g B C* ⟹ *bij-betw* (*g o f*) *A C*
by(*auto simp add*:*bij-betw-def comp-inj-on*)

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 $?g\ b \in A$ **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** $\exists\, b{\in}B.\ a = ?g\ b$ **by** *blast*
qed
ultimately show *?thesis* **by**(*auto simp*:*bij-betw-def*)
qed

lemma *surj-image-vimage-eq*: $surj\ f \Longrightarrow f\ `\ (f\ -`\ A) = A$
by (*simp add*: *surj-range*)

lemma *inj-vimage-image-eq*: $inj\ f \Longrightarrow f\ -`\ (f\ `\ A) = A$
by (*simp add*: *inj-on-def*, *blast*)

lemma *vimage-subsetD*: $surj\ f \Longrightarrow f\ -`\ B <= A \Longrightarrow B <= f\ `\ A$
apply (*unfold surj-def*)
apply (*blast intro*: *sym*)
done

lemma *vimage-subsetI*: $inj\ f \Longrightarrow B <= f\ `\ A \Longrightarrow f\ -`\ B <= A$
by (*unfold inj-on-def*, *blast*)

lemma *vimage-subset-eq*: $bij\ f \Longrightarrow (f\ -`\ B <= A) = (B <= f\ `\ A)$
apply (*unfold bij-def*)
apply (*blast del*: *subsetI intro*: *vimage-subsetI vimage-subsetD*)
done

lemma *inj-on-Un-image-eq-iff*: $inj\text{-}on\ f\ (A \cup B) \Longrightarrow f\ `\ A = f\ `\ B \longleftrightarrow A = B$
by(*blast dest*: *inj-onD*)

lemma *inj-on-image-Int*:
$[|\ inj\text{-}on\ f\ C;\ \ A<=C;\ \ B<=C\ |] \Longrightarrow f`(A\ Int\ B) = f`A\ Int\ f`B$
apply (*simp add*: *inj-on-def*, *blast*)
done

lemma *inj-on-image-set-diff*:
$[|\ inj\text{-}on\ f\ C;\ \ A<=C;\ \ B<=C\ |] \Longrightarrow f`(A-B) = f`A - f`B$
apply (*simp add*: *inj-on-def*, *blast*)
done

lemma *image-Int*: $inj\ f \Longrightarrow f`(A\ Int\ B) = f`A\ Int\ f`B$
by (*simp add*: *inj-on-def*, *blast*)

lemma *image-set-diff*: $inj\ f \Longrightarrow f`(A-B) = f`A - f`B$
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

lemma (**in** *ordered-ab-group-add*) *inj-uminus*[*simp*, *intro*]: *inj-on uminus A*
by (*auto intro*!: *inj-onI*)

lemma (**in** *linorder*) *strict-mono-imp-inj-on*: *strict-mono f ⟹ inj-on f A*
by (*auto intro*!: *inj-onI dest*: *strict-mono-eq*)

10.5 Function Updating

definition
fun-upd :: (*'a => 'b*) *=> 'a => 'b => ('a => 'b*) **where**
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*) == *CONST 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*

lemma *fun-upd-comp*: *f* ∘ (*g*(*x* := *y*)) = (*f* ∘ *g*)(*x* := *f y*)

by (*auto intro*: *ext*)

10.6 *override-on*

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

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

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

10.7 *swap*

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

lemma *swap-self* [*simp*]: *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 *swap-triple*:
assumes *a* $\neq$ *c* **and** *b* $\neq$ *c*
shows *swap a b* (*swap b c* (*swap a b f*)) = *swap a c f*
using *assms* **by** (*simp add*: *expand-fun-eq swap-def*)

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

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

lemma *inj-on-swap-iff* [*simp*]:
assumes *A*: *a* $\in$ *A b* $\in$ *A* **shows** *inj-on* (*swap a b f*) *A* = *inj-on f A*
proof
assume *inj-on* (*swap a b f*) *A*

with *A* **have** *inj-on* (*swap a b* (*swap a b f*)) *A*
by (*iprover intro*: *inj-on-imp-inj-on-swap*)
thus *inj-on f A* **by** *simp*
next
assume *inj-on f A*
with *A* **show** *inj-on* (*swap a b f*) *A* **by** (*iprover intro*: *inj-on-imp-inj-on-swap*)
qed

lemma *surj-imp-surj-swap*: *surj f* ==> *surj* (*swap a b f*)
apply (*simp add*: *surj-def swap-def*, *clarify*)
apply (*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-const (**open**) *swap*

10.8 Inversion of injective functions

definition *the-inv-into* :: *'a set* => (*'a* => *'b*) => (*'b* => *'a*) **where**
the-inv-into A f == %*x*. *THE y*. *y* : *A* & *f y* = *x*

lemma *the-inv-into-f-f*:
[| *inj-on f A*; *x* : *A* |] ==> *the-inv-into A f* (*f x*) = *x*
apply (*simp add*: *the-inv-into-def inj-on-def*)
apply *blast*
done

lemma *f-the-inv-into-f*:
inj-on f A ==> *y* : *f*'*A* ==> *f* (*the-inv-into A f y*) = *y*
apply (*simp add*: *the-inv-into-def*)
apply (*rule the1I2*)
apply(*blast dest*: *inj-onD*)
apply *blast*
done

lemma *the-inv-into-into*:

```
  [| inj-on f A; x : f ` A; A <= B |] ==> the-inv-into A f x : B
apply (simp add: the-inv-into-def)
apply (rule the1I2)
 apply(blast dest: inj-onD)
apply blast
done

lemma the-inv-into-onto[simp]:
  inj-on f A ==> the-inv-into A f ` (f ` A) = A
by (fast intro:the-inv-into-into the-inv-into-f-f[symmetric])

lemma the-inv-into-f-eq:
  [| inj-on f A; f x = y; x : A |] ==> the-inv-into A f y = x
  apply (erule subst)
  apply (erule the-inv-into-f-f, assumption)
  done

lemma the-inv-into-comp:
  [| inj-on f (g ` A); inj-on g A; x : f ` g ` A |] ==>
  the-inv-into A (f o g) x = (the-inv-into A g o the-inv-into (g ` A) f) x
apply (rule the-inv-into-f-eq)
  apply (fast intro: comp-inj-on)
 apply (simp add: f-the-inv-into-f the-inv-into-into)
apply (simp add: the-inv-into-into)
done

lemma inj-on-the-inv-into:
  inj-on f A ⟹ inj-on (the-inv-into A f) (f ` A)
by (auto intro: inj-onI simp: image-def the-inv-into-f-f)

lemma bij-betw-the-inv-into:
  bij-betw f A B ⟹ bij-betw (the-inv-into A f) B A
by (auto simp add: bij-betw-def inj-on-the-inv-into the-inv-into-into)

abbreviation the-inv :: ('a ⇒ 'b) ⇒ ('b ⇒ 'a) where
  the-inv f ≡ the-inv-into UNIV f

lemma the-inv-f-f:
  assumes inj f
  shows the-inv f (f x) = x using assms UNIV-I
  by (rule the-inv-into-f-f)
```

10.9 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
⟩⟩
```

10.10 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 = Unsynchronized.ref [];
    fun mk-upd (x, (-, y)) t = Const (Fun.fun-upd,
      (aT --> bT) --> aT --> bT --> aT --> bT) $ t $ aF x $ y ()
  in
    (fn x =>
       case AList.lookup op = (!tab) x of
         NONE =>
           let val p as (y, -) = bG i
           in (tab := (x, p) :: !tab; y) end
       | SOME (y, -) => y,
     fn () => Basics.fold mk-upd (!tab) (Const (HOL.undefined, aT --> bT)))
  end;
⟩⟩
```

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

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

end

11 Product-Type: Cartesian products

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

11.1 *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* −)

11.2 The *unit* type

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

definition

Unity :: *unit* (′(′))
where
() = *Abs-unit True*

lemma *unit-eq* [*no-atp*]: *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*,*no-atp*]: (%*u*::*unit*. *f* ()) = *f*
by (*rule ext*) *simp*

instantiation *unit* :: *default*
begin

definition *default* = ()

instance ..

end

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* ())
(*Scala Unit*)

code-const *Unity*
(*SML* ())
(*OCaml* ())
(*Haskell* ())
(*Scala* ())

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

code-const *eq-class.eq* :: *unit* ⇒ *unit* ⇒ *bool*
(*Haskell* **infixl** *4* ==)

code-reserved *SML*
unit

code-reserved *OCaml*
unit

code-reserved *Scala*
Unit

11.3 The product type

11.3.1 Type definition

definition *Pair-Rep* :: *'a* ⇒ *'b* ⇒ *'a* ⇒ *'b* ⇒ *bool* **where**
Pair-Rep a b = (λ*x y*. *x* = *a* ∧ *y* = *b*)

global

typedef (*Prod*)
(*'a*, *'b*) ∗ (**infixr** ∗ *20*)
= {*f*. ∃ *a b*. *f* = *Pair-Rep* (*a*::*'a*) (*b*::*'b*)}
proof
fix *a b* **show** *Pair-Rep a b* ∈ *?Prod*
by *rule*+
qed

type-notation (*xsymbols*)
∗ ((- ×/ -) [*21*, *20*] *20*)
type-notation (*HTML* **output**)
∗ ((- ×/ -) [*21*, *20*] *20*)

consts
Pair :: *'a* ⇒ *'b* ⇒ *'a* × *'b*

local

defs
Pair-def: *Pair a b* == *Abs-Prod* (*Pair-Rep a b*)

rep-datatype (*prod*) *Pair* **proof** −
fix *P* :: *'a* × *'b* ⇒ *bool* **and** *p*
assume ⋀*a b*. *P* (*Pair a b*)
then show *P p* **by** (*cases p*) (*auto simp add*: *Prod-def Pair-def Pair-Rep-def*)
next
fix *a c* :: *'a* **and** *b d* :: *'b*
have *Pair-Rep a b* = *Pair-Rep c d* ⟷ *a* = *c* ∧ *b* = *d*
by (*auto simp add*: *Pair-Rep-def expand-fun-eq*)
moreover have *Pair-Rep a b* ∈ *Prod* **and** *Pair-Rep c d* ∈ *Prod*
by (*auto simp add*: *Prod-def*)
ultimately show *Pair a b* = *Pair c d* ⟷ *a* = *c* ∧ *b* = *d*
by (*simp add*: *Pair-def Abs-Prod-inject*)
qed

11.3.2 Tuple syntax

global consts
split :: (*'a* ⇒ *'b* ⇒ *'c*) ⇒ *'a* × *'b* ⇒ *'c*

local defs
split-prod-case: *split* == *prod-case*

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*) == *CONST Pair x y*
-tuple x (*-tuple-args y z*) == *-tuple x* (*-tuple-arg* (*-tuple y z*))
%(*x*, *y*, *zs*). *b* == *CONST split* (%*x* (*y*, *zs*). *b*)
%(*x*, *y*). *b* == *CONST split* (%*x y*. *b*)
-abs (*CONST Pair x y*) *t* => %(*x*, *y*). *t*
— The last rule accommodates tuples in 'case C ... (x,y) ... =¿ ...' The (x,y) is parsed as 'Pair x y' because it is logic, not pttrn

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 @{syntax-const -abs} $
            (Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
        end
    | split-tr' [Abs (x, T, (s as Const (@{const-syntax split}, -) $ t))] =
        (* split (%x. (split (%y z. t))) => %(x,y,z). t *)
        let
          val Const (@{syntax-const -abs}, -) $
            (Const (@{syntax-const -pattern}, -) $ y $ z) $ t' = split-tr' [t];
          val (x', t'') = atomic-abs-tr' (x, T, t');
        in
          Syntax.const @{syntax-const -abs} $
            (Syntax.const @{syntax-const -pattern} $ x' $
              (Syntax.const @{syntax-const -patterns} $ y $ z)) $ t''
        end
    | split-tr' [Const (@{const-syntax 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 (@{syntax-const -abs}, -) $ x-y $ Abs abs] =
        (* split (%pttrn z. t) => %(pttrn,z). t *)
        let val (z, t) = atomic-abs-tr' abs in
          Syntax.const @{syntax-const -abs} $
            (Syntax.const @{syntax-const -pattern} $ x-y $ z) $ t
        end
    | split-tr' - = raise Match;
in [(@{const-syntax 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 (@{const-syntax 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 @{syntax-const -abs} $
```

```
          (Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
        end)
  | split-guess-names-tr' - T [t] =
      (case head-of t of
        Const (@{const-syntax 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 @{syntax-const -abs} $
            (Syntax.const @{syntax-const -pattern} $ x' $ y) $ t''
        end)
  | split-guess-names-tr' - - - = raise Match;
in [(@{const-syntax split}, split-guess-names-tr')] end
⟫
```

11.3.3 Code generator setup

```
lemma split-case-cert:
  assumes CASE ≡ split f
  shows CASE (a, b) ≡ f a b
  using assms by (simp add: split-prod-case)

setup ⟪
  Code.add-case @{thm split-case-cert}
⟫

code-type *
  (SML infix 2 *)
  (OCaml infix 2 *)
  (Haskell !((-),/ (-)))
  (Scala ((-),/ (-)))

code-const Pair
  (SML !((-),/ (-)))
  (OCaml !((-),/ (-)))
  (Haskell !((-),/ (-)))
  (Scala !((-),/ (-)))

code-instance * :: eq
  (Haskell -)

code-const eq-class.eq :: 'a::eq × 'b::eq ⇒ 'a × 'b ⇒ bool
  (Haskell infixl 4 ==)
```

```
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 (@{const-name 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 (@{const-name Let}, -), t2 :: t3 :: ts) =>
    let
      fun dest-let (l as Const (@{const-name 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, flat
                (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 (@{const-name 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
⟩⟩
```

11.3.4 Fundamental operations and properties

lemma *surj-pair* [*simp*]: *EX x y. p = (x, y)*
  by (*cases p*) *simp*

global consts
  fst :: *'a × 'b ⇒ 'a*
  snd :: *'a × 'b ⇒ 'b*

local defs
fst-def: *fst p == case p of (a, b) ⇒ a*
snd-def: *snd p == case p of (a, b) ⇒ b*

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

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

code-const *fst* **and** *snd*
(*Haskell fst* **and** *snd*)

lemma *prod-case-unfold*: *prod-case = (%c p. c (fst p) (snd p))*
by (*simp add*: *expand-fun-eq split*: *prod.split*)

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 *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*)

lemma *split-conv* [*simp*, *code*]: *split f (a, b) = f a b*
by (*simp add*: *split-prod-case*)

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 *split-Pair* [*simp*]: *(λ(x, y). (x, y)) = id*
by (*simp add*: *split-prod-case expand-fun-eq split*: *prod.split*)

lemma *split-eta*: *(λ(x, y). f (x, y)) = f*
— Subsumes the old *split-Pair* when *f* is the identity function.
by (*simp add*: *split-prod-case expand-fun-eq split*: *prod.split*)

lemma *split-comp*: *split* $(f \circ g)$ $x = f$ $(g$ $(fst\ x))$ $(snd\ x)$
by (*cases x*) *simp*

lemma *split-twice*: *split f* (*split g p*) = *split* $(\lambda x\ y.$ *split f* $(g\ x\ y))$ *p*
by (*cases p*) *simp*

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

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*)

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 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)
```

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 @{thms cond-split-eta};
  fun Pair-pat k 0 (Bound m) = (m = k)
    | Pair-pat k i (Const (@{const-name 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 (@{const-name 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 (@{const_name 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 (@{const_name split}, _) $ Abs (_, _, t)) =
        (case split_pat eta_term_pat 1 t of
          SOME (_, ft) => SOME (metaeq ss s (let val (f $ arg) = ft in f end))
        | NONE => NONE)
    | eta_proc _ _ = NONE;
in
  val split_beta_proc = Simplifier.simproc @{theory} split_beta [split f z] (K beta_proc);
  val split_eta_proc = Simplifier.simproc @{theory} split_eta [split f] (K eta_proc);
end;

Addsimprocs [split_beta_proc, split_eta_proc];
⟫
```

lemma *split_beta* [*mono*]: (%(*x*, *y*). *P x y*) *z* = *P* (*fst z*) (*snd z*)
by (*subst surjective_pairing*, *rule split_conv*)

lemma *split_split* [*no_atp*]: *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* [*no_atp*]: *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-prod-case*)

lemma *splitE'*: *split c p z* ==> (!!*x y*. *p* = (*x*, *y*) ==> *c x y z* ==> *Q*) ==> *Q*
by (*induct p*) (*auto simp add*: *split-prod-case*)

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-prod-case prod-case-unfold*] *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* (@{*const-name split*},-) $ - $ (*Const* (@{*const-name*
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* @{*thms 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*,*no-atp*]: (%*u*. *EX x y*. *u* = (*x*, *y*) & *P* (*x*, *y*)) = *P*
by (*rule ext*) *fast*

lemma *split-eta-SetCompr2* [*simp*,*no-atp*]: (%*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*.

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*

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* **..**

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*

lemma *split-def*:
split = (λ*c p*. *c* (*fst p*) (*snd p*))
unfolding *split-prod-case prod-case-unfold* **..**

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*)

use *Tools/split-rule.ML*
setup *Split-Rule.setup*

hide-const *internal-split*

11.3.5 Derived operations

global consts
curry :: ('*a* × '*b* ⇒ '*c*) ⇒ '*a* ⇒ '*b* ⇒ '*c*

local defs
curry-def: *curry* == (%*c x y*. *c* (*Pair x y*))

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

lemma *curryI* [*intro*!]: *f* (*a*, *b*) ⟹ *curry f a b*
by (*simp add*: *curry-def*)

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

lemma *curryE*: *curry f a b* ⟹ (*f* (*a*, *b*) ⟹ *Q*) ⟹ *Q*
by (*simp add*: *curry-def*)

lemma *curry-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*)

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** $o\!\rightarrow$ *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*)

code-const *scomp*
(*Eval* **infixl** *3* #−>)

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 *fst-prod-fun*[*simp*]: $fst\ (prod\text{-}fun\ f\ g\ x) = f\ (fst\ x)$
by (*cases x*, *auto*)

lemma *snd-prod-fun*[*simp*]: $snd\ (prod\text{-}fun\ f\ g\ x) = g\ (snd\ x)$
by (*cases x*, *auto*)

lemma *fst-comp-prod-fun*[*simp*]: $fst \circ prod\text{-}fun\ f\ g = f \circ fst$
by (*rule ext*) *auto*

lemma *snd-comp-prod-fun*[*simp*]: $snd \circ prod\text{-}fun\ f\ g = g \circ snd$
by (*rule ext*) *auto*

lemma *prod-fun-compose*:

prod-fun (*f1 o f2*) (*g1 o g2*) = (*prod-fun f1 g1 o prod-fun f2 g2*)
by (*rule ext*) *auto*

lemma *prod-fun-ident* [*simp*]: *prod-fun* (%*x*. *x*) (%*y*. *y*) = (%*z*. *z*)
by (*rule ext*) *auto*

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

lemma *prod-fun-imageE* [*elim*!]:
assumes *major*: *c*: (*prod-fun f g*)‘*r*
and *cases*: !!*x y*. [| *c*=(*f*(*x*),*g*(*y*)); (*x*,*y*):*r* |] ==> *P*
shows *P*
apply (*rule major* [*THEN imageE*])
apply (*case-tac x*)
apply (*rule cases*)
apply (*blast intro*: *prod-fun*)
apply *blast*
done

definition *apfst* :: (′*a* ⇒ ′*c*) ⇒ ′*a* × ′*b* ⇒ ′*c* × ′*b* **where**
apfst f = *prod-fun f id*

definition *apsnd* :: (′*b* ⇒ ′*c*) ⇒ ′*a* × ′*b* ⇒ ′*a* × ′*c* **where**
apsnd f = *prod-fun id f*

lemma *apfst-conv* [*simp*, *code*]:
apfst f (*x*, *y*) = (*f x*, *y*)
by (*simp add*: *apfst-def*)

lemma *apsnd-conv* [*simp*, *code*]:
apsnd f (*x*, *y*) = (*x*, *f y*)
by (*simp add*: *apsnd-def*)

lemma *fst-apfst* [*simp*]:
fst (*apfst f x*) = *f* (*fst x*)
by (*cases x*) *simp*

lemma *fst-apsnd* [*simp*]:
fst (*apsnd f x*) = *fst x*
by (*cases x*) *simp*

lemma *snd-apfst* [*simp*]:
snd (*apfst f x*) = *snd x*
by (*cases x*) *simp*

lemma *snd-apsnd* [*simp*]:
snd (*apsnd f x*) = *f* (*snd x*)
by (*cases x*) *simp*

lemma *apfst-compose*:
apfst f (*apfst g x*) = *apfst* (*f* ∘ *g*) *x*
by (*cases x*) *simp*

lemma *apsnd-compose*:
apsnd f (*apsnd g x*) = *apsnd* (*f* ∘ *g*) *x*
by (*cases x*) *simp*

lemma *apfst-apsnd* [*simp*]:
apfst f (*apsnd g x*) = (*f* (*fst x*), *g* (*snd x*))
by (*cases x*) *simp*

lemma *apsnd-apfst* [*simp*]:
apsnd f (*apfst g x*) = (*g* (*fst x*), *f* (*snd x*))
by (*cases x*) *simp*

lemma *apfst-id* [*simp*] :
apfst id = *id*
by (*simp add*: *expand-fun-eq*)

lemma *apsnd-id* [*simp*] :
apsnd id = *id*
by (*simp add*: *expand-fun-eq*)

lemma *apfst-eq-conv* [*simp*]:
apfst f x = *apfst g x* ⟷ *f* (*fst x*) = *g* (*fst x*)
by (*cases x*) *simp*

lemma *apsnd-eq-conv* [*simp*]:
apsnd f x = *apsnd g x* ⟷ *f* (*snd x*) = *g* (*snd x*)
by (*cases x*) *simp*

lemma *apsnd-apfst-commute*:
apsnd f (*apfst g p*) = *apfst g* (*apsnd f p*)
by *simp*

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* == *CONST 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*) ∈ *A* × *B* – introduces no eigenvariables.

lemma *SigmaD1*: (*a*, *b*) : *Sigma A B* ==> *a* : *A*
by *blast*

lemma *SigmaD2*: (*a*, *b*) : *Sigma A B* ==> *b* : *B a*
by *blast*

lemma *SigmaE2*:
[| (*a*, *b*) : *Sigma A B*;
[| *a*:*A*; *b*:*B*(*a*) |] ==> *P*
|] ==> *P*
by *blast*

lemma *Sigma-cong*:
⟦*A* = *B*; !!*x*. *x* ∈ *B* ⟹ *C x* = *D x*⟧
⟹ (*SIGMA x*: *A*. *C x*) = (*SIGMA x*: *B*. *D x*)
by *auto*

lemma *Sigma-mono*: [| *A* <= *C*; !!*x*. *x*:*A* ==> *B x* <= *D x* |] ==> *Sigma A B* <= *Sigma C D*
by *blast*

lemma *Sigma-empty1* [*simp*]: *Sigma* {} *B* = {}
by *blast*

lemma *Sigma-empty2* [*simp*]: *A* <*> {} = {}
by *blast*

lemma *UNIV-Times-UNIV* [*simp*]: $UNIV <*> UNIV = UNIV$
by *auto*

lemma *Compl-Times-UNIV1* [*simp*]: $- (UNIV <*> A) = UNIV <*> (-A)$
by *auto*

lemma *Compl-Times-UNIV2* [*simp*]: $- (A <*> UNIV) = (-A) <*> UNIV$
by *auto*

lemma *mem-Sigma-iff* [*iff*]: $((a,b): Sigma\ A\ B) = (a{:}A\ \&\ b{:}B(a))$
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*,*no-atp*]:
$(ALL\ z{:}\ Sigma\ A\ B.\ P\ z) = (ALL\ x{:}A.\ ALL\ y{:}\ B\ x.\ P(x,y))$
by *blast*

lemma *split-paired-Bex-Sigma* [*simp*,*no-atp*]:
$(EX\ z{:}\ Sigma\ A\ B.\ P\ z) = (EX\ x{:}A.\ EX\ y{:}\ B\ x.\ P(x,y))$
by *blast*

lemma *Sigma-Un-distrib1*: $(SIGMA\ i{:}I\ Un\ J.\ C(i)) = (SIGMA\ i{:}I.\ C(i))\ Un\ (SIGMA\ j{:}J.\ C(j))$
by *blast*

lemma *Sigma-Un-distrib2*: $(SIGMA\ i{:}I.\ A(i)\ Un\ B(i)) = (SIGMA\ i{:}I.\ A(i))\ Un\ (SIGMA\ i{:}I.\ B(i))$
by *blast*

lemma *Sigma-Int-distrib1*: $(SIGMA\ i{:}I\ Int\ J.\ C(i)) = (SIGMA\ i{:}I.\ C(i))\ Int\ (SIGMA\ j{:}J.\ C(j))$
by *blast*

lemma *Sigma-Int-distrib2*: $(SIGMA\ i{:}I.\ A(i)\ Int\ B(i)) = (SIGMA\ i{:}I.\ A(i))\ Int\ (SIGMA\ i{:}I.\ B(i))$
by *blast*

lemma *Sigma-Diff-distrib1*: $(SIGMA\ i{:}I - J.\ C(i)) = (SIGMA\ i{:}I.\ C(i)) - (SIGMA\ j{:}J.\ C(j))$
by *blast*

lemma *Sigma-Diff-distrib2*: $(SIGMA\ i{:}I.\ A(i) - B(i)) = (SIGMA\ i{:}I.\ A(i)) - (SIGMA\ i{:}I.\ B(i))$
by *blast*

lemma *Sigma-Union*: $Sigma\ (Union\ X)\ B = (UN\ A{:}X.\ Sigma\ A\ B)$
by *blast*

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: $(A\ Un\ B) <*> C = (A <*> C)\ Un\ (B <*> C)$
by *blast*

lemma *Times-Int-distrib1*: $(A\ Int\ B) <*> C = (A <*> C)\ Int\ (B <*> C)$
by *blast*

lemma *Times-Diff-distrib1*: $(A - B) <*> C = (A <*> C) - (B <*> C)$
by *blast*

lemma *Times-empty*[*simp*]: $A \times B = \{\} \longleftrightarrow A = \{\} \vee B = \{\}$
by *auto*

lemma *fst-image-times*[*simp*]: $fst\ `\ (A \times B) = (if\ B = \{\}\ then\ \{\}\ else\ A)$
by (*auto intro*!: *image-eqI*)

lemma *snd-image-times*[*simp*]: $snd\ `\ (A \times B) = (if\ A = \{\}\ then\ \{\}\ else\ B)$
by (*auto intro*!: *image-eqI*)

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*

lemma *vimage-Times*: $f\ -`\ (A \times B) = ((fst \circ f)\ -`\ A) \cap ((snd \circ f)\ -`\ B)$
by (*auto*, *case-tac f x*, *auto*)

The following *prod-fun* lemmas are due to Joachim Breitner:

lemma *prod-fun-inj-on*:
assumes *inj-on f A* **and** *inj-on g B*
shows *inj-on* (*prod-fun f g*) $(A \times B)$
proof (*rule inj-onI*)

fix $x :: 'a \times 'c$ **and** $y :: 'a \times 'c$
assume $x \in A \times B$ **hence** *fst* $x \in A$ **and** *snd* $x \in B$ **by** *auto*
assume $y \in A \times B$ **hence** *fst* $y \in A$ **and** *snd* $y \in B$ **by** *auto*
assume *prod-fun f g x* = *prod-fun f g y*
hence *fst* (*prod-fun f g x*) = *fst* (*prod-fun f g y*) **by** (*auto*)
hence *f* (*fst x*) = *f* (*fst y*) **by** (*cases x*,*cases y*,*auto*)
with ‹*inj-on f A*› **and** ‹*fst x* ∈ *A*› **and** ‹*fst y* ∈ *A*›
have *fst x* = *fst y* **by** (*auto dest*:*dest*:*inj-onD*)
moreover from ‹*prod-fun f g x* = *prod-fun f g y*›
have *snd* (*prod-fun f g x*) = *snd* (*prod-fun f g y*) **by** (*auto*)
hence *g* (*snd x*) = *g* (*snd y*) **by** (*cases x*,*cases y*,*auto*)
with ‹*inj-on g B*› **and** ‹*snd x* ∈ *B*› **and** ‹*snd y* ∈ *B*›
have *snd x* = *snd y* **by** (*auto dest*:*dest*:*inj-onD*)
ultimately show *x* = *y* **by**(*rule prod-eqI*)
qed

lemma *prod-fun-surj*:
assumes *surj f* **and** *surj g*
shows *surj* (*prod-fun f g*)
unfolding *surj-def*
proof
fix $y :: 'b \times 'd$
from ‹*surj f*› **obtain** *a* **where** *fst y* = *f a* **by** (*auto elim*:*surjE*)
moreover
from ‹*surj g*› **obtain** *b* **where** *snd y* = *g b* **by** (*auto elim*:*surjE*)
ultimately have (*fst y*, *snd y*) = *prod-fun f g* (*a*,*b*) **by** *auto*
thus ∃ *x*. *y* = *prod-fun f g x* **by** *auto*
qed

lemma *prod-fun-surj-on*:
assumes *f* ‘ *A* = *A'* **and** *g* ‘ *B* = *B'*
shows *prod-fun f g* ‘ (*A* × *B*) = *A'* × *B'*
unfolding *image-def*
proof(*rule set-ext*,*rule iffI*)
fix $x :: 'a \times 'c$
assume $x \in \{y::'a \times 'c.\ \exists x::'b \times 'd \in A \times B.\ y = prod\text{-}fun\ f\ g\ x\}$
then obtain *y* **where** *y* ∈ *A* × *B* **and** *x* = *prod-fun f g y* **by** *blast*
from ‹*image f A* = *A'*› **and** ‹*y* ∈ *A* × *B*› **have** *f* (*fst y*) ∈ *A'* **by** *auto*
moreover from ‹*image g B* = *B'*› **and** ‹*y* ∈ *A* × *B*› **have** *g* (*snd y*) ∈ *B'* **by** *auto*
ultimately have (*f* (*fst y*), *g* (*snd y*)) ∈ (*A'* × *B'*) **by** *auto*
with ‹*x* = *prod-fun f g y*› **show** *x* ∈ *A'* × *B'* **by** (*cases y*, *auto*)
next
fix $x :: 'a \times 'c$
assume *x* ∈ *A'* × *B'* **hence** *fst x* ∈ *A'* **and** *snd x* ∈ *B'* **by** *auto*
from ‹*image f A* = *A'*› **and** ‹*fst x* ∈ *A'*› **have** *fst x* ∈ *image f A* **by** *auto*
then obtain *a* **where** *a* ∈ *A* **and** *fst x* = *f a* **by** (*rule imageE*)
moreover from ‹*image g B* = *B'*› **and** ‹*snd x* ∈ *B'*›
obtain *b* **where** *b* ∈ *B* **and** *snd x* = *g b* **by** *auto*

ultimately have (*fst x*, *snd x*) = *prod-fun f g* (*a*,*b*) **by** *auto*
moreover from ‹*a* ∈ *A*› **and** ‹*b* ∈ *B*› **have** (*a* , *b*) ∈ *A* × *B* **by** *auto*
ultimately have ∃ *y* ∈ *A* × *B*. *x* = *prod-fun f g y* **by** *auto*
thus *x* ∈ {*x*. ∃ *y* ∈ *A* × *B*. *x* = *prod-fun f g y*} **by** *auto*
qed

lemma *swap-inj-on*:
inj-on (λ(*i*, *j*). (*j*, *i*)) *A*
by (*auto intro*!: *inj-onI*)

lemma *swap-product*:
(%(*i*, *j*). (*j*, *i*)) ‘ (*A* × *B*) = *B* × *A*
by (*simp add*: *split-def image-def*) *blast*

lemma *image-split-eq-Sigma*:
(λ*x*. (*f x*, *g x*)) ‘ *A* = *Sigma* (*f* ‘ *A*) (λ*x*. *g* ‘ (*f* −‘ {*x*} ∩ *A*))
proof (*safe intro*!: *imageI vimageI*)
fix *a b* **assume** ∗: *a* ∈ *A b* ∈ *A* **and** *eq*: *f a* = *f b*
show (*f b*, *g a*) ∈ (λ*x*. (*f x*, *g x*)) ‘ *A*
using ∗ *eq*[*symmetric*] **by** *auto*
qed *simp-all*

11.4 Inductively defined sets

use *Tools/inductive-set.ML*
setup *Inductive-Set.setup*

11.5 Legacy theorem bindings and duplicates

lemma *PairE*:
obtains *x y* **where** *p* = (*x*, *y*)
by (*fact prod.exhaust*)

lemma *Pair-inject*:
assumes (*a*, *b*) = (*a*′, *b*′)
and *a* = *a*′ ==> *b* = *b*′ ==> *R*
shows *R*
using *assms* **by** *simp*

lemmas *Pair-eq* = *prod.inject*

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

end

12 Sum-Type: The Disjoint Sum of Two Types

theory *Sum-Type*

imports *Typedef Inductive Fun*
begin

12.1 Construction of the sum type and its basic abstract operations

definition *Inl-Rep* :: $'a \Rightarrow 'a \Rightarrow 'b \Rightarrow bool \Rightarrow bool$ **where**
Inl-Rep a x y p $\longleftrightarrow x = a \wedge p$

definition *Inr-Rep* :: $'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool \Rightarrow bool$ **where**
Inr-Rep b x y p $\longleftrightarrow y = b \wedge \neg p$

global

typedef (*Sum*) $('a, 'b)$ + (**infixr** + *10*) = $\{f.\ (\exists\, a.\ f = \textit{Inl-Rep}\ (a::'a)) \vee (\exists\, b.\ f = \textit{Inr-Rep}\ (b::'b))\}$
by *auto*

local

lemma *Inl-RepI*: *Inl-Rep a* $\in$ *Sum*
by (*auto simp add*: *Sum-def*)

lemma *Inr-RepI*: *Inr-Rep b* $\in$ *Sum*
by (*auto simp add*: *Sum-def*)

lemma *inj-on-Abs-Sum*: *A* $\subseteq$ *Sum* $\Longrightarrow$ *inj-on Abs-Sum A*
by (*rule inj-on-inverseI*, *rule Abs-Sum-inverse*) *auto*

lemma *Inl-Rep-inject*: *inj-on Inl-Rep A*
proof (*rule inj-onI*)
show $\bigwedge a\ c.$ *Inl-Rep a* = *Inl-Rep c* $\Longrightarrow$ *a* = *c*
by (*auto simp add*: *Inl-Rep-def expand-fun-eq*)
qed

lemma *Inr-Rep-inject*: *inj-on Inr-Rep A*
proof (*rule inj-onI*)
show $\bigwedge b\ d.$ *Inr-Rep b* = *Inr-Rep d* $\Longrightarrow$ *b* = *d*
by (*auto simp add*: *Inr-Rep-def expand-fun-eq*)
qed

lemma *Inl-Rep-not-Inr-Rep*: *Inl-Rep a* $\neq$ *Inr-Rep b*
by (*auto simp add*: *Inl-Rep-def Inr-Rep-def expand-fun-eq*)

definition *Inl* :: $'a \Rightarrow 'a + 'b$ **where**
Inl = *Abs-Sum* $\circ$ *Inl-Rep*

definition *Inr* :: $'b \Rightarrow 'a + 'b$ **where**
Inr = *Abs-Sum* $\circ$ *Inr-Rep*

lemma *inj-Inl* [*simp*]: *inj-on Inl A*
by (*auto simp add*: *Inl-def intro*!: *comp-inj-on Inl-Rep-inject inj-on-Abs-Sum Inl-RepI*)

lemma *Inl-inject*: *Inl x* = *Inl y* ⟹ *x* = *y*
using *inj-Inl* **by** (*rule injD*)

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
by (*auto simp add*: *Inr-def intro*!: *comp-inj-on Inr-Rep-inject inj-on-Abs-Sum Inr-RepI*)

lemma *Inr-inject*: *Inr x* = *Inr y* ⟹ *x* = *y*
using *inj-Inr* **by** (*rule injD*)

lemma *Inl-not-Inr*: *Inl a* ≠ *Inr b*
proof −
  from *Inl-RepI* [*of a*] *Inr-RepI* [*of b*] **have** {*Inl-Rep a*, *Inr-Rep b*} ⊆ *Sum* **by** *auto*
  with *inj-on-Abs-Sum* **have** *inj-on Abs-Sum* {*Inl-Rep a*, *Inr-Rep b*} **.**
  with *Inl-Rep-not-Inr-Rep* [*of a b*] *inj-on-contraD* **have** *Abs-Sum* (*Inl-Rep a*) ≠ *Abs-Sum* (*Inr-Rep b*) **by** *auto*
  then show *?thesis* **by** (*simp add*: *Inl-def Inr-def*)
qed

lemma *Inr-not-Inl*: *Inr b* ≠ *Inl a*
  using *Inl-not-Inr* **by** (*rule not-sym*)

lemma *sumE*:
  assumes ⋀*x*::′*a*. *s* = *Inl x* ⟹ *P*
    and ⋀*y*::′*b*. *s* = *Inr y* ⟹ *P*
  shows *P*
proof (*rule Abs-Sum-cases* [*of s*])
  fix *f*
  assume *s* = *Abs-Sum f* **and** *f* ∈ *Sum*
  with *assms* **show** *P* **by** (*auto simp add*: *Sum-def Inl-def Inr-def*)
qed

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 (*auto dest*: *Inl-inject Inr-inject simp add*: *Inl-not-Inr*)

12.2 Projections

lemma *sum-case-KK* [*simp*]: *sum-case* (λ*x*. *a*) (λ*x*. *a*) = (λ*x*. *a*)
  by (*rule ext*) (*simp split*: *sum.split*)

lemma *surjective-sum*: *sum-case* ($\lambda x{::}'a.\ f\ (Inl\ x)$) ($\lambda y{::}'b.\ f\ (Inr\ y)$) = *f*
proof
  fix *s* :: $'a + 'b$
  show (*case s of Inl* ($x{::}'a$) $\Rightarrow$ *f* (*Inl x*) | *Inr* ($y{::}'b$) $\Rightarrow$ *f* (*Inr y*)) = *f s*
    by (*cases s*) *simp-all*
qed

lemma *sum-case-inject*:
  assumes *a*: *sum-case f1 f2* = *sum-case g1 g2*
  assumes *r*: *f1* = *g1* $\Longrightarrow$ *f2* = *g2* $\Longrightarrow$ *P*
  shows *P*
proof (*rule r*)
  show *f1* = *g1* **proof**
    fix *x* :: $'a$
    from *a* **have** *sum-case f1 f2* (*Inl x*) = *sum-case g1 g2* (*Inl x*) **by** *simp*
    then show *f1 x* = *g1 x* **by** *simp*
  qed
  show *f2* = *g2* **proof**
    fix *y* :: $'b$
    from *a* **have** *sum-case f1 f2* (*Inr y*) = *sum-case g1 g2* (*Inr y*) **by** *simp*
    then show *f2 y* = *g2 y* **by** *simp*
  qed
qed

lemma *sum-case-weak-cong*:
  s = *t* $\Longrightarrow$ *sum-case f g s* = *sum-case f g t*
  — Prevents simplification of *f* and *g*: much faster.
  by *simp*

primrec *Projl* :: $'a + 'b \Rightarrow 'a$ **where**
  Projl-Inl: *Projl* (*Inl x*) = *x*

primrec *Projr* :: $'a + 'b \Rightarrow 'b$ **where**
  Projr-Inr: *Projr* (*Inr x*) = *x*

primrec *Suml* :: $('a \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$ **where**
  Suml f (*Inl x*) = *f x*

primrec *Sumr* :: $('b \Rightarrow 'c) \Rightarrow 'a + 'b \Rightarrow 'c$ **where**
  Sumr f (*Inr x*) = *f x*

lemma *Suml-inject*:
  assumes *Suml f* = *Suml g* **shows** *f* = *g*
proof
  fix *x* :: $'a$
  let *?s* = *Inl x* :: $'a + 'b$
  from *assms* **have** *Suml f ?s* = *Suml g ?s* **by** *simp*
  then show *f x* = *g x* **by** *simp*
qed

lemma *Sumr-inject*:
assumes *Sumr f* = *Sumr g* **shows** *f* = *g*
proof
fix *x* :: *'b*
let *?s* = *Inr x* :: *'a* + *'b*
from *assms* **have** *Sumr f ?s* = *Sumr g ?s* **by** *simp*
then show *f x* = *g x* **by** *simp*
qed

12.3 The Disjoint Sum of Sets

definition *Plus* :: *'a set* ⇒ *'b set* ⇒ (*'a* + *'b*) *set* (**infixr** <+> *65*) **where**
A <+> *B* = *Inl* ‘ *A* ∪ *Inr* ‘ *B*

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

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

Exhaustion rule for sums, a degenerate form of induction

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

lemma *Plus-eq-empty-conv* [*simp*]: *A* <+> *B* = {} ⟷ *A* = {} ∧ *B* = {}
by *auto*

lemma *UNIV-Plus-UNIV* [*simp*]: *UNIV* <+> *UNIV* = *UNIV*
proof (*rule set-ext*)
fix *u* :: *'a* + *'b*
show *u* ∈ *UNIV* <+> *UNIV* ⟷ *u* ∈ *UNIV* **by** (*cases u*) *auto*
qed

hide-const (**open**) *Suml Sumr Projl Projr*

end

13 Rings: Rings

theory *Rings*
imports *Groups*
begin

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +
assumes *left-distrib*[*algebra-simps*, *field-simps*]: (*a* + *b*) ∗ *c* = *a* ∗ *c* + *b* ∗ *c*

assumes *right-distrib*[*algebra-simps*, *field-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* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infixl** *dvd* 50) **where**
  [*code del*]: b *dvd* $a \longleftrightarrow (\exists k.\ a = b * k)$

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* [*no-atp*, *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* $c \Longrightarrow a$ *dvd* c
by (*simp add*: *dvd-def mult-assoc*, *blast*)

lemma *dvd-mult-right*: $a * b$ *dvd* $c \Longrightarrow b$ *dvd* c
unfolding *mult-ac* [*of* a] **by** (*rule dvd-mult-left*)

lemma *dvd-0-left*: *0 dvd* $a \Longrightarrow a = 0$
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*: $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow a * b \neq 0$
begin

lemma *divisors-zero*:
assumes $a * b = 0$
shows $a = 0 \vee b = 0$
proof (*rule classical*)
assume $\neg (a = 0 \vee b = 0)$
then have $a \neq 0$ **and** $b \neq 0$ **by** *auto*
with *no-zero-divisors* **have** $a * b \neq 0$ **by** *blast*
with *assms* **show** *?thesis* **by** *simp*
qed

end

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 minus-unique*) (*simp add*: *left-distrib* [*symmetric*])

lemma *minus-mult-right*: $-(a * b) = a * -b$
by (*rule minus-unique*) (*simp add*: *right-distrib* [*symmetric*])

Extract signs from products

lemmas *mult-minus-left* [*simp*, *no-atp*] = *minus-mult-left* [*symmetric*]
lemmas *mult-minus-right* [*simp*,*no-atp*] = *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*, *field-simps*]: $a * (b - c) = a * b - a * c$
by (*simp add*: *right-distrib diff-minus*)

lemma *left-diff-distrib*[*algebra-simps*, *field-simps*]: $(a - b) * c = a * c - b * c$
by (*simp add*: *left-distrib diff-minus*)

lemmas *ring-distribs*[*no-atp*] =
right-distrib left-distrib left-diff-distrib right-diff-distrib

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*[*no-atp*] =
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 only*: *diff-minus dvd-add dvd-minus-iff*)

end

class *ring-no-zero-divisors* = *ring* + *no-zero-divisors*
begin

lemma *mult-eq-0-iff* [*simp*]:
shows *a* ∗ *b* = *0* ⟷ (*a* = *0* ∨ *b* = *0*)
proof (*cases a* = *0* ∨ *b* = *0*)
case *False* **then have** *a* ≠ *0* **and** *b* ≠ *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*, *no-atp*]:
a ∗ *c* = *b* ∗ *c* ⟷ *c* = *0* ∨ *a* = *b*
proof −
have (*a* ∗ *c* = *b* ∗ *c*) = ((*a* − *b*) ∗ *c* = *0*)
by (*simp add*: *algebra-simps*)
thus *?thesis* **by** (*simp add*: *disj-commute*)
qed

lemma *mult-cancel-left* [*simp*, *no-atp*]:
c ∗ *a* = *c* ∗ *b* ⟷ *c* = *0* ∨ *a* = *b*
proof −
have (*c* ∗ *a* = *c* ∗ *b*) = (*c* ∗ (*a* − *b*) = *0*)
by (*simp add*: *algebra-simps*)
thus *?thesis* **by** *simp*
qed

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

lemma *square-eq-1-iff*:
x ∗ *x* = *1* ⟷ *x* = *1* ∨ *x* = − *1*
proof −
have (*x* − *1*) ∗ (*x* + *1*) = *x* ∗ *x* − *1*
by (*simp add*: *algebra-simps*)
hence *x* ∗ *x* = *1* ⟷ (*x* − *1*) ∗ (*x* + *1*) = *0*
by *simp*
thus *?thesis*
by (*simp add*: *eq-neg-iff-add-eq-0*)
qed

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*: *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 ⟷ (∃k. b * c = (a * k) * c)
    unfolding dvd-def by (simp add: mult-ac)
  also have (∃k. b * c = (a * k) * c) ⟷ c = 0 ∨ a dvd b
    unfolding dvd-def by simp
  finally show ?thesis .
qed

end

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

class division-ring = ring-1 + inverse +
  assumes left-inverse [simp]:  a ≠ 0 ⟹ inverse a * a = 1
  assumes right-inverse [simp]: a ≠ 0 ⟹ a * inverse a = 1
  assumes divide-inverse: a / b = a * inverse b
begin

subclass ring-1-no-zero-divisors
proof
  fix a b :: 'a
  assume a: a ≠ 0 and b: b ≠ 0
  show a * b ≠ 0
  proof
    assume ab: a * b = 0
    hence 0 = inverse a * (a * b) * inverse b by simp
    also have ... = (inverse a * a) * (b * inverse b)
      by (simp only: mult-assoc)
    also have ... = 1 using a b by simp
    finally show False by simp
  qed
qed

lemma nonzero-imp-inverse-nonzero:
  a ≠ 0 ⟹ inverse a ≠ 0
proof
  assume ianz: inverse a = 0
  assume a ≠ 0
  hence 1 = a * inverse a by simp
  also have ... = 0 by (simp add: ianz)
  finally have 1 = 0 .
  thus False by (simp add: eq-commute)
qed

lemma inverse-zero-imp-zero:
  inverse a = 0 ⟹ a = 0
apply (rule classical)
apply (drule nonzero-imp-inverse-nonzero)
```

apply *auto*
done

lemma *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*)

lemma *right-inverse-eq*: $b \neq 0 \Longrightarrow a \mathbin{/} b = 1 \longleftrightarrow a = b$
proof
  assume *neq*: $b \neq 0$
  {
    hence $a = (a \mathbin{/} b) * b$ **by** (*simp add*: *divide-inverse mult-assoc*)
    also assume $a \mathbin{/} b = 1$
    finally show $a = b$ **by** *simp*
  next
    assume $a = b$
    with *neq* **show** $a \mathbin{/} b = 1$ **by** (*simp add*: *divide-inverse*)
  }
qed

lemma *nonzero-inverse-eq-divide*: $a \neq 0 \Longrightarrow \mathit{inverse}\ a = 1 \mathbin{/} a$
by (*simp add*: *divide-inverse*)

lemma *divide-self* [*simp*]: $a \neq 0 \Longrightarrow a \mathbin{/} a = 1$
by (*simp add*: *divide-inverse*)

lemma *divide-zero-left* [*simp*]: $0 \mathbin{/} a = 0$
by (*simp add*: *divide-inverse*)

lemma *inverse-eq-divide*: $\mathit{inverse}\ a = 1 \mathbin{/} a$
by (*simp add*: *divide-inverse*)

lemma *add-divide-distrib*: $(a+b) \mathbin{/} c = a/c + b/c$
by (*simp add*: *divide-inverse algebra-simps*)

lemma *divide-1* [*simp*]: $a \mathbin{/} 1 = a$
  by (*simp add*: *divide-inverse*)

lemma *times-divide-eq-right* [*simp*]: $a * (b \mathbin{/} c) = (a * b) \mathbin{/} c$
  by (*simp add*: *divide-inverse mult-assoc*)

lemma *minus-divide-left*: $-\ (a \mathbin{/} b) = (-a) \mathbin{/} b$
  by (*simp add*: *divide-inverse*)

lemma *nonzero-minus-divide-right*: $b \neq 0 \Longrightarrow -\ (a \mathbin{/} b) = a \mathbin{/} (-\ b)$
  by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *nonzero-minus-divide-divide*: $b \neq 0 \Longrightarrow (-a) \mathbin{/} (-b) = a \mathbin{/} b$
  by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *divide-minus-left* [*simp*, *no-atp*]: $(-a) \mathbin{/} b = -\ (a \mathbin{/} b)$
  by (*simp add*: *divide-inverse*)

lemma *diff-divide-distrib*: $(a - b) \mathbin{/} c = a \mathbin{/} c - b \mathbin{/} c$
  by (*simp add*: *diff-minus add-divide-distrib*)

lemma *nonzero-eq-divide-eq* [*field-simps*]: $c \neq 0 \Longrightarrow a = b \,/\, c \longleftrightarrow a * c = b$
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* [*field-simps*]: $c \neq 0 \Longrightarrow b \,/\, c = a \longleftrightarrow b = a * c$
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*: $c \neq 0 \Longrightarrow b = a * c \Longrightarrow b \,/\, c = a$
  by (*simp add*: *divide-inverse mult-assoc*)

lemma *eq-divide-imp*: $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b \,/\, c$
  by (*drule sym*) (*simp add*: *divide-inverse mult-assoc*)

end

class *division-ring-inverse-zero* = *division-ring* +
  assumes *inverse-zero* [*simp*]: *inverse 0* = *0*
begin

lemma *divide-zero* [*simp*]:
  $a \,/\, 0 = 0$
  by (*simp add*: *divide-inverse*)

lemma *divide-self-if* [*simp*]:
  $a \,/\, a = (\textit{if } a = 0 \textit{ then } 0 \textit{ else } 1)$
  by *simp*

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
  inverse a = *0* $\longleftrightarrow$ *a* = *0*
  by *rule* (*fact inverse-zero-imp-zero*, *simp*)

lemma *inverse-minus-eq* [*simp*]:
  inverse (− *a*) = − *inverse a*
proof *cases*
  assume *a=0* **thus** *?thesis* **by** *simp*
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
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
by (*force dest*!: *inverse-eq-imp-eq*)

lemma *inverse-inverse-eq* [*simp*]:
inverse (*inverse a*) = *a*
proof *cases*
assume *a=0* **thus** *?thesis* **by** *simp*
next
assume *a≠0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

end

class *mult-mono* = *times* + *zero* + *ord* +
assumes *mult-left-mono*: *a ≤ b ⟹ 0 ≤ c ⟹ c * a ≤ c * b*
assumes *mult-right-mono*: *a ≤ b ⟹ 0 ≤ c ⟹ a * c ≤ b * c*

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 *ordered-semiring* = *mult-mono* + *semiring-0* + *ordered-ab-semigroup-add*
begin

lemma *mult-mono*:
a ≤ b ⟹ c ≤ d ⟹ 0 ≤ b ⟹ 0 ≤ c
*⟹ a * c ≤ 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 *ordered-cancel-semiring* = *mult-mono* + *ordered-ab-semigroup-add*
+ *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* ..
subclass *ordered-semiring* ..

lemma *mult-nonneg-nonneg*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * b$
using *mult-left-mono* [*of 0 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 0 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 0 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 0*], *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 *linordered-semiring* = *semiring* + *comm-monoid-add* + *linordered-cancel-ab-semigroup-add*
+ *mult-mono*
begin

subclass *ordered-cancel-semiring* ..

subclass *ordered-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 *linordered-semiring-1* = *linordered-semiring* + *semiring-1*
begin

lemma *convex-bound-le*:
  assumes $x \leq a$ $y \leq a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
  shows $u * x + v * y \leq a$
proof−
  from *assms* **have** $u * x + v * y \leq u * a + v * a$
    by (*simp add*: *add-mono mult-left-mono*)
  thus *?thesis* **using** *assms* **unfolding** *left-distrib*[*symmetric*] **by** *simp*
qed

end

class *linordered-semiring-strict* = *semiring* + *comm-monoid-add* + *linordered-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 *linordered-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 0 b a*] **by** *simp*

lemma *mult-pos-neg*: $0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$
using *mult-strict-left-mono* [*of b 0 a*] **by** *simp*

lemma *mult-neg-pos*: $a < 0 \Longrightarrow 0 < b \Longrightarrow a * b < 0$
using *mult-strict-right-mono* [*of a 0 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 0*], *auto*)

lemma *zero-less-mult-pos*:
$0 < a * b \Longrightarrow 0 < a \Longrightarrow 0 < b$
apply (*cases* $b \leq 0$)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

lemma *zero-less-mult-pos2*:
$0 < b * a \Longrightarrow 0 < a \Longrightarrow 0 < b$
apply (*cases* $b \leq 0$)
apply (*auto simp add*: *le-less not-less*)
apply (*drule-tac mult-pos-neg2* [*of a b*])
apply (*auto dest*: *less-not-sym*)
done

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 *linordered-semiring-1-strict* = *linordered-semiring-strict* + *semiring-1*
begin

subclass *linordered-semiring-1* **..**

lemma *convex-bound-lt*:
assumes $x < a$ $y < a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
shows $u * x + v * y < a$
proof −
from *assms* **have** $u * x + v * y < u * a + v * a$
by (*cases* $u = 0$)

(*auto intro*!: *add-less-le-mono mult-strict-left-mono mult-left-mono*)
thus *?thesis* **using** *assms* **unfolding** *left-distrib*[*symmetric*] **by** *simp*
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 *ordered-comm-semiring* = *comm-semiring-0*
+ *ordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *ordered-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 *ordered-cancel-comm-semiring* = *comm-semiring-0-cancel*
+ *ordered-ab-semigroup-add* + *mult-mono1*
begin

subclass *ordered-comm-semiring* **..**
subclass *ordered-cancel-semiring* **..**

end

class *linordered-comm-semiring-strict* = *comm-semiring-0* + *linordered-cancel-ab-semigroup-add*
+
assumes *mult-strict-left-mono-comm*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *linordered-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 *ordered-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 *ordered-ring* = *ring* + *ordered-cancel-semiring*
begin

subclass *ordered-ab-group-add* **..**

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* - - − *c*])
apply *simp-all*
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* - - − *c*])
apply *simp-all*
done

lemma *mult-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a * b$
using *mult-right-mono-neg* [*of a 0 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 *linordered-ring* = *ring* + *linordered-semiring* + *linordered-ab-group-add* + *abs-if*
begin

subclass *ordered-ring* **..**

subclass *ordered-ab-group-add-abs*
proof
  fix *a b*
  show $|a + b| \leq |a| + |b|$
    by (*auto simp add*: *abs-if not-less*)
    (*auto simp del*: *minus-add-distrib simp add*: *minus-add-distrib* [*symmetric*],
     auto intro!: *less-imp-le add-neg-neg*)
qed (*auto simp add*: *abs-if*)

lemma *zero-le-square* [*simp*]: $0 \leq a * a$
  using *linear* [*of 0 a*]
  by (*auto simp add*: *mult-nonneg-nonneg mult-nonpos-nonpos*)

lemma *not-square-less-zero* [*simp*]: $\neg (a * a < 0)$
  by (*simp add*: *not-less*)

end

class *linordered-ring-strict* = *ring* + *linordered-semiring-strict*
  + *ordered-ab-group-add* + *abs-if*
begin

subclass *linordered-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 0 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
  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
  done
```

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations

$\leq$ and equality.

These "disjunction" versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

lemma *mult-less-cancel-right-disj*:
$a * c < b * c \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
apply (*cases* $c = 0$)
apply (*auto simp add*: *neq-iff mult-strict-right-mono*
mult-strict-right-mono-neg)
apply (*auto simp add*: *not-less*
not-le [*symmetric*, *of* $a*c$]
not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *less-imp-le mult-right-mono*
mult-right-mono-neg)
done

lemma *mult-less-cancel-left-disj*:
$c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
apply (*cases* $c = 0$)
apply (*auto simp add*: *neq-iff mult-strict-left-mono*
mult-strict-left-mono-neg)
apply (*auto simp add*: *not-less*
not-le [*symmetric*, *of* $c*a$]
not-le [*symmetric*, *of* a])
apply (*erule-tac* [!] *notE*)
apply (*auto simp add*: *less-imp-le mult-left-mono*
mult-left-mono-neg)
done

The "conjunction of implication" lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma *mult-less-cancel-right*:
$a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-right-disj* [*of* a c b] **by** *auto*

lemma *mult-less-cancel-left*:
$c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-left-disj* [*of* c a b] **by** *auto*

lemma *mult-le-cancel-right*:
$a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-right-disj*)

lemma *mult-le-cancel-left*:
$c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-left-disj*)

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

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 *ordered-comm-ring* = *comm-ring* + *ordered-comm-semiring*
begin

subclass *ordered-ring* ..
subclass *ordered-cancel-comm-semiring* ..

end

class *linordered-semidom* = *comm-semiring-1-cancel* + *linordered-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 0 a b c*] **by** *simp*

lemma *zero-le-one* [*simp*]: $0 \leq 1$
by (*rule zero-less-one* [*THEN less-imp-le*])

lemma *not-one-le-zero* [*simp*]: $\neg\ 1 \leq 0$
by (*simp add*: *not-le*)

lemma *not-one-less-zero* [*simp*]: $\neg\ 1 < 0$
by (*simp add*: *not-less*)

lemma *less-1-mult*:
assumes $1 < m$ **and** $1 < n$
shows $1 < m * n$
using *assms mult-strict-mono* [*of 1 m 1 n*]
by (*simp add*: *less-trans* [*OF zero-less-one*])

end

class *linordered-idom* = *comm-ring-1* +
linordered-comm-semiring-strict + *ordered-ab-group-add* +
abs-if + *sgn-if*

begin

subclass *linordered-semiring-1-strict* **..**
subclass *linordered-ring-strict* **..**
subclass *ordered-comm-ring* **..**
subclass *idom* **..**

subclass *linordered-semidom*
proof
have $0 \leq 1 * 1$ **by** (*rule zero-le-square*)
thus $0 < 1$ **by** (*simp add*: *le-less*)
qed

lemma *linorder-neqE-linordered-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*

lemma *sgn-1-neg*:
$sgn\ a = -\ 1 \longleftrightarrow a < 0$
unfolding *sgn-if* **by** *auto*

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*: $|k| = k * sgn\ k$
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 ⟷ a < 0
  unfolding *sgn-if* **by** *auto*

lemma *abs-dvd-iff* [*simp*]: *|m| dvd k ⟷ m dvd k*
  by (*simp add*: *abs-if*)

lemma *dvd-abs-iff* [*simp*]: *m dvd |k| ⟷ m dvd k*
  by (*simp add*: *abs-if*)

lemma *dvd-if-abs-eq*:
  |l| = |k| ⟹ l dvd k
by(*subst abs-dvd-iff*[*symmetric*]) *simp*

end

Simprules for comparisons where common factors can be cancelled.

lemmas *mult-compare-simps*[*no-atp*] =
    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

Reasoning about inequalities with division

context *linordered-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

context *linordered-idom*
begin

lemma *mult-right-le-one-le*:

$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow y \leq 1 \Longrightarrow x * y \leq x$
by (*auto simp add*: *mult-le-cancel-left2*)

lemma *mult-left-le-one-le*:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow y \leq 1 \Longrightarrow y * x \leq x$
by (*auto simp add*: *mult-le-cancel-right2*)

end

Absolute Value

context *linordered-idom*
begin

lemma *mult-sgn-abs*:
sgn $x * |x| = x$
unfolding *abs-if sgn-if* **by** *auto*

lemma *abs-one* [*simp*]:
$|1| = 1$
by (*simp add*: *abs-if zero-less-one* [*THEN less-not-sym*])

end

class *ordered-ring-abs* = *ordered-ring* + *ordered-ab-group-add-abs* +
assumes *abs-eq-mult*:
$(0 \leq a \vee a \leq 0) \wedge (0 \leq b \vee b \leq 0) \Longrightarrow |a * b| = |a| * |b|$

context *linordered-idom*
begin

subclass *ordered-ring-abs* **proof**
qed (*auto simp add*: *abs-if not-less mult-less-0-iff*)

lemma *abs-mult*:
$|a * b| = |a| * |b|$
by (*rule abs-eq-mult*) *auto*

lemma *abs-mult-self*:
$|a| * |a| = a * a$
by (*simp add*: *abs-if*)

lemma *abs-mult-less*:
$|a| < c \Longrightarrow |b| < d \Longrightarrow |a| * |b| < c * d$
proof −
assume *ac*: $|a| < c$
hence *cpos*: $0<c$ **by** (*blast intro*: *le-less-trans abs-ge-zero*)
assume $|b| < d$
thus *?thesis* **by** (*simp add*: *ac cpos mult-strict-mono*)
qed

lemma *less-minus-self-iff*:
  $a < -a \longleftrightarrow a < 0$
  by (*simp only*: *less-le less-eq-neg-nonpos equal-neg-zero*)

lemma *abs-less-iff*:
  $|a| < b \longleftrightarrow a < b \wedge -a < b$
  by (*simp add*: *less-le abs-le-iff*) (*auto simp add*: *abs-if*)

lemma *abs-mult-pos*:
  $0 \leq x \Longrightarrow |y| * x = |y * x|$
  by (*simp add*: *abs-mult*)

end

code-modulename *SML*
  Rings Arith

code-modulename *OCaml*
  Rings Arith

code-modulename *Haskell*
  Rings Arith

end

14 Fields: Fields

theory *Fields*
imports *Rings*
begin

class *field* = *comm-ring-1* + *inverse* +
  assumes *field-inverse*: $a \neq 0 \Longrightarrow inverse\ a * a = 1$
  assumes *field-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*)
next
  fix $a\ b :: 'a$
  show $a\ /\ b = a * inverse\ b$ **by** (*rule field-divide-inverse*)
qed

subclass *idom* **..**

There is no slick version using division by zero.

lemma *inverse-add*:
[| $a \neq 0$; $b \neq 0$ |]
==> $inverse\ a + inverse\ b = (a + b) * inverse\ a * inverse\ b$
by (*simp add*: *division-ring-inverse-add mult-ac*)

lemma *nonzero-mult-divide-mult-cancel-left* [*simp*, *no-atp*]:
assumes [*simp*]: $b \neq 0$ **and** [*simp*]: $c \neq 0$ **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*, *no-atp*]:
$[\![b \neq 0; c \neq 0]\!] \Longrightarrow (a * c) / (b * c) = a / b$
by (*simp add*: *mult-commute* [*of* - *c*])

lemma *times-divide-eq-left* [*simp*]: $(b / c) * a = (b * a) / c$
by (*simp add*: *divide-inverse mult-ac*)

These are later declared as simp rules.

lemmas *times-divide-eq* [*no-atp*] = *times-divide-eq-right times-divide-eq-left*

lemma *add-frac-eq*:
assumes $y \neq 0$ **and** $z \neq 0$
shows $x / y + w / z = (x * z + w * y) / (y * z)$
proof −
have $x / y + w / z = (x * z) / (y * z) + (y * w) / (y * z)$
using *assms* **by** *simp*
also have ... = $(x * z + y * w) / (y * z)$
by (*simp only*: *add-divide-distrib*)
finally show *?thesis*
by (*simp only*: *mult-commute*)
qed

Special Cancellation Simprules for Division

lemma *nonzero-mult-divide-cancel-right* [*simp*, *no-atp*]:
$b \neq 0 \Longrightarrow a * b / b = a$
using *nonzero-mult-divide-mult-cancel-right* [*of 1 b a*] **by** *simp*

lemma *nonzero-mult-divide-cancel-left* [*simp*, *no-atp*]:
$a \neq 0 \Longrightarrow a * b / a = b$
using *nonzero-mult-divide-mult-cancel-left* [*of 1 a b*] **by** *simp*

lemma *nonzero-divide-mult-cancel-right* [*simp*, *no-atp*]:
⟦*a* ≠ *0*; *b* ≠ *0*⟧ ⟹ *b* / (*a* ∗ *b*) = *1* / *a*
using *nonzero-mult-divide-mult-cancel-right* [*of a b 1*] **by** *simp*

lemma *nonzero-divide-mult-cancel-left* [*simp*, *no-atp*]:
⟦*a* ≠ *0*; *b* ≠ *0*⟧ ⟹ *a* / (*a* ∗ *b*) = *1* / *b*
using *nonzero-mult-divide-mult-cancel-left* [*of b a 1*] **by** *simp*

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*, *no-atp*]:
⟦*b* ≠ *0*; *c* ≠ *0*⟧ ⟹ (*c* ∗ *a*) / (*b* ∗ *c*) = *a* / *b*
using *nonzero-mult-divide-mult-cancel-left* [*of b c a*] **by** (*simp add*: *mult-ac*)

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*, *no-atp*]:
⟦*b* ≠ *0*; *c* ≠ *0*⟧ ⟹ (*a* ∗ *c*) / (*c* ∗ *b*) = *a* / *b*
using *nonzero-mult-divide-mult-cancel-right* [*of b c a*] **by** (*simp add*: *mult-ac*)

lemma *add-divide-eq-iff* [*field-simps*]:
z ≠ *0* ⟹ *x* + *y* / *z* = (*z* ∗ *x* + *y*) / *z*
by (*simp add*: *add-divide-distrib*)

lemma *divide-add-eq-iff* [*field-simps*]:
z ≠ *0* ⟹ *x* / *z* + *y* = (*x* + *z* ∗ *y*) / *z*
by (*simp add*: *add-divide-distrib*)

lemma *diff-divide-eq-iff* [*field-simps*]:
z ≠ *0* ⟹ *x* − *y* / *z* = (*z* ∗ *x* − *y*) / *z*
by (*simp add*: *diff-divide-distrib*)

lemma *divide-diff-eq-iff* [*field-simps*]:
z ≠ *0* ⟹ *x* / *z* − *y* = (*x* − *z* ∗ *y*) / *z*
by (*simp add*: *diff-divide-distrib*)

lemma *diff-frac-eq*:
y ≠ *0* ⟹ *z* ≠ *0* ⟹ *x* / *y* − *w* / *z* = (*x* ∗ *z* − *w* ∗ *y*) / (*y* ∗ *z*)
by (*simp add*: *field-simps*)

lemma *frac-eq-eq*:
y ≠ *0* ⟹ *z* ≠ *0* ⟹ (*x* / *y* = *w* / *z*) = (*x* ∗ *z* = *w* ∗ *y*)
by (*simp add*: *field-simps*)

end

class *field-inverse-zero* = *field* +
assumes *field-inverse-zero*: *inverse 0* = *0*
begin

subclass *division-ring-inverse-zero* **proof**
qed (*fact field-inverse-zero*)

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*
proof *cases*
  assume *a* ≠ *0* & *b* ≠ *0*
  thus *?thesis* **by** (*simp add*: *nonzero-inverse-mult-distrib mult-ac*)
next
  assume ~ (*a* ≠ *0* & *b* ≠ *0*)
  thus *?thesis* **by** *force*
qed

lemma *inverse-divide* [*simp*]:
  inverse (*a* / *b*) = *b* / *a*
  by (*simp add*: *divide-inverse mult-commute*)

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*
apply (*cases b* = *0*)
apply *simp-all*
done

lemma *mult-divide-mult-cancel-right*:
  c ≠ *0* ⟹ (*a* ∗ *c*) / (*b* ∗ *c*) = *a* / *b*
apply (*cases b* = *0*)
apply *simp-all*
done

lemma *divide-divide-eq-right* [*simp*, *no-atp*]:
  a / (*b* / *c*) = (*a* ∗ *c*) / *b*
  by (*simp add*: *divide-inverse mult-ac*)

lemma *divide-divide-eq-left* [*simp*, *no-atp*]:
  (*a* / *b*) / *c* = *a* / (*b* ∗ *c*)
  by (*simp add*: *divide-inverse mult-assoc*)

Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if* [*simp*,*no-atp*]:
  shows (*c* ∗ *a*) / (*c* ∗ *b*) = (*if c* = *0 then 0 else a* / *b*)
  by (*simp add*: *mult-divide-mult-cancel-left*)

Division and Unary Minus

lemma *minus-divide-right*:
  − (*a* / *b*) = *a* / − *b*

by (*simp add*: *divide-inverse*)

lemma *divide-minus-right* [*simp*, *no-atp*]:
a / − *b* = − (*a* / *b*)
by (*simp add*: *divide-inverse*)

lemma *minus-divide-divide*:
(− *a*) / (− *b*) = *a* / *b*
apply (*cases b=0*, *simp*)
apply (*simp add*: *nonzero-minus-divide-divide*)
done

lemma *eq-divide-eq*:
a = *b* / *c* ⟷ (*if c* ≠ *0 then a* ∗ *c* = *b else a* = *0*)
by (*simp add*: *nonzero-eq-divide-eq*)

lemma *divide-eq-eq*:
b / *c* = *a* ⟷ (*if c* ≠ *0 then b* = *a* ∗ *c else a* = *0*)
by (*force simp add*: *nonzero-divide-eq-eq*)

lemma *inverse-eq-1-iff* [*simp*]:
inverse x = *1* ⟷ *x* = *1*
by (*insert inverse-eq-iff-eq* [*of x 1*], *simp*)

lemma *divide-eq-0-iff* [*simp*, *no-atp*]:
a / *b* = *0* ⟷ *a* = *0* ∨ *b* = *0*
by (*simp add*: *divide-inverse*)

lemma *divide-cancel-right* [*simp*, *no-atp*]:
a / *c* = *b* / *c* ⟷ *c* = *0* ∨ *a* = *b*
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)
done

lemma *divide-cancel-left* [*simp*, *no-atp*]:
c / *a* = *c* / *b* ⟷ *c* = *0* ∨ *a* = *b*
apply (*cases c=0*, *simp*)
apply (*simp add*: *divide-inverse*)
done

lemma *divide-eq-1-iff* [*simp*, *no-atp*]:
a / *b* = *1* ⟷ *b* ≠ *0* ∧ *a* = *b*
apply (*cases b=0*, *simp*)
apply (*simp add*: *right-inverse-eq*)
done

lemma *one-eq-divide-iff* [*simp*, *no-atp*]:
1 = *a* / *b* ⟷ *b* ≠ *0* ∧ *a* = *b*
by (*simp add*: *eq-commute* [*of 1*])

lemma *times-divide-times-eq*:
(*x* / *y*) * (*z* / *w*) = (*x* * *z*) / (*y* * *w*)
by *simp*

lemma *add-frac-num*:
y ≠ *0* ⟹ *x* / *y* + *z* = (*x* + *z* * *y*) / *y*
by (*simp add*: *add-divide-distrib*)

lemma *add-num-frac*:
y ≠ *0* ⟹ *z* + *x* / *y* = (*x* + *z* * *y*) / *y*
by (*simp add*: *add-divide-distrib add.commute*)

end

Ordered Fields

class *linordered-field* = *field* + *linordered-idom*
begin

lemma *positive-imp-inverse-positive*:
assumes *a-gt-0*: *0* < *a*
shows *0* < *inverse a*
proof −
have *0* < *a* * *inverse a*
by (*simp add*: *a-gt-0* [*THEN less-imp-not-eq2*])
thus *0* < *inverse a*
by (*simp add*: *a-gt-0* [*THEN less-not-sym*] *zero-less-mult-iff*)
qed

lemma *negative-imp-inverse-negative*:
a < *0* ⟹ *inverse a* < *0*
by (*insert positive-imp-inverse-positive* [*of* −*a*],
simp add: *nonzero-inverse-minus-eq less-imp-not-eq*)

lemma *inverse-le-imp-le*:
assumes *invle*: *inverse a* ≤ *inverse b* **and** *apos*: *0* < *a*
shows *b* ≤ *a*
proof (*rule classical*)
assume ~ *b* ≤ *a*
hence *a* < *b* **by** (*simp add*: *linorder-not-le*)
hence *bpos*: *0* < *b* **by** (*blast intro*: *apos less-trans*)
hence *a* * *inverse a* ≤ *a* * *inverse b*
by (*simp add*: *apos invle less-imp-le mult-left-mono*)
hence (*a* * *inverse a*) * *b* ≤ (*a* * *inverse b*) * *b*
by (*simp add*: *bpos less-imp-le mult-right-mono*)
thus *b* ≤ *a* **by** (*simp add*: *mult-assoc apos bpos less-imp-not-eq2*)
qed

lemma *inverse-positive-imp-positive*:

assumes *inv-gt-0*: $0 < inverse\ a$ **and** *nz*: $a \neq 0$
shows $0 < a$
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-negative-imp-negative*:
assumes *inv-less-0*: $inverse\ a < 0$ **and** *nz*: $a \neq 0$
shows $a < 0$
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 *linordered-field-no-lb*:
$\forall x.\ \exists y.\ y < x$
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 *linordered-field-no-ub*:
$\forall\ x.\ \exists y.\ y > x$
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 *less-imp-inverse-less*:
assumes *less*: $a < b$ **and** *apos*: $0 < a$
shows $inverse\ b < inverse\ a$
proof (*rule ccontr*)
assume $\sim\ inverse\ b < inverse\ a$
hence $inverse\ a \leq inverse\ b$ **by** *simp*
hence $\sim\ (a < b)$
by (*simp add*: *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
apply (*simp add*: *less-le* [*of inverse a*] *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*,*no-atp*]:
0 < a ⟹ 0 < b ⟹ inverse a < inverse b ⟷ b < a
by (*blast intro*: *less-imp-inverse-less dest*: *inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
a ≤ b ⟹ 0 < a ⟹ inverse b ≤ inverse a
by (*force simp add*: *le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le* [*simp*,*no-atp*]:
0 < a ⟹ 0 < b ⟹ inverse a ≤ inverse b ⟷ b ≤ a
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
apply (*rule classical*)
apply (*subgoal-tac a < 0*)
prefer *2* **apply** *force*
apply (*insert inverse-le-imp-le* [*of* −*b* −*a*])
apply (*simp add*: *nonzero-inverse-minus-eq*)
done

lemma *less-imp-inverse-less-neg*:
a < b ⟹ b < 0 ⟹ inverse b < inverse a
apply (*subgoal-tac a < 0*)
prefer *2* **apply** (*blast intro*: *less-trans*)
apply (*insert less-imp-inverse-less* [*of* −*b* −*a*])
apply (*simp add*: *nonzero-inverse-minus-eq*)
done

lemma *inverse-less-imp-less-neg*:
inverse a < inverse b ⟹ b < 0 ⟹ b < a
apply (*rule classical*)
apply (*subgoal-tac a < 0*)
prefer *2*
apply *force*
apply (*insert inverse-less-imp-less* [*of* −*b* −*a*])
apply (*simp add*: *nonzero-inverse-minus-eq*)
done

lemma *inverse-less-iff-less-neg* [*simp*,*no-atp*]:

$a < 0 \Longrightarrow b < 0 \Longrightarrow \mathit{inverse}\ a < \mathit{inverse}\ b \longleftrightarrow b < a$
apply (*insert inverse-less-iff-less* [*of* $-b$ $-a$])
apply (*simp del*: *inverse-less-iff-less*
add: *nonzero-inverse-minus-eq*)
done

lemma *le-imp-inverse-le-neg*:
$a \leq b \Longrightarrow b < 0 \Longrightarrow \mathit{inverse}\ b \leq \mathit{inverse}\ a$
by (*force simp add*: *le-less less-imp-inverse-less-neg*)

lemma *inverse-le-iff-le-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow b < 0 \Longrightarrow \mathit{inverse}\ a \leq \mathit{inverse}\ b \longleftrightarrow b \leq a$
by (*blast intro*: *le-imp-inverse-le-neg dest*: *inverse-le-imp-le-neg*)

lemma *one-less-inverse*:
$0 < a \Longrightarrow a < 1 \Longrightarrow 1 < \mathit{inverse}\ a$
using *less-imp-inverse-less* [*of a 1*, *unfolded inverse-1*] **.**

lemma *one-le-inverse*:
$0 < a \Longrightarrow a \leq 1 \Longrightarrow 1 \leq \mathit{inverse}\ a$
using *le-imp-inverse-le* [*of a 1*, *unfolded inverse-1*] **.**

lemma *pos-le-divide-eq* [*field-simps*]: $0 < c \Longrightarrow (a \leq b/c) = (a*c \leq b)$
proof −
assume *less*: $0<c$
hence $(a \leq b/c) = (a*c \leq (b/c)*c)$
by (*simp add*: *mult-le-cancel-right less-not-sym* [*OF less*] *del*: *times-divide-eq*)
also have ... $= (a*c \leq b)$
by (*simp add*: *less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-le-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow (a \leq b/c) = (b \leq a*c)$
proof −
assume *less*: $c<0$
hence $(a \leq b/c) = ((b/c)*c \leq a*c)$
by (*simp add*: *mult-le-cancel-right less-not-sym* [*OF less*] *del*: *times-divide-eq*)
also have ... $= (b \leq a*c)$
by (*simp add*: *less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *pos-less-divide-eq* [*field-simps*]:
$0 < c \Longrightarrow (a < b/c) = (a*c < b)$
proof −
assume *less*: $0<c$
hence $(a < b/c) = (a*c < (b/c)*c)$
by (*simp add*: *mult-less-cancel-right-disj less-not-sym* [*OF less*] *del*: *times-divide-eq*)
also have ... $= (a*c < b)$

by (*simp add*: *less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-less-divide-eq* [*field-simps*]:
$c < 0 \Longrightarrow (a < b/c) = (b < a*c)$
proof −
assume *less*: *c<0*
hence $(a < b/c) = ((b/c)*c < a*c)$
by (*simp add*: *mult-less-cancel-right-disj less-not-sym* [*OF less*] *del*: *times-divide-eq*)
also have ... $= (b < a*c)$
by (*simp add*: *less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *pos-divide-less-eq* [*field-simps*]:
$0 < c \Longrightarrow (b/c < a) = (b < a*c)$
proof −
assume *less*: *0<c*
hence $(b/c < a) = ((b/c)*c < a*c)$
by (*simp add*: *mult-less-cancel-right-disj less-not-sym* [*OF less*] *del*: *times-divide-eq*)
also have ... $= (b < a*c)$
by (*simp add*: *less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-divide-less-eq* [*field-simps*]:
$c < 0 \Longrightarrow (b/c < a) = (a*c < b)$
proof −
assume *less*: *c<0*
hence $(b/c < a) = (a*c < (b/c)*c)$
by (*simp add*: *mult-less-cancel-right-disj less-not-sym* [*OF less*] *del*: *times-divide-eq*)
also have ... $= (a*c < b)$
by (*simp add*: *less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *pos-divide-le-eq* [*field-simps*]: $0 < c \Longrightarrow (b/c \leq a) = (b \leq a*c)$
proof −
assume *less*: *0<c*
hence $(b/c \leq a) = ((b/c)*c \leq a*c)$
by (*simp add*: *mult-le-cancel-right less-not-sym* [*OF less*] *del*: *times-divide-eq*)
also have ... $= (b \leq a*c)$
by (*simp add*: *less-imp-not-eq2* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

lemma *neg-divide-le-eq* [*field-simps*]: $c < 0 \Longrightarrow (b/c \leq a) = (a*c \leq b)$
proof −

assume *less*: $c<0$
hence $(b/c \leq a) = (a*c \leq (b/c)*c)$
by (*simp add*: *mult-le-cancel-right less-not-sym* [*OF less*] *del*: *times-divide-eq*)
also have $... = (a*c \leq b)$
by (*simp add*: *less-imp-not-eq* [*OF less*] *divide-inverse mult-assoc*)
finally show *?thesis* **.**
qed

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* [*no-atp*] = *algebra-simps*
zero-less-mult-iff mult-less-0-iff

lemmas (**in** −) *sign-simps* [*no-atp*] = *algebra-simps*
zero-less-mult-iff mult-less-0-iff

lemma *divide-pos-pos*:
$0 < x ==> 0 < y ==> 0 < x / y$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-pos*:
$0 <= x ==> 0 < y ==> 0 <= x / y$
by(*simp add*:*field-simps*)

lemma *divide-neg-pos*:
$x < 0 ==> 0 < y ==> x / y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-pos*:
$x <= 0 ==> 0 < y ==> x / y <= 0$
by(*simp add*:*field-simps*)

lemma *divide-pos-neg*:
$0 < x ==> y < 0 ==> x / y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-neg*:
$0 <= x ==> y < 0 ==> x / y <= 0$
by(*simp add*:*field-simps*)

lemma *divide-neg-neg*:
$x < 0 ==> y < 0 ==> 0 < x / y$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-neg*:
$x <= 0 ==> y < 0 ==> 0 <= x / y$

by(*simp add*:*field-simps*)

lemma *divide-strict-right-mono*:
[|$a < b$; $0 < c$|] ==> a / $c < b$ / c
by (*simp add*: *less-imp-not-eq2 divide-inverse mult-strict-right-mono positive-imp-inverse-positive*)

lemma *divide-strict-right-mono-neg*:
[|$b < a$; $c < 0$|] ==> a / $c < b$ / c
apply (*drule divide-strict-right-mono* [*of* - - $-c$], *simp*)
apply (*simp add*: *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
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono*)

lemma *divide-left-mono*:
[|$b \leq a$; $0 \leq c$; $0 < a*b$|] ==> c / $a \leq c$ / b
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-right-mono*)

lemma *divide-strict-left-mono-neg*:
[|$a < b$; $c < 0$; $0 < a*b$|] ==> c / $a < c$ / b
by(*auto simp*: *field-simps times-divide-eq zero-less-mult-iff mult-strict-right-mono-neg*)

lemma *mult-imp-div-pos-le*: $0 < y$ ==> $x <= z * y$ ==>
x / $y <= z$
by (*subst pos-divide-le-eq*, *assumption*+)

lemma *mult-imp-le-div-pos*: $0 < y$ ==> $z * y <= x$ ==>
$z <= x$ / y
by(*simp add*:*field-simps*)

lemma *mult-imp-div-pos-less*: $0 < y$ ==> $x < z * y$ ==>
x / $y < z$
by(*simp add*:*field-simps*)

lemma *mult-imp-less-div-pos*: $0 < y$ ==> $z * y < x$ ==>
$z < x$ / y
by(*simp add*:*field-simps*)

lemma *frac-le*: $0 <= x$ ==>
$x <= y$ ==> $0 < w$ ==> $w <= z$ ==> 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 <= 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 < x ==>
    x <= y ==> 0 < w ==> w < z  ==> x / z < y / w
  apply (rule mult-imp-div-pos-less)
  apply simp-all
  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.

```
lemma less-half-sum: a < b ==> a < (a+b) / (1+1)
by (simp add: field-simps zero-less-two)

lemma gt-half-sum: a < b ==> (a+b)/(1+1) < b
by (simp add: field-simps zero-less-two)

subclass dense-linorder
proof
  fix x y :: 'a
  from less-add-one show ∃ y. x < y ..
 from less-add-one have x + (- 1) < (x + 1) + (- 1) by (rule add-strict-right-mono)
  then have x - 1 < x + 1 - 1 by (simp only: diff-minus [symmetric])
  then have x - 1 < x by (simp add: algebra-simps)
  then show ∃ y. y < x ..
  show x < y ⟹ ∃ z>x. z < y by (blast intro!: less-half-sum gt-half-sum)
qed

lemma nonzero-abs-inverse:
     a ≠ 0 ==> |inverse a| = inverse |a|
apply (auto simp add: neq-iff abs-if nonzero-inverse-minus-eq
                      negative-imp-inverse-negative)
apply (blast intro: positive-imp-inverse-positive elim: less-asym)
done
```

lemma *nonzero-abs-divide*:
  $b \neq 0 \Longrightarrow |a\ /\ b| = |a|\ /\ |b|$
by (*simp add*: *divide-inverse abs-mult nonzero-abs-inverse*)

lemma *field-le-epsilon*:
assumes *e*: $\bigwedge e.\ 0 < e \Longrightarrow x \leq y + e$
shows $x \leq y$
proof (*rule dense-le*)
fix *t* **assume** $t < x$
hence $0 < x - t$ **by** (*simp add*: *less-diff-eq*)
from *e* [*OF this*] **have** $x + 0 \leq x + (y - t)$ **by** (*simp add*: *algebra-simps*)
then have $0 \leq y - t$ **by** (*simp only*: *add-le-cancel-left*)
then show $t \leq y$ **by** (*simp add*: *algebra-simps*)
qed

end

class *linordered-field-inverse-zero* = *linordered-field* + *field-inverse-zero*
begin

lemma *le-divide-eq*:
  $(a \leq b/c) =$
   (*if* $0 < c$ *then* $a*c \leq b$
         else if $c < 0$ *then* $b \leq a*c$
         else $a \leq 0$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-le-divide-eq neg-le-divide-eq linorder-neq-iff*)
done

lemma *inverse-positive-iff-positive* [*simp*]:
  $(0 < inverse\ a) = (0 < a)$
apply (*cases* $a = 0$, *simp*)
apply (*blast intro*: *inverse-positive-imp-positive positive-imp-inverse-positive*)
done

lemma *inverse-negative-iff-negative* [*simp*]:
  $(inverse\ a < 0) = (a < 0)$
apply (*cases* $a = 0$, *simp*)
apply (*blast intro*: *inverse-negative-imp-negative negative-imp-inverse-negative*)
done

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]:
  $0 \leq inverse\ a \longleftrightarrow 0 \leq a$
  by (*simp add*: *not-less* [*symmetric*])

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]:
  $inverse\ a \leq 0 \longleftrightarrow a \leq 0$
  by (*simp add*: *not-less* [*symmetric*])

lemma *one-less-inverse-iff*:
  $1 < inverse\ x \longleftrightarrow 0 < x \wedge x < 1$
proof *cases*
  assume $0 < x$
    with *inverse-less-iff-less* [*OF zero-less-one*, *of x*]
    show *?thesis* **by** *simp*
next
  assume *notless*: $\sim (0 < x)$
  have $\sim (1 < inverse\ x)$
  proof
    assume $1 < inverse\ x$
    also with *notless* **have** $... \leq 0$ **by** *simp*
    also have $... < 1$ **by** (*rule zero-less-one*)
    finally show *False* **by** *auto*
  qed
  with *notless* **show** *?thesis* **by** *simp*
qed

lemma *one-le-inverse-iff*:
  $1 \leq inverse\ x \longleftrightarrow 0 < x \wedge x \leq 1$
proof (*cases* $x = 1$)
  case *True* **then show** *?thesis* **by** *simp*
next
  case *False* **then have** $inverse\ x \neq 1$ **by** *simp*
  then have $1 \neq inverse\ x$ **by** *blast*
  then have $1 \leq inverse\ x \longleftrightarrow 1 < inverse\ x$ **by** (*simp add*: *le-less*)
  with *False* **show** *?thesis* **by** (*auto simp add*: *one-less-inverse-iff*)
qed

lemma *inverse-less-1-iff*:
  $inverse\ x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$
  by (*simp add*: *not-le* [*symmetric*] *one-le-inverse-iff*)

lemma *inverse-le-1-iff*:
  $inverse\ x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$
  by (*simp add*: *not-less* [*symmetric*] *one-less-inverse-iff*)

lemma *divide-le-eq*:
  $(b/c \leq a) =$
   (*if* $0 < c$ *then* $b \leq a*c$
         else if $c < 0$ *then* $a*c \leq b$
         else $0 \leq a$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-divide-le-eq neg-divide-le-eq*)
done

lemma *less-divide-eq*:
  $(a < b/c) =$

(*if* $0 < c$ *then* $a*c < b$
else if $c < 0$ *then* $b < a*c$
else $a < 0$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-less-divide-eq neg-less-divide-eq*)
done

lemma *divide-less-eq*:
$(b/c < a) =$
(*if* $0 < c$ *then* $b < a*c$
else if $c < 0$ *then* $a*c < b$
else $0 < a$)
apply (*cases* $c=0$, *simp*)
apply (*force simp add*: *pos-divide-less-eq neg-divide-less-eq*)
done

Division and Signs

lemma *zero-less-divide-iff*:
$(0 < a/b) = (0 < a \ \& \ 0 < b \mid a < 0 \ \& \ b < 0)$
by (*simp add*: *divide-inverse zero-less-mult-iff*)

lemma *divide-less-0-iff*:
$(a/b < 0) =$
$(0 < a \ \& \ b < 0 \mid a < 0 \ \& \ 0 < b)$
by (*simp add*: *divide-inverse mult-less-0-iff*)

lemma *zero-le-divide-iff*:
$(0 \leq a/b) =$
$(0 \leq a \ \& \ 0 \leq b \mid a \leq 0 \ \& \ b \leq 0)$
by (*simp add*: *divide-inverse zero-le-mult-iff*)

lemma *divide-le-0-iff*:
$(a/b \leq 0) =$
$(0 \leq a \ \& \ b \leq 0 \mid a \leq 0 \ \& \ 0 \leq b)$
by (*simp add*: *divide-inverse mult-le-0-iff*)

Division and the Number One

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [*simp*,*no-atp*]:
$(0 = 1/a) = (a = 0)$
apply (*cases* $a=0$, *simp*)
apply (*auto simp add*: *nonzero-eq-divide-eq*)
done

lemma *one-divide-eq-0-iff* [*simp*,*no-atp*]:
$(1/a = 0) = (a = 0)$
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 $0 < x$

lemma *zero-le-divide-1-iff* [*simp*, *no-atp*]:
$0 \leq 1 \,/\, a \longleftrightarrow 0 \leq a$
by (*simp add*: *zero-le-divide-iff*)

lemma *zero-less-divide-1-iff* [*simp*, *no-atp*]:
$0 < 1 \,/\, a \longleftrightarrow 0 < a$
by (*simp add*: *zero-less-divide-iff*)

lemma *divide-le-0-1-iff* [*simp*, *no-atp*]:
$1 \,/\, a \leq 0 \longleftrightarrow a \leq 0$
by (*simp add*: *divide-le-0-iff*)

lemma *divide-less-0-1-iff* [*simp*, *no-atp*]:
$1 \,/\, a < 0 \longleftrightarrow a < 0$
by (*simp add*: *divide-less-0-iff*)

lemma *divide-right-mono*:
$[|a \leq b; 0 \leq c|] \Longrightarrow a/c \leq b/c$
by (*force simp add*: *divide-strict-right-mono le-less*)

lemma *divide-right-mono-neg*: $a <= b$
$\Longrightarrow c <= 0 \Longrightarrow b \,/\, c <= a \,/\, c$
apply (*drule divide-right-mono* [*of* - - − *c*])
apply *auto*
done

lemma *divide-left-mono-neg*: $a <= b$
$\Longrightarrow c <= 0 \Longrightarrow 0 < a * b \Longrightarrow c \,/\, a <= c \,/\, b$
apply (*drule divide-left-mono* [*of* - - − *c*])
apply (*auto simp add*: *mult-commute*)
done

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1* [*no-atp*]:
$(1 \leq b \,/\, a) = ((0 < a \;\&\; a \leq b) \mid (a < 0 \;\&\; b \leq a))$
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1* [*no-atp*]:
$(b \,/\, a \leq 1) = ((0 < a \;\&\; b \leq a) \mid (a < 0 \;\&\; a \leq b) \mid a=0)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1* [*no-atp*]:
$(1 < b \,/\, a) = ((0 < a \;\&\; a < b) \mid (a < 0 \;\&\; b < a))$
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1* [*no-atp*]:
$(b \mathbin{/} a < 1) = ((0 < a \mathbin{\&} b < a) \mid (a < 0 \mathbin{\&} a < b) \mid a{=}0)$
by (*auto simp add*: *divide-less-eq*)

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
by (*auto simp add*: *le-divide-eq*)

lemma *le-divide-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
by (*auto simp add*: *divide-le-eq*)

lemma *divide-le-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (1 < b/a) = (a < b)$
by (*auto simp add*: *less-divide-eq*)

lemma *less-divide-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow (1 < b/a) = (b < a)$
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1-pos* [*simp*,*no-atp*]:
$0 < a \Longrightarrow (b/a < 1) = (b < a)$
by (*auto simp add*: *divide-less-eq*)

lemma *divide-less-eq-1-neg* [*simp*,*no-atp*]:
$a < 0 \Longrightarrow b/a < 1 \mathrel{<->} a < b$
by (*auto simp add*: *divide-less-eq*)

lemma *eq-divide-eq-1* [*simp*,*no-atp*]:
$(1 = b/a) = ((a \neq 0 \mathbin{\&} a = b))$
by (*auto simp add*: *eq-divide-eq*)

lemma *divide-eq-eq-1* [*simp*,*no-atp*]:
$(b/a = 1) = ((a \neq 0 \mathbin{\&} a = b))$
by (*auto simp add*: *divide-eq-eq*)

lemma *abs-inverse* [*simp*]:
$|inverse\ a| =$
$inverse\ |a|$
apply (*cases a=0*, *simp*)

apply (*simp add*: *nonzero-abs-inverse*)
done

lemma *abs-divide* [*simp*]:
  $|a\ /\ b| = |a|\ /\ |b|$
apply (*cases b=0*, *simp*)
apply (*simp add*: *nonzero-abs-divide*)
done

lemma *abs-div-pos*: $0 < y \Longrightarrow$
  $|x|\ /\ y = |x\ /\ y|$
 apply (*subst abs-divide*)
 apply (*simp add*: *order-less-imp-le*)
done

lemma *field-le-mult-one-interval*:
 assumes $*$: $\bigwedge z.\ [\![\ 0 < z\ ;\ z < 1\ ]\!] \Longrightarrow z * x \leq y$
 shows $x \leq y$
proof (*cases* $0 < x$)
 assume $0 < x$
 thus *?thesis*
  using *dense-le-bounded*[*of 0 1 y*/*x*] $*$
  unfolding *le-divide-eq if-P*[*OF* ‹$0 < x$›] **by** *simp*
next
 assume $\neg 0 < x$ **hence** $x \leq 0$ **by** *simp*
 obtain s::$'a$ **where** s: $0 < s\ s < 1$ **using** *dense*[*of 0 1*::$'a$] **by** *auto*
 hence $x \leq s * x$ **using** *mult-le-cancel-right*[*of 1 x s*] ‹$x \leq 0$› **by** *auto*
 also note $*$[*OF s*]
 finally show *?thesis* **.**
qed

end

code-modulename *SML*
 Fields Arith

code-modulename *OCaml*
 Fields Arith

code-modulename *Haskell*
 Fields Arith

end

15 Nat: Natural numbers

theory *Nat*
imports *Inductive Typedef Fun Fields*

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

15.1 Type *ind*

typedecl *ind*

axiomatization
Zero-Rep :: *ind* **and**
Suc-Rep :: *ind* => *ind*
where
— the axiom of infinity in 2 parts
Suc-Rep-inject: *Suc-Rep x* = *Suc-Rep y* ==> *x* = *y* **and**
Suc-Rep-not-Zero-Rep: *Suc-Rep x* ≠ *Zero-Rep*

15.2 Type nat

Type definition

inductive *Nat* :: *ind* ⇒ *bool*
where
Zero-RepI: *Nat Zero-Rep*
| *Suc-RepI*: *Nat i* ⟹ *Nat* (*Suc-Rep i*)

global

typedef (**open** *Nat*)
nat = *Nat*
by (*rule exI*, *unfold mem-def*, *rule Nat.Zero-RepI*)

definition *Suc* :: *nat* => *nat* **where**
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*)

lemma *Suc-Rep-inject'*: *Suc-Rep x* = *Suc-Rep y* ⟷ *x* = *y*
by (*rule iffI*, *rule Suc-Rep-inject*) *simp-all*

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*]
Suc-Rep-inject' 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*: $Suc\ x = Suc\ y \Longrightarrow x = y$
by (*rule inj-Suc* [*THEN injD*])

lemma *n-not-Suc-n*: $n \neq Suc\ n$
by (*induct n*) *simp-all*

lemma *Suc-n-not-n*: $Suc\ n \neq n$
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

15.3 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

hide-fact (**open**) *add-0 add-0-right diff-0*

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

15.3.1 Addition

lemma *nat-add-assoc*: $(m + n) + k = m + ((n + k)::nat)$
by (*rule add-assoc*)

lemma *nat-add-commute*: $m + n = n + (m::nat)$
by (*rule add-commute*)

lemma *nat-add-left-commute*: $x + (y + z) = y + ((x + z)::nat)$
by (*rule add-left-commute*)

lemma *nat-add-left-cancel* [*simp*]: $(k + m = k + n) = (m = (n::nat))$
by (*rule add-left-cancel*)

lemma *nat-add-right-cancel* [*simp*]: $(m + k = n + k) = (m=(n::nat))$
by (*rule add-right-cancel*)

Reasoning about $m + 0 = 0$, etc.

lemma *add-is-0* [*iff*]:
fixes *m n* :: *nat*
shows $(m + n = 0) = (m = 0 \ \& \ n = 0)$
by (*cases m*) *simp-all*

lemma *add-is-1*:
$(m+n= Suc\ 0) = (m= Suc\ 0 \ \& \ n=0 \mid m=0 \ \& \ n= Suc\ 0)$
by (*cases m*) *simp-all*

lemma *one-is-add*:
$(Suc\ 0 = m + n) = (m = Suc\ 0 \ \& \ n = 0 \mid m = 0 \ \& \ n = Suc\ 0)$
by (*rule trans*, *rule eq-commute*, *rule add-is-1*)

lemma *add-eq-self-zero*:
fixes *m n* :: *nat*
shows $m + n = m \Longrightarrow n = 0$
by (*induct m*) *simp-all*

lemma *inj-on-add-nat*[*simp*]: *inj-on* (%*n*::*nat*. *n*+*k*) *N*
apply (*induct k*)
apply *simp*
apply(*drule comp-inj-on*[*OF - inj-Suc*])
apply (*simp add*:*o-def*)
done

15.3.2 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

15.3.3 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*,*no-atp*]: $(Suc\ 0 = m * n) = (m = Suc\ 0\ \&\ n = Suc\ 0)$
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*

15.4 Orders on *nat*

15.4.1 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*) ≤ *n* ⟷ *True* **by** (*simp add*: *less-eq-nat.simps*)
lemma *le0* [*iff*]: *0* ≤ (*n*::*nat*) **by** (*simp add*: *less-eq-nat.simps*)

definition *less-nat* **where**
less-eq-Suc-le: *n* < *m* ⟷ *Suc n* ≤ *m*

lemma *Suc-le-mono* [*iff*]: *Suc n* ≤ *Suc m* ⟷ *n* ≤ *m*
by (*simp add*: *less-eq-nat.simps*(*2*))

lemma *Suc-le-eq* [*code*]: *Suc m* ≤ *n* ⟷ *m* < *n*
unfolding *less-eq-Suc-le* **..**

lemma *le-0-eq* [*iff*]: (*n*::*nat*) ≤ *0* ⟷ *n* = *0*
by (*induct n*) (*simp-all add*: *less-eq-nat.simps*(*2*))

lemma *not-less0* [*iff*]: ¬ *n* < (*0*::*nat*)
by (*simp add*: *less-eq-Suc-le*)

lemma *less-nat-zero-code* [*code*]: *n* < (*0*::*nat*) ⟷ *False*
by *simp*

lemma *Suc-less-eq* [*iff*]: *Suc m* < *Suc n* ⟷ *m* < *n*
by (*simp add*: *less-eq-Suc-le*)

lemma *less-Suc-eq-le* [*code*]: *m* < *Suc n* ⟷ *m* ≤ *n*
by (*simp add*: *less-eq-Suc-le*)

lemma *le-SucI*: *m* ≤ *n* ⟹ *m* ≤ *Suc n*
by (*induct m arbitrary*: *n*)
(*simp-all add*: *less-eq-nat.simps*(*2*) *split*: *nat.splits*)

lemma *Suc-leD*: *Suc m* ≤ *n* ⟹ *m* ≤ *n*
by (*cases n*) (*auto intro*: *le-SucI*)

lemma *less-SucI*: *m* < *n* ⟹ *m* < *Suc n*
by (*simp add*: *less-eq-Suc-le*) (*erule Suc-leD*)

lemma *Suc-lessD*: *Suc m* < *n* ⟹ *m* < *n*
by (*simp add*: *less-eq-Suc-le*) (*erule Suc-leD*)

instance
proof
fix *n m* :: *nat*
show *n* < *m* ⟷ *n* ≤ *m* ∧ ¬ *m* ≤ *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

15.4.2 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*)

15.4.3 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*, *no-atp*]: $(n < (1::nat)) = (n = 0)$
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

15.4.4 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-antisym*: $[\!|\ m \leq n; n \leq m\ |\!] \Longrightarrow m = (n::nat)$
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*: $(m::nat) \leq n \Longrightarrow m \neq n \Longrightarrow m < n$
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*: $m \leq n \Longrightarrow (n < Suc\ m) = (n = m)$
unfolding *less-Suc-eq-le* **by** *auto*

lemma *not-less-less-Suc-eq*: $\sim n < m \Longrightarrow (n < Suc\ m) = (n = m)$
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*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n) \Longrightarrow f\ 0 = c$
by *simp*

lemma *def-nat-rec-Suc*: $(!!n.\ f\ n == nat\text{-}rec\ c\ h\ n) \Longrightarrow f\ (Suc\ n) = h\ n\ (f\ n)$
by *simp*

lemma *not0-implies-Suc*: $n \neq 0 \Longrightarrow \exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr0-implies-Suc*: $n > 0 \Longrightarrow \exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr-implies-not0*: **fixes** *n* :: *nat* **shows** $m<n \Longrightarrow n \neq 0$
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*: $((n::nat) = 0 \Longrightarrow False) \Longrightarrow 0 < n$
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*,*no-atp*]: $!!n::nat.\ (\sim (0 < n)) = (n = 0)$
using *neq0-conv* **by** *blast*

lemma *Suc-le-D*: $(Suc\ n \leq m') \Longrightarrow (?\ m.\ m' = Suc\ m)$
by (*induct m'*) *simp-all*

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $(m < Suc\ n) = (m = 0 \mid (\exists j.\ m = Suc\ j\ \&\ j < n))$
by (*cases m*) *simp-all*

15.4.5 *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*)

15.4.6 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!: *Nat.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* :: *linordered-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* ~= *0* ⟹ *b* ~= *0* ⟹ *a* * *b* ~= *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*

15.4.7 Additional theorems about *op* ≤

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder* **proof**
  fix *P* **and** *n* :: *nat*
  assume *step*: ⋀*n*::*nat*. (⋀*m*. *m* < *n* ⟹ *P m*) ⟹ *P n*
  have ⋀*q*. *q* ≤ *n* ⟹ *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* ≤ *m* ∨ *n* = *Suc m* **by** (*simp add*: *le-Suc-eq*)
    thus *?case*
    proof
      assume *n* ≤ *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*: $\neg P(0) \Longrightarrow P(n::nat) \Longrightarrow \exists k{<}n.\ (\forall i{\leq}k.\ \neg P\ i)\ \&\ P(k{+}1)$
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*. $\forall m::nat.\ m < n \longrightarrow P\ m \Longrightarrow P\ n$ **shows** *P n*
using *assms less-induct* **by** *blast*

lemma *measure-induct-rule* [*case-names less*]:
fixes *f* :: $'a \Rightarrow nat$
assumes *step*: $\bigwedge x.\ (\bigwedge y.\ f\ y < f\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows *P a*
by (*induct m*≡*f a arbitrary*: *a rule*: *less-induct*) (*auto intro*: *step*)

old style induction rules:

lemma *measure-induct*:
fixes *f* :: $'a \Rightarrow nat$
shows $(\bigwedge x.\ \forall y.\ f\ y < f\ x \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$
by (*rule measure-induct-rule* [*of f P a*]) *iprover*

lemma *full-nat-induct*:
assumes *step*: (!!*n*. (*ALL m*. *Suc m* <= *n* --> *P m*) ==> *P n*)
shows *P n*
by (*rule less-induct*) (*auto intro*: *step simp*:*le-simps*)

An induction rule for estabilishing binary relations

lemma *less-Suc-induct*:
assumes *less*: *i* < *j*
and *step*: !!*i*. *P i* (*Suc i*)
and *trans*: !!*i j k*. *i* < *j* ==> *j* < *k* ==> *P i j* ==> *P j k* ==> *P i k*
shows *P i j*
proof −
from *less* **obtain** *k* **where** *j*: *j* = *Suc* (*i* + *k*) **by** (*auto dest*: *less-imp-Suc-add*)
have *P i* (*Suc* (*i* + *k*))
proof (*induct k*)
case *0*
show *?case* **by** (*simp add*: *step*)
next

```
    case (Suc k)
    have 0 + i < Suc k + i by (rule add-less-mono1) simp
    hence i < Suc (i + k) by (simp add: add-commute)
    from trans[OF this lessI Suc step]
    show ?case by simp
  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. ¬ P n ⟹ ∃ m<n. ¬ P m ⟧ ⟹ P n
by (induct n rule: less-induct, auto)
```

```
lemma infinite-descent0[case-names 0 smaller]:
  ⟦ P 0; !!n. n>0 ⟹ ¬ P n ⟹ (∃ m::nat. m < n ∧ ¬P m) ⟧ ⟹ P n
by (rule infinite-descent) (case-tac n>0, auto)
```

Infinite descent using a mapping to $\mathbb{N}$: $P(x)$ is true for all $x \in D$ if there exists a $V : D \to \mathbb{N}$ and

- case "0": given $V(x) = 0$ prove $P(x)$,
- case "smaller": given $V(x) > 0$ and $\neg P(x)$ prove there exists a $y \in D$ such that $V(y) < V(x)$ and $\neg P(y)$.

NB: the proof also shows how to use the previous lemma.

```
corollary infinite-descent0-measure [case-names 0 smaller]:
  assumes A0: !!x. V x = (0::nat) ⟹ P x
    and   A1: !!x. V x > 0 ⟹ ¬P x ⟹ (∃ y. V y < V x ∧ ¬P y)
  shows P x
proof −
  obtain n where n = V x by auto
  moreover have ⋀x. V x = n ⟹ P x
  proof (induct n rule: infinite-descent0)
    case 0 — i.e. V(x) = 0
    with A0 show P x by auto
  next — now n > 0 and P(x) does not hold for some x with V(x) = n
    case (smaller n)
    then obtain x where vxn: V x = n  and V x > 0 ∧ ¬ P x by auto
```

with *A1* **obtain** y **where** $V\ y < V\ x \wedge \neg\ P\ y$ **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*: $i \leq j \Longrightarrow i + k \leq j + (k::nat)$
by (*rule add-right-mono*)

non-strict, in both arguments

lemma *add-le-mono*: $[|\ i \leq j;\ \ k \leq l\ |] \Longrightarrow i + k \leq j + (l::nat)$
by (*rule add-mono*)

lemma *le-add2*: $n \leq ((m + n)::nat)$
by (*insert add-right-mono* [*of 0 m n*], *simp*)

lemma *le-add1*: $n \leq ((n + m)::nat)$
by (*simp add*: *add-commute*, *rule le-add2*)

lemma *less-add-Suc1*: $i < Suc\ (i + m)$
by (*rule le-less-trans*, *rule le-add1*, *rule lessI*)

lemma *less-add-Suc2*: $i < Suc\ (m + i)$
by (*rule le-less-trans*, *rule le-add2*, *rule lessI*)

lemma *less-iff-Suc-add*: $(m < n) = (\exists\, k.\ n = Suc\ (m + k))$
by (*iprover intro*!: *less-add-Suc1 less-imp-Suc-add*)

lemma *trans-le-add1*: $(i::nat) \leq j \Longrightarrow i \leq j + m$
by (*rule le-trans*, *assumption*, *rule le-add1*)

lemma *trans-le-add2*: $(i::nat) \leq j \Longrightarrow i \leq m + j$
by (*rule le-trans*, *assumption*, *rule le-add2*)

lemma *trans-less-add1*: $(i::nat) < j \Longrightarrow i < j + m$
by (*rule less-le-trans*, *assumption*, *rule le-add1*)

lemma *trans-less-add2*: $(i::nat) < j \Longrightarrow i < m + j$
by (*rule less-le-trans*, *assumption*, *rule le-add2*)

lemma *add-lessD1*: $i + j < (k::nat) \Longrightarrow i < k$
apply (*rule le-less-trans* [*of* - *i+j*])
apply (*simp-all add*: *le-add1*)
done

lemma *not-add-less1* [*iff*]: $\sim (i + j < (i::nat))$
apply (*rule notI*)
apply (*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*: $!!k::nat.\ k < l \Longrightarrow m + l = k + n \Longrightarrow m < n$
by (*force simp del*: *add-Suc-right*
simp add: *less-iff-Suc-add add-Suc-right* [*symmetric*] *add-ac*)

15.4.8 More results about difference

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add-diff-inverse*: $\sim\ m < n \Longrightarrow n + (m - n) = (m::nat)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *le-add-diff-inverse* [*simp*]: $n \leq m \Longrightarrow n + (m - n) = (m::nat)$
by (*simp add*: *add-diff-inverse linorder-not-less*)

lemma *le-add-diff-inverse2* [*simp*]: $n \leq m \Longrightarrow (m - n) + n = (m::nat)$
by (*simp add*: *add-commute*)

lemma *Suc-diff-le*: $n \leq m \Longrightarrow Suc\ m - n = Suc\ (m - n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-less-Suc*: $m - n < Suc\ m$
apply (*induct m n rule*: *diff-induct*)
apply (*erule-tac* [*3*] *less-SucE*)
apply (*simp-all add*: *less-Suc-eq*)
done

lemma *diff-le-self* [*simp*]: $m - n \leq (m::nat)$
by (*induct m n rule*: *diff-induct*) (*simp-all add*: *le-SucI*)

lemma *le-iff-add*: $(m::nat) \leq n = (\exists k.\ n = m + k)$
by (*auto simp*: *le-add1 dest*!: *le-add-diff-inverse sym* [*of* - *n*])

lemma *less-imp-diff-less*: $(j::nat) < k \Longrightarrow j - n < k$
by (*rule le-less-trans*, *rule diff-le-self*)

lemma *diff-Suc-less* [*simp*]: $0<n \Longrightarrow n - Suc\ i < n$
by (*cases n*) (*auto simp add*: *le-simps*)

lemma *diff-add-assoc*: $k \leq (j::nat) \Longrightarrow (i + j) - k = i + (j - k)$
by (*induct j k rule*: *diff-induct*) *simp-all*

lemma *diff-add-assoc2*: $k \leq (j::nat) \Longrightarrow (j + i) - k = (j - k) + i$
by (*simp add*: *add-commute diff-add-assoc*)

lemma *le-imp-diff-is-add*: $i \leq (j::nat) \Longrightarrow (j - i = k) = (j = k + i)$
by (*auto simp add*: *diff-add-inverse2*)

lemma *diff-is-0-eq* [*simp*]: $((m::nat) - n = 0) = (m \leq n)$
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*)

15.4.9 Monotonicity of Multiplication

lemma *mult-le-mono1*: $i \le (j::nat) \Longrightarrow i * k \le j * k$
by (*simp add*: *mult-right-mono*)

lemma *mult-le-mono2*: $i \le (j::nat) \Longrightarrow k * i \le k * j$
by (*simp add*: *mult-left-mono*)

$\le$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \le (j::nat) \Longrightarrow k \le l \Longrightarrow i * k \le j * l$
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*]: $(Suc\ 0 \le m * n) = (Suc\ 0 \le m \;\&\; Suc\ 0 \le n)$
apply (*induct m*)

apply *simp*
apply (*case-tac n*)
apply *simp-all*
done

lemma *mult-less-cancel2* [*simp*]: $((m::nat) * k < n * k) = (0 < k \ \& \ m < n)$
apply (*safe intro*!: *mult-less-mono1*)
apply (*case-tac k*, *auto*)
apply (*simp del*: *le-0-eq add*: *linorder-not-le* [*symmetric*])
apply (*blast intro*: *mult-le-mono1*)
done

lemma *mult-less-cancel1* [*simp*]: $(k * (m::nat) < k * n) = (0 < k \ \& \ m < n)$
by (*simp add*: *mult-commute* [*of k*])

lemma *mult-le-cancel1* [*simp*]: $(k * (m::nat) \leq k * n) = (0 < k \longrightarrow m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *mult-le-cancel2* [*simp*]: $((m::nat) * k \leq n * k) = (0 < k \longrightarrow m \leq n)$
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *Suc-mult-less-cancel1*: $(Suc\ k * m < Suc\ k * n) = (m < n)$
by (*subst mult-less-cancel1*) *simp*

lemma *Suc-mult-le-cancel1*: $(Suc\ k * m \leq Suc\ k * n) = (m \leq n)$
by (*subst mult-le-cancel1*) *simp*

lemma *le-square*: $m \leq m * (m::nat)$
by (*cases m*) (*auto intro*: *le-add1*)

lemma *le-cube*: $(m::nat) \leq m * (m * m)$
by (*cases m*) (*auto intro*: *le-add1*)

Lemma for *gcd*

lemma *mult-eq-self-implies-10*: $(m::nat) = m * n \Longrightarrow n = 1 \mid m = 0$
apply (*drule sym*)
apply (*rule disjCI*)
apply (*rule nat-less-cases*, *erule-tac* [*2*] -)
apply (*drule-tac* [*2*] *mult-less-mono2*)
apply (*auto*)
done

the lattice order on *nat*

instantiation *nat* :: *distrib-lattice*
begin

definition
$(inf :: nat \Rightarrow nat \Rightarrow nat) = min$

definition
(*sup* :: *nat* ⇒ *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

15.5 Natural operation of natural numbers on functions

We use the same logical constant for the power operations on functions and relations, in order to share the same syntax.

consts *compow* :: *nat* ⇒ ('*a* ⇒ '*b*) ⇒ ('*a* ⇒ '*b*)

abbreviation *compower* :: ('*a* ⇒ '*b*) ⇒ *nat* ⇒ '*a* ⇒ '*b* (**infixr** ^^ *80*) **where**
f ^^ *n* ≡ *compow n f*

notation (*latex* **output**)
compower ((-⁻) [*1000*] *1000*)

notation (*HTML* **output**)
compower ((-⁻) [*1000*] *1000*)

f ^^ *n* = *f o* ... *o f*, the n-fold composition of *f*

overloading
funpow == *compow* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ ('*a* ⇒ '*a*)
begin

primrec *funpow* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a* **where**
funpow 0 f = *id*
| *funpow* (*Suc n*) *f* = *f o funpow n f*

end

for code generation

definition *funpow* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a* **where**
funpow-code-def [*code-post*]: *funpow* = *compow*

lemmas [*code-unfold*] = *funpow-code-def* [*symmetric*]

lemma [*code*]:
funpow 0 f = *id*
funpow (*Suc n*) *f* = *f o funpow n f*
unfolding *funpow-code-def* **by** *simp-all*

hide-const (**open**) *funpow*

lemma *funpow-add*:
f ^^ (m + n) = f ^^ m ∘ f ^^ n
by (*induct m*) *simp-all*

lemma *funpow-swap1*:
f ((f ^^ n) x) = (f ^^ n) (f x)
proof −
have *f ((f ^^ n) x) = (f ^^ (n + 1)) x* **by** *simp*
also have ... *= (f ^^ n o f ^^ 1) x* **by** (*simp only*: *funpow-add*)
also have ... *= (f ^^ n) (f x)* **by** *simp*
finally show *?thesis* **.**
qed

15.6 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*:
of-nat n = of-nat-aux (λi. i + 1) n 0
proof (*induct n*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc n*)
have ⋀*i*. *of-nat-aux (λi. i + 1) n (i + 1) = of-nat-aux (λi. i + 1) n i + 1*
by (*induct n*) *simp-all*
from *this* [*of 0*] **have** *of-nat-aux (λi. i + 1) n 1 = of-nat-aux (λi. i + 1) n 0 + 1*
by *simp*
with *Suc* **show** *?case* **by** (*simp add*: *add-commute*)

qed

end

declare *of-nat-code* [*code*, *code-unfold*, *code-inline del*]

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*, *no-atp*]: *0* = *of-nat n* ⟷ *0* = *n*
  by (*rule of-nat-eq-iff* [*of 0 n*, *unfolded of-nat-0*])

lemma *of-nat-eq-0-iff* [*simp*, *no-atp*]: *of-nat m* = *0* ⟷ *m* = *0*
  by (*rule of-nat-eq-iff* [*of m 0*, *unfolded of-nat-0*])

lemma *inj-of-nat*: *inj of-nat*
  by (*simp add*: *inj-on-def*)

end

context *linordered-semidom*
begin

lemma *zero-le-imp-of-nat*: *0* ≤ *of-nat m*
  by (*induct m*) *simp-all*

lemma *less-imp-of-nat-less*: *m* < *n* ⟹ *of-nat m* < *of-nat n*
  apply (*induct m n rule*: *diff-induct*, *simp-all*)
  apply (*rule add-pos-nonneg* [*OF zero-less-one zero-le-imp-of-nat*])
  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 *linordered-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*]: $0 \leq$ *of-nat n*
by (*rule of-nat-le-iff* [*of 0*, *simplified*])

lemma *of-nat-le-0-iff* [*simp*, *no-atp*]: *of-nat m* $\leq 0 \longleftrightarrow m = 0$
by (*rule of-nat-le-iff* [*of - 0*, *simplified*])

lemma *of-nat-0-less-iff* [*simp*]: $0 <$ *of-nat n* $\longleftrightarrow 0 < n$
by (*rule of-nat-less-iff* [*of 0*, *simplified*])

lemma *of-nat-less-0-iff* [*simp*]: $\neg$ *of-nat m* < 0
by (*rule of-nat-less-iff* [*of - 0*, *simplified*])

end

context *ring-1*
begin

lemma *of-nat-diff*: $n \leq m \Longrightarrow$ *of-nat* $(m - n) =$ *of-nat m* $-$ *of-nat n*
by (*simp add*: *algebra-simps of-nat-add* [*symmetric*])

end

context *linordered-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*) *simp-all*

lemma *of-nat-eq-id* [*simp*]: *of-nat* $=$ *id*
by (*auto simp add*: *expand-fun-eq*)

15.7 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

lemma *Nats-cases* [*cases set*: *Nats*]:
assumes *x* ∈ ℕ
obtains (*of-nat*) *n* **where** *x* = *of-nat n*
unfolding *Nats-def*
proof −
from ‹*x* ∈ ℕ› **have** *x* ∈ *range of-nat* **unfolding** *Nats-def* **.**
then obtain *n* **where** *x* = *of-nat n* **..**
then show *thesis* **..**
qed

lemma *Nats-induct* [*case-names of-nat*, *induct set*: *Nats*]:
x ∈ ℕ ⟹ (⋀*n*. *P* (*of-nat n*)) ⟹ *P x*
by (*rule Nats-cases*) *auto*

end

15.8 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*
setup ⟪ *Lin-Arith.global-setup* ⟫
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. $f\ n \leq f(Suc\ n)$ **and** $n \leq n'$
shows $f\ n \leq f\ n'$
proof (*cases* $n < n'$)
case *True*
thus *?thesis*
by (*induct n n' rule*: *less-Suc-induct*[*consumes 1*]) (*auto intro*: *mono*)
qed (*insert* ‹$n \leq n'$›, *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. $f\ n < f(Suc\ n)$) $\Longrightarrow f(n) < f(m) \longleftrightarrow n<m$
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* = ($\forall\, n.\ f\ n \leq f\ (Suc\ n)$)
unfolding *mono-def*
by (*auto intro*:*lift-Suc-mono-le*[*of f*])

lemma *mono-nat-linear-lb*:
(!!$m\ n$::*nat*. $m<n \Longrightarrow f\ m < f\ n) \Longrightarrow f(m)+k \leq f(m+k)$
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*: [| $a < (b::nat)$; $c \leq a$ |] ==> $a-c < b-c$
by *arith*

lemma *less-diff-conv*: $(i < j-k) = (i+k < (j::nat))$
by *arith*

lemma *le-diff-conv*: $(j-k \leq (i::nat)) = (j \leq i+k)$
by *arith*

lemma *le-diff-conv2*: $k \leq j$ ==> $(i \leq j-k) = (i+k \leq (j::nat))$
by *arith*

lemma *diff-diff-cancel* [*simp*]: $i \leq (n::nat)$ ==> $n - (n - i) = i$
by *arith*

lemma *le-add-diff*: $k \leq (n::nat)$ ==> $m \leq n + m - k$
by *arith*

lemma *diff-less*[*simp*]: !!$m::nat$. [| $0<n$; $0<m$ |] ==> $m - n < m$
by *arith*

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $((m-k) - (n-k)) = (m-n)$
by (*simp split add*: *nat-diff-split*)

hide-fact (**open**) *diff-diff-eq*

lemma *eq-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k = n-k) = (m=n)$
by (*auto split add*: *nat-diff-split*)

lemma *less-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k < n-k) = (m<n)$
by (*auto split add*: *nat-diff-split*)

lemma *le-diff-iff*: [| $k \leq m$; $k \leq (n::nat)$ |] ==> $(m-k \leq n-k) = (m \leq n)$
by (*auto split add*: *nat-diff-split*)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq (n::nat)$ ==> $(m-l) \leq (n-l)$
by (*simp split add*: *nat-diff-split*)

lemma *diff-le-mono2*: $m \leq (n::nat)$ ==> $(l-n) \leq (l-m)$
by (*simp split add*: *nat-diff-split*)

lemma *diff-less-mono2*: [| $m < (n::nat)$; $m<l$ |] ==> $(l-n) < (l-m)$
by (*simp split add*: *nat-diff-split*)

lemma *diffs0-imp-equal*: !!$m::nat$. [| $m-n = 0$; $n-m = 0$ |] ==> $m=n$
by (*simp split add*: *nat-diff-split*)

lemma *min-diff*: *min* $(m - (i::nat))$ $(n - i) = min\ m\ n - i$
by *auto*

lemma *inj-on-diff-nat*:
assumes *k-le-n*: $\forall n \in N.\ k \leq (n::nat)$
shows *inj-on* $(\lambda n.\ n - k)$ N
proof (*rule inj-onI*)
fix $x\ y$
assume *a*: $x \in N\ y \in N\ x - k = y - k$
with *k-le-n* **have** $x - k + k = y - k + k$ **by** *auto*
with *a k-le-n* **show** $x = y$ **by** *auto*
qed

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j$ --> $i - (j - k) = i + (k::nat) - j$
by *arith*

lemma *diff-Suc-diff-eq1* [*simp*]: $k \leq j$ ==> $m - Suc\ (j - k) = m + k - Suc\ j$
by *arith*

lemma *diff-Suc-diff-eq2* [*simp*]: $k \leq j$ ==> $Suc\ (j - k) - m = Suc\ j - (k + m)$
by *arith*

Lemmas for ex/Factorization

lemma *one-less-mult*: [| $Suc\ 0 < n$; $Suc\ 0 < m$ |] ==> $Suc\ 0 < m*n$
by (*cases m*) *auto*

lemma *n-less-m-mult-n*: [| $Suc\ 0 < n$; $Suc\ 0 < m$ |] ==> $n<m*n$
by (*cases m*) *auto*

lemma *n-less-n-mult-m*: [| $Suc\ 0 < n$; $Suc\ 0 < m$ |] ==> $n<n*m$
by (*cases m*) *auto*

Specialized induction principles that work "backwards":

lemma *inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: $i <= j$
assumes *base*: $P\ j$
assumes *step*: !!i. [| $i < j$; $P\ (Suc\ i)$ |] ==> $P\ i$
shows $P\ i$
using *less*
proof (*induct d*==$j - i$ *arbitrary*: i)
case (*0 i*)
hence $i = j$ **by** *simp*

with *base* **show** *?case* **by** *simp*
next
case (*Suc d i*)
hence $i < j$ *P* (*Suc i*)
by *simp-all*
thus *P i* **by** (*rule step*)
qed

lemma *strict-inc-induct*[*consumes 1*, *case-names base step*]:
assumes *less*: $i < j$
assumes *base*: !!*i*. *j* = *Suc i* ==> *P i*
assumes *step*: !!*i*. [| $i < j$; *P* (*Suc i*) |] ==> *P i*
shows *P i*
using *less*
proof (*induct d*==$j - i - 1$ *arbitrary*: *i*)
case (*0 i*)
with ‹$i < j$› **have** *j* = *Suc i* **by** *simp*
with *base* **show** *?case* **by** *simp*
next
case (*Suc d i*)
hence $i < j$ *P* (*Suc i*)
by *simp-all*
thus *P i* **by** (*rule step*)
qed

lemma *zero-induct-lemma*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P* ($k - i$)
using *inc-induct*[*of* $k - i$ *k P*, *simplified*] **by** *blast*

lemma *zero-induct*: *P k* ==> (!!*n*. *P* (*Suc n*) ==> *P n*) ==> *P 0*
using *inc-induct*[*of 0 k P*] **by** *blast*

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.

15.9 The divides relation on *nat*

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-antisym*: [| *m dvd n*; *n dvd m* |] ==> *m* = (*n*::*nat*)
unfolding *dvd-def*
by (*force dest*: *mult-eq-self-implies-10 simp add*: *mult-assoc*)

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-antisym*)

lemma *dvd-diff-nat*[*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** *dvd-diff-nat*, *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 dvd-diff-nat*)
apply (*rule dvd-refl*)
done

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*:
fixes *m n* :: *nat*
shows *0 < m* ⟹ *m < n* ⟹ ¬ *n dvd m*
by (*auto elim*!: *dvdE*) (*auto simp add*: *gr0-conv-Suc*)

15.10 size of a datatype value

class *size* =
fixes *size* :: *'a* ⇒ *nat* — see further theory *Wellfounded*

15.11 code module namespace

code-modulename *SML*
Nat Arith

code-modulename *OCaml*
Nat Arith

code-modulename *Haskell*
Nat Arith

end

16 Datatype: Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

theory *Datatype*
imports *Product-Type Sum-Type Nat*
uses
(*Tools/Datatype/datatype.ML*)
(*Tools/inductive-realizer.ML*)
(*Tools/Datatype/datatype-realizer.ML*)
begin

16.1 The datatype universe

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: $Push\text{-}Node == (\%n\ x.\ Abs\text{-}Node\ (apfst\ (Push\ n)\ (Rep\text{-}Node\ x)))$

Push-def: $Push == (\%b\ h.\ nat\text{-}case\ b\ h)$

Atom-def: $Atom == (\%x.\ \{Abs\text{-}Node((\%k.\ Inr\ 0, x))\})$
Scons-def: $Scons\ M\ N == (Push\text{-}Node\ (Inr\ 1)\ `\ M)\ Un\ (Push\text{-}Node\ (Inr\ (Suc\ 1))\ `\ N)$

Leaf-def: $Leaf == Atom\ o\ Inl$
Numb-def: $Numb == Atom\ o\ Inr$

In0-def: $In0(M) == Scons\ (Numb\ 0)\ M$
In1-def: $In1(M) == Scons\ (Numb\ 1)\ M$

Lim-def: $Lim\ f == Union\ \{z.\ ?\ x.\ z = Push\text{-}Node\ (Inl\ x)\ `\ (f\ x)\}$

ndepth-def: $ndepth(n) == (\%(f,x).\ LEAST\ k.\ f\ k = Inr\ 0)\ (Rep\text{-}Node\ n)$
ntrunc-def: $ntrunc\ k\ N == \{n.\ n{:}N\ \&\ ndepth(n){<}k\}$

uprod-def: $uprod\ A\ B == UN\ x{:}A.\ UN\ y{:}B.\ \{\ Scons\ x\ y\ \}$
usum-def: $usum\ A\ B == In0`A\ Un\ In1`B$

Split-def: $Split\ c\ M == THE\ u.\ EX\ x\ y.\ M = Scons\ x\ y\ \&\ u = c\ x\ y$

Case-def: $Case\ c\ d\ M == THE\ u.\ (EX\ x\ .\ M = In0(x)\ \&\ u = c(x))$
$\quad | (EX\ y\ .\ M = In1(y)\ \&\ u = d(y))$

dprod-def: $dprod\ r\ s == UN\ (x,x'){:}r.\ UN\ (y,y'){:}s.\ \{(Scons\ x\ y,\ Scons\ x'\ y')\}$

dsum-def: $dsum\ r\ s == (UN\ (x,x'){:}r.\ \{(In0(x),In0(x'))\})\ Un$
$\quad (UN\ (y,y'){:}s.\ \{(In1(y),In1(y'))\})$

lemma *apfst-convE*:
$[|\ q = apfst\ f\ p;\ !!x\ y.\ [|\ p = (x,y);\ q = (f(x),y)\ |] ==> R\ |] ==> R$
by (*force simp add*: *apfst-def*)

lemma *Push-inject1*: $Push\ i\ f = Push\ j\ g\ ==> i{=}j$
apply (*simp add*: *Push-def expand-fun-eq*)
apply (*drule-tac x=0* **in** *spec*, *simp*)
done

lemma *Push-inject2*: $Push\ i\ f = Push\ j\ g\ ==> f{=}g$
apply (*auto simp add*: *Push-def expand-fun-eq*)
apply (*drule-tac x=Suc x* **in** *spec*, *simp*)
done

lemma *Push-inject*:

[| *Push i f* =*Push j g*; [| *i*=*j*; *f*=*g* |] ==> *P* |] ==> *P*
by (*blast dest*: *Push-inject1 Push-inject2*)

lemma *Push-neq-K0*: *Push* (*Inr* (*Suc k*)) *f* = (%*z*. *Inr 0*) ==> *P*
by (*auto simp add*: *Push-def expand-fun-eq split*: *nat.split-asm*)

lemmas *Abs-Node-inj* = *Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*, *standard*]

lemma *Node-K0-I*: (%*k*. *Inr 0*, *a*) : *Node*
by (*simp add*: *Node-def*)

lemma *Node-Push-I*: *p*: *Node* ==> *apfst* (*Push i*) *p* : *Node*
apply (*simp add*: *Node-def Push-def*)
apply (*fast intro*!: *apfst-conv nat-case-Suc* [*THEN trans*])
done

16.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N* ≠ *Atom*(*a*)
unfolding *Atom-def Scons-def Push-Node-def One-nat-def*
by (*blast intro*: *Node-K0-I Rep-Node* [*THEN Node-Push-I*]
dest!: *Abs-Node-inj*
elim!: *apfst-convE sym* [*THEN Push-neq-K0*])

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'*
unfolding *Scons-def One-nat-def*
by (*blast dest*!: *Push-Node-inject*)

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M' N'* ==> *N*<=*N'*
unfolding *Scons-def One-nat-def*
by (*blast dest*!: *Push-Node-inject*)

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*)
unfolding *Leaf-def o-def* **by** (*rule 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*)
unfolding *Numb-def o-def* **by** (*rule 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*)
unfolding *Leaf-def o-def* **by** (*rule ntrunc-Atom*)

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
unfolding *Numb-def o-def* **by** (*rule ntrunc-Atom*)

lemma *ntrunc-Scons* [*simp*]:
ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)
unfolding *Scons-def ntrunc-def One-nat-def*
by (*auto simp add*: *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*)

16.3 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) $\neq$ *In1*(N)
unfolding *In0-def In1-def One-nat-def* **by** *auto*

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 Scons-mono*)

lemma *In1-mono*: *M<=N ==> In1*(*M*) *<= In1*(*N*)

by (*simp add*: *In1-def Scons-mono*)

lemma *Split* [*simp*]: *Split c* (*Scons M N*) = *c M N*
by (*simp add*: *Split-def*)

lemma *Case-In0* [*simp*]: *Case c d* (*In0 M*) = *c*(*M*)
by (*simp add*: *Case-def*)

lemma *Case-In1* [*simp*]: *Case c d* (*In1 N*) = *d*(*N*)
by (*simp add*: *Case-def*)

lemma *ntrunc-UN1*: *ntrunc k* (*UN x. f*(*x*)) = (*UN x. ntrunc k* (*f x*))
by (*simp add*: *ntrunc-def*, *blast*)

lemma *Scons-UN1-x*: *Scons* (*UN x. f x*) *M* = (*UN x. Scons* (*f x*) *M*)
by (*simp add*: *Scons-def*, *blast*)

lemma *Scons-UN1-y*: *Scons M* (*UN x. f x*) = (*UN x. Scons M* (*f x*))
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-UN1*: *In0*(*UN x. f*(*x*)) = (*UN x. In0*(*f*(*x*)))
by (*simp add*: *In0-def Scons-UN1-y*)

lemma *In1-UN1*: *In1*(*UN x. f*(*x*)) = (*UN x. In1*(*f*(*x*)))
by (*simp add*: *In1-def Scons-UN1-y*)

lemma *dprodI* [*intro*!]:
[| (*M*,*M'*):*r*; (*N*,*N'*):*s* |] ==> (*Scons M N*, *Scons M' N'*) : *dprod r s*
by (*auto simp add*: *dprod-def*)

lemma *dprodE* [*elim*!]:
[| *c* : *dprod r s*;
!!*x y x' y'*. [| (*x*,*x'*) : *r*; (*y*,*y'*) : *s*;
c = (*Scons x y*, *Scons x' y'*) |] ==> *P*
|] ==> *P*
by (*auto simp add*: *dprod-def*)

lemma *dsum-In0I* [*intro*]: (*M*,*M'*):*r* ==> (*In0*(*M*), *In0*(*M'*)) : *dsum r s*
by (*auto simp add*: *dsum-def*)

lemma *dsum-In1I* [*intro*]: (*N*,*N'*):*s* ==> (*In1*(*N*), *In1*(*N'*)) : *dsum r s*
by (*auto simp add*: *dsum-def*)

lemma *dsumE* [*elim*!]:
[| *w* : *dsum r s*;
!!*x x'*. [| (*x*,*x'*) : *r*; *w* = (*In0*(*x*), *In0*(*x'*)) |] ==> *P*;
!!*y y'*. [| (*y*,*y'*) : *s*; *w* = (*In1*(*y*), *In1*(*y'*)) |] ==> *P*
|] ==> *P*
by (*auto simp add*: *dsum-def*)

lemma *dprod-mono*: [| *r*<=*r'*; *s*<=*s'* |] ==> *dprod r s* <= *dprod r' s'*
by *blast*

lemma *dsum-mono*: [| *r*<=*r'*; *s*<=*s'* |] ==> *dsum r s* <= *dsum r' s'*
by *blast*

lemma *dprod-Sigma*: (*dprod* (*A* <*>* *B*) (*C* <*>* *D*)) <= (*uprod A C*) <*> (*uprod B D*)
by *blast*

lemmas *dprod-subset-Sigma* = *subset-trans* [*OF dprod-mono dprod-Sigma*, *standard*]

lemma *dprod-subset-Sigma2*:
(*dprod* (*Sigma A B*) (*Sigma C D*)) <=
Sigma (*uprod A C*) (*Split* (%*x y*. *uprod* (*B x*) (*D y*)))
by *auto*

lemma *dsum-Sigma*: (*dsum* (*A* <*>* *B*) (*C* <*>* *D*)) <= (*usum A C*) <*> (*usum B D*)
by *blast*

lemmas *dsum-subset-Sigma* = *subset-trans* [*OF dsum-mono dsum-Sigma*, *standard*]

hides popular names

hide-type (**open**) *node item*

hide-const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

use *Tools/Datatype/datatype.ML*

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

use *Tools/Datatype/datatype-realizer.ML*
setup *Datatype-Realizer.setup*

end

17 Record: Extensible records with structural subtyping

theory *Record*
imports *Datatype*
uses (*Tools/record.ML*)
begin

17.1 Introduction

Records are isomorphic to compound tuple types. To implement efficient records, we make this isomorphism explicit. Consider the record access/update simplification *alpha* (*beta-update f rec*) = *alpha rec* for distinct fields alpha and beta of some record rec with n fields. There are n ^ *2* such theorems, which prohibits storage of all of them for large n. The rules can be proved on the fly by case decomposition and simplification in O(n) time. By creating O(n) isomorphic-tuple types while defining the record, however, we can prove the access/update simplification in $O(log(n)$ ^*2*$)$ time.

The O(n) cost of case decomposition is not because O(n) steps are taken, but rather because the resulting rule must contain O(n) new variables and an O(n) size concrete record construction. To sidestep this cost, we would like to avoid case decomposition in proving access/update theorems.

Record types are defined as isomorphic to tuple types. For instance, a record type with fields $'a$, $'b$, $'c$ and $'d$ might be introduced as isomorphic to $'a \times ('b \times ('c \times 'd))$. If we balance the tuple tree to $('a \times 'b) \times ('c \times 'd)$ then accessors can be defined by converting to the underlying type then using O(log(n)) fst or snd operations. Updators can be defined similarly, if we introduce a *fst-update* and *snd-update* function. Furthermore, we can prove the access/update theorem in O(log(n)) steps by using simple rewrites on fst, snd, *fst-update* and *snd-update*.

The catch is that, although O(log(n)) steps were taken, the underlying type we converted to is a tuple tree of size O(n). Processing this term type wastes

performance. We avoid this for large n by taking each subtree of size K and defining a new type isomorphic to that tuple subtree. A record can now be defined as isomorphic to a tuple tree of these O(n/K) new types, or, if $n > K*K$, we can repeat the process, until the record can be defined in terms of a tuple tree of complexity less than the constant K.

If we prove the access/update theorem on this type with the analagous steps to the tuple tree, we consume $O(log(n)\hat{}2)$ time as the intermediate terms are $O(log(n))$ in size and the types needed have size bounded by K. To enable this analagous traversal, we define the functions seen below: *iso-tuple-fst*, *iso-tuple-snd*, *iso-tuple-fst-update* and *iso-tuple-snd-update*. These functions generalise tuple operations by taking a parameter that encapsulates a tuple isomorphism. The rewrites needed on these functions now need an additional assumption which is that the isomorphism works.

These rewrites are typically used in a structured way. They are here presented as the introduction rule *isomorphic-tuple.intros* rather than as a rewrite rule set. The introduction form is an optimisation, as net matching can be performed at one term location for each step rather than the simplifier searching the term for possible pattern matches. The rule set is used as it is viewed outside the locale, with the locale assumption (that the isomorphism is valid) left as a rule assumption. All rules are structured to aid net matching, using either a point-free form or an encapsulating predicate.

17.2 Operators and lemmas for types isomorphic to tuples

datatype $('a, 'b, 'c)$ *tuple-isomorphism* =
  Tuple-Isomorphism $'a \Rightarrow 'b \times 'c$ $'b \times 'c \Rightarrow 'a$

primrec
  repr :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'a \Rightarrow 'b \times 'c$ **where**
  repr (*Tuple-Isomorphism r a*) = *r*

primrec
  abst :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'b \times 'c \Rightarrow 'a$ **where**
  abst (*Tuple-Isomorphism r a*) = *a*

definition
  iso-tuple-fst :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'a \Rightarrow 'b$ **where**
  iso-tuple-fst isom = *fst* ∘ *repr isom*

definition
  iso-tuple-snd :: $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow 'a \Rightarrow 'c$ **where**
  iso-tuple-snd isom = *snd* ∘ *repr isom*

definition
  iso-tuple-fst-update ::
    $('a, 'b, 'c)$ *tuple-isomorphism* $\Rightarrow ('b \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'a)$ **where**

iso-tuple-fst-update isom f = abst isom ∘ apfst f ∘ repr isom

definition
iso-tuple-snd-update ::
('a, 'b, 'c) *tuple-isomorphism* ⇒ ('c ⇒ 'c) ⇒ ('a ⇒ 'a) **where**
iso-tuple-snd-update isom f = abst isom ∘ apsnd f ∘ repr isom

definition
iso-tuple-cons ::
('a, 'b, 'c) *tuple-isomorphism* ⇒ 'b ⇒ 'c ⇒ 'a **where**
iso-tuple-cons isom = curry (abst isom)

17.3 Logical infrastructure for records

definition
iso-tuple-surjective-proof-assist :: 'a ⇒ 'b ⇒ ('a ⇒ 'b) ⇒ *bool* **where**
iso-tuple-surjective-proof-assist x y f ⟷ f x = y

definition
iso-tuple-update-accessor-cong-assist ::
(('b ⇒ 'b) ⇒ ('a ⇒ 'a)) ⇒ ('a ⇒ 'b) ⇒ *bool* **where**
iso-tuple-update-accessor-cong-assist upd acc ⟷
(∀ f v. *upd* (λx. f (*acc* v)) v = *upd* f v) ∧ (∀ v. *upd id* v = v)

definition
iso-tuple-update-accessor-eq-assist ::
(('b ⇒ 'b) ⇒ ('a ⇒ 'a)) ⇒ ('a ⇒ 'b) ⇒ 'a ⇒ ('b ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ *bool*
where
iso-tuple-update-accessor-eq-assist upd acc v f v' x ⟷
upd f v = v' ∧ acc v = x ∧ iso-tuple-update-accessor-cong-assist upd acc

lemma *update-accessor-congruence-foldE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd acc*
and *r*: *r = r'* **and** *v*: *acc r' = v'*
and *f*: ⋀v. *v' = v ⟹ f v = f' v*
shows *upd f r = upd f' r'*
using *uac r v* [*symmetric*]
apply (*subgoal-tac upd* (λx. *f* (*acc r'*)) *r' = upd* (λx. *f'* (*acc r'*)) *r'*)
apply (*simp add*: *iso-tuple-update-accessor-cong-assist-def*)
apply (*simp add*: *f*)
done

lemma *update-accessor-congruence-unfoldE*:
iso-tuple-update-accessor-cong-assist upd acc ⟹
r = r' ⟹ acc r' = v' ⟹ (⋀v. *v = v' ⟹ f v = f' v*) *⟹*
upd f r = upd f' r'
apply (*erule*(2) *update-accessor-congruence-foldE*)
apply *simp*
done

lemma *iso-tuple-update-accessor-cong-assist-id*:
iso-tuple-update-accessor-cong-assist upd acc $\Longrightarrow$ *upd id* = *id*
by *rule* (*simp add*: *iso-tuple-update-accessor-cong-assist-def*)

lemma *update-accessor-noopE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd acc*
and *acc*: *f* (*acc x*) = *acc x*
shows *upd f x* = *x*
using *uac*
by (*simp add*: *acc iso-tuple-update-accessor-cong-assist-id* [*OF uac*, *unfolded id-def*]
cong: *update-accessor-congruence-unfoldE* [*OF uac*])

lemma *update-accessor-noop-compE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd acc*
and *acc*: *f* (*acc x*) = *acc x*
shows *upd* (*g* ∘ *f*) *x* = *upd g x*
by (*simp add*: *acc cong*: *update-accessor-congruence-unfoldE*[*OF uac*])

lemma *update-accessor-cong-assist-idI*:
iso-tuple-update-accessor-cong-assist id id
by (*simp add*: *iso-tuple-update-accessor-cong-assist-def*)

lemma *update-accessor-cong-assist-triv*:
iso-tuple-update-accessor-cong-assist upd acc $\Longrightarrow$
iso-tuple-update-accessor-cong-assist upd acc
by *assumption*

lemma *update-accessor-accessor-eqE*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x $\Longrightarrow$ *acc v* = *x*
by (*simp add*: *iso-tuple-update-accessor-eq-assist-def*)

lemma *update-accessor-updator-eqE*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x $\Longrightarrow$ *upd f v* = *v'*
by (*simp add*: *iso-tuple-update-accessor-eq-assist-def*)

lemma *iso-tuple-update-accessor-eq-assist-idI*:
v' = *f v* $\Longrightarrow$ *iso-tuple-update-accessor-eq-assist id id v f v' v*
by (*simp add*: *iso-tuple-update-accessor-eq-assist-def update-accessor-cong-assist-idI*)

lemma *iso-tuple-update-accessor-eq-assist-triv*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x $\Longrightarrow$
iso-tuple-update-accessor-eq-assist upd acc v f v' x
by *assumption*

lemma *iso-tuple-update-accessor-cong-from-eq*:
iso-tuple-update-accessor-eq-assist upd acc v f v' x $\Longrightarrow$
iso-tuple-update-accessor-cong-assist upd acc

by (*simp add*: *iso-tuple-update-accessor-eq-assist-def*)

lemma *iso-tuple-surjective-proof-assistI*:
f x = *y* ⟹ *iso-tuple-surjective-proof-assist x y f*
by (*simp add*: *iso-tuple-surjective-proof-assist-def*)

lemma *iso-tuple-surjective-proof-assist-idE*:
iso-tuple-surjective-proof-assist x y id ⟹ *x* = *y*
by (*simp add*: *iso-tuple-surjective-proof-assist-def*)

locale *isomorphic-tuple* =
fixes *isom* :: ('*a*, '*b*, '*c*) *tuple-isomorphism*
assumes *repr-inv*: ⋀*x*. *abst isom* (*repr isom x*) = *x*
and *abst-inv*: ⋀*y*. *repr isom* (*abst isom y*) = *y*
begin

lemma *repr-inj*: *repr isom x* = *repr isom y* ⟷ *x* = *y*
by (*auto dest*: *arg-cong* [*of repr isom x repr isom y abst isom*]
simp add: *repr-inv*)

lemma *abst-inj*: *abst isom x* = *abst isom y* ⟷ *x* = *y*
by (*auto dest*: *arg-cong* [*of abst isom x abst isom y repr isom*]
simp add: *abst-inv*)

lemmas *simps* = *Let-def repr-inv abst-inv repr-inj abst-inj*

lemma *iso-tuple-access-update-fst-fst*:
f o h g = *j o f* ⟹
(*f o iso-tuple-fst isom*) *o* (*iso-tuple-fst-update isom o h*) *g* =
j o (*f o iso-tuple-fst isom*)
by (*clarsimp simp*: *iso-tuple-fst-update-def iso-tuple-fst-def simps*
intro!: *ext elim*!: *o-eq-elim*)

lemma *iso-tuple-access-update-snd-snd*:
f o h g = *j o f* ⟹
(*f o iso-tuple-snd isom*) *o* (*iso-tuple-snd-update isom o h*) *g* =
j o (*f o iso-tuple-snd isom*)
by (*clarsimp simp*: *iso-tuple-snd-update-def iso-tuple-snd-def simps*
intro!: *ext elim*!: *o-eq-elim*)

lemma *iso-tuple-access-update-fst-snd*:
(*f o iso-tuple-fst isom*) *o* (*iso-tuple-snd-update isom o h*) *g* =
id o (*f o iso-tuple-fst isom*)
by (*clarsimp simp*: *iso-tuple-snd-update-def iso-tuple-fst-def simps*
intro!: *ext elim*!: *o-eq-elim*)

lemma *iso-tuple-access-update-snd-fst*:
(*f o iso-tuple-snd isom*) *o* (*iso-tuple-fst-update isom o h*) *g* =
id o (*f o iso-tuple-snd isom*)

by (*clarsimp simp*: *iso-tuple-fst-update-def iso-tuple-snd-def simps intro*!: *ext elim*!: *o-eq-elim*)

lemma *iso-tuple-update-swap-fst-fst*:
h f o j g = j g o h f ⟹
(*iso-tuple-fst-update isom o h*) *f o* (*iso-tuple-fst-update isom o j*) *g* =
(*iso-tuple-fst-update isom o j*) *g o* (*iso-tuple-fst-update isom o h*) *f*
by (*clarsimp simp*: *iso-tuple-fst-update-def simps apfst-compose intro*!: *ext*)

lemma *iso-tuple-update-swap-snd-snd*:
h f o j g = j g o h f ⟹
(*iso-tuple-snd-update isom o h*) *f o* (*iso-tuple-snd-update isom o j*) *g* =
(*iso-tuple-snd-update isom o j*) *g o* (*iso-tuple-snd-update isom o h*) *f*
by (*clarsimp simp*: *iso-tuple-snd-update-def simps apsnd-compose intro*!: *ext*)

lemma *iso-tuple-update-swap-fst-snd*:
(*iso-tuple-snd-update isom o h*) *f o* (*iso-tuple-fst-update isom o j*) *g* =
(*iso-tuple-fst-update isom o j*) *g o* (*iso-tuple-snd-update isom o h*) *f*
by (*clarsimp simp*: *iso-tuple-fst-update-def iso-tuple-snd-update-def simps intro*!: *ext*)

lemma *iso-tuple-update-swap-snd-fst*:
(*iso-tuple-fst-update isom o h*) *f o* (*iso-tuple-snd-update isom o j*) *g* =
(*iso-tuple-snd-update isom o j*) *g o* (*iso-tuple-fst-update isom o h*) *f*
by (*clarsimp simp*: *iso-tuple-fst-update-def iso-tuple-snd-update-def simps intro*!: *ext*)

lemma *iso-tuple-update-compose-fst-fst*:
h f o j g = k (*f o g*) ⟹
(*iso-tuple-fst-update isom o h*) *f o* (*iso-tuple-fst-update isom o j*) *g* =
(*iso-tuple-fst-update isom o k*) (*f o g*)
by (*clarsimp simp*: *iso-tuple-fst-update-def simps apfst-compose intro*!: *ext*)

lemma *iso-tuple-update-compose-snd-snd*:
h f o j g = k (*f o g*) ⟹
(*iso-tuple-snd-update isom o h*) *f o* (*iso-tuple-snd-update isom o j*) *g* =
(*iso-tuple-snd-update isom o k*) (*f o g*)
by (*clarsimp simp*: *iso-tuple-snd-update-def simps apsnd-compose intro*!: *ext*)

lemma *iso-tuple-surjective-proof-assist-step*:
iso-tuple-surjective-proof-assist v a (*iso-tuple-fst isom o f*) ⟹
iso-tuple-surjective-proof-assist v b (*iso-tuple-snd isom o f*) ⟹
iso-tuple-surjective-proof-assist v (*iso-tuple-cons isom a b*) *f*
by (*clarsimp simp*: *iso-tuple-surjective-proof-assist-def simps iso-tuple-fst-def iso-tuple-snd-def iso-tuple-cons-def*)

lemma *iso-tuple-fst-update-accessor-cong-assist*:
assumes *iso-tuple-update-accessor-cong-assist f g*
shows *iso-tuple-update-accessor-cong-assist*

```
    (iso-tuple-fst-update isom o f) (g o iso-tuple-fst isom)
proof −
  from assms have f id = id
    by (rule iso-tuple-update-accessor-cong-assist-id)
  with assms show ?thesis
    by (clarsimp simp: iso-tuple-update-accessor-cong-assist-def simps
      iso-tuple-fst-update-def iso-tuple-fst-def)
qed

lemma iso-tuple-snd-update-accessor-cong-assist:
  assumes iso-tuple-update-accessor-cong-assist f g
  shows iso-tuple-update-accessor-cong-assist
    (iso-tuple-snd-update isom o f) (g o iso-tuple-snd isom)
proof −
  from assms have f id = id
    by (rule iso-tuple-update-accessor-cong-assist-id)
  with assms show ?thesis
    by (clarsimp simp: iso-tuple-update-accessor-cong-assist-def simps
      iso-tuple-snd-update-def iso-tuple-snd-def)
qed

lemma iso-tuple-fst-update-accessor-eq-assist:
  assumes iso-tuple-update-accessor-eq-assist f g a u a' v
  shows iso-tuple-update-accessor-eq-assist
    (iso-tuple-fst-update isom o f) (g o iso-tuple-fst isom)
    (iso-tuple-cons isom a b) u (iso-tuple-cons isom a' b) v
proof −
  from assms have f id = id
    by (auto simp add: iso-tuple-update-accessor-eq-assist-def
      intro: iso-tuple-update-accessor-cong-assist-id)
  with assms show ?thesis
    by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
      iso-tuple-fst-update-def iso-tuple-fst-def
      iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)
qed

lemma iso-tuple-snd-update-accessor-eq-assist:
  assumes iso-tuple-update-accessor-eq-assist f g b u b' v
  shows iso-tuple-update-accessor-eq-assist
    (iso-tuple-snd-update isom o f) (g o iso-tuple-snd isom)
    (iso-tuple-cons isom a b) u (iso-tuple-cons isom a b') v
proof −
  from assms have f id = id
    by (auto simp add: iso-tuple-update-accessor-eq-assist-def
      intro: iso-tuple-update-accessor-cong-assist-id)
  with assms show ?thesis
    by (clarsimp simp: iso-tuple-update-accessor-eq-assist-def
      iso-tuple-snd-update-def iso-tuple-snd-def
      iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)
```

qed

lemma *iso-tuple-cons-conj-eqI*:
  $a = c \wedge b = d \wedge P \longleftrightarrow Q \Longrightarrow$
    iso-tuple-cons isom a b = *iso-tuple-cons isom c d* $\wedge$ $P \longleftrightarrow Q$
  by (*clarsimp simp*: *iso-tuple-cons-def simps*)

lemmas *intros* =
  iso-tuple-access-update-fst-fst
  iso-tuple-access-update-snd-snd
  iso-tuple-access-update-fst-snd
  iso-tuple-access-update-snd-fst
  iso-tuple-update-swap-fst-fst
  iso-tuple-update-swap-snd-snd
  iso-tuple-update-swap-fst-snd
  iso-tuple-update-swap-snd-fst
  iso-tuple-update-compose-fst-fst
  iso-tuple-update-compose-snd-snd
  iso-tuple-surjective-proof-assist-step
  iso-tuple-fst-update-accessor-eq-assist
  iso-tuple-snd-update-accessor-eq-assist
  iso-tuple-fst-update-accessor-cong-assist
  iso-tuple-snd-update-accessor-cong-assist
  iso-tuple-cons-conj-eqI

end

lemma *isomorphic-tuple-intro*:
  fixes *repr abst*
  assumes *repr-inj*: $\bigwedge x\ y.$ *repr* x = *repr* $y \longleftrightarrow x = y$
    and *abst-inv*: $\bigwedge z.$ *repr* (*abst* z) $= z$
    and v: $v \equiv$ *Tuple-Isomorphism repr abst*
  shows *isomorphic-tuple v*
proof
  fix x **have** *repr* (*abst* (*repr* x)) = *repr* x
    by (*simp add*: *abst-inv*)
  then show *Record.abst v* (*Record.repr v* x) $= x$
    by (*simp add*: v *repr-inj*)
next
  fix y
  show *Record.repr v* (*Record.abst v* y) $= y$
    by (*simp add*: v) (*fact abst-inv*)
qed

definition
  tuple-iso-tuple $\equiv$ *Tuple-Isomorphism id id*

lemma *tuple-iso-tuple*:
  isomorphic-tuple tuple-iso-tuple

by (*simp add*: *isomorphic-tuple-intro* [*OF - - reflexive*] *tuple-iso-tuple-def*)

lemma *refl-conj-eq*: $Q = R \Longrightarrow P \wedge Q \longleftrightarrow P \wedge R$
by *simp*

lemma *iso-tuple-UNIV-I*: $x \in UNIV \equiv True$
by *simp*

lemma *iso-tuple-True-simp*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
by *simp*

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
by *simp*

lemma *K-record-comp*: $(\lambda x.\ c) \circ f = (\lambda x.\ c)$
by (*simp add*: *comp-def*)

lemma *o-eq-dest-lhs*: $a\ o\ b = c \Longrightarrow a\ (b\ v) = c\ v$
by *clarsimp*

lemma *o-eq-id-dest*: $a\ o\ b = id\ o\ c \Longrightarrow a\ (b\ v) = c\ v$
by *clarsimp*

17.4 Concrete record syntax

nonterminals
ident field-type field-types field fields field-update field-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*... =/ -) |')))

-field-update :: *ident* => 'a => *field-update* ((*2*- :=/ -))
:: *field-update* => *field-updates* (-)
-field-updates :: *field-update* => *field-updates* => *field-updates* (-,/ -)
-record-update :: 'a => *field-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* => *field-updates* => *'b* (-/(*3*(|-|)) [*900*, *0*] *900*)

17.5 Record package

use *Tools/record.ML*
setup *Record.setup*

hide-const (**open**) *Tuple-Isomorphism repr abst iso-tuple-fst iso-tuple-snd iso-tuple-fst-update iso-tuple-snd-update iso-tuple-cons iso-tuple-surjective-proof-assist iso-tuple-update-accessor-cong-assist iso-tuple-update-accessor-eq-assist tuple-iso-tuple*

end

18 Power: Exponentiation

theory *Power*
imports *Nat*
begin

18.1 Powers for Arbitrary Monoids

class *power* = *one* + *times*
begin

primrec *power* :: *'a* ⇒ *nat* ⇒ *'a* (**infixr** ^ *80*) **where**
  power-0: *a* ^ *0* = *1*
| *power-Suc*: *a* ^ *Suc n* = *a* ∗ *a* ^ *n*

notation (*latex* **output**)
  power ((-[^]) [*1000*] *1000*)

notation (*HTML* **output**)
  power ((-[^]) [*1000*] *1000*)

end

context *monoid-mult*
begin

subclass *power* **..**

lemma *power-one* [*simp*]:
  1 ^ *n* = *1*

by (*induct n*) *simp-all*

lemma *power-one-right* [*simp*]:
a ^ 1 = a
by *simp*

lemma *power-commutes*:
*a ^ n * a = a * a ^ n*
by (*induct n*) (*simp-all add*: *mult-assoc*)

lemma *power-Suc2*:
*a ^ Suc n = a ^ n * a*
by (*simp add*: *power-commutes*)

lemma *power-add*:
*a ^ (m + n) = a ^ m * a ^ n*
by (*induct m*) (*simp-all add*: *algebra-simps*)

lemma *power-mult*:
*a ^ (m * n) = (a ^ m) ^ n*
by (*induct n*) (*simp-all add*: *power-add*)

end

context *comm-monoid-mult*
begin

lemma *power-mult-distrib*:
*(a * b) ^ n = (a ^ n) * (b ^ n)*
by (*induct n*) (*simp-all add*: *mult-ac*)

end

context *semiring-1*
begin

lemma *of-nat-power*:
of-nat (m ^ n) = of-nat m ^ n
by (*induct n*) (*simp-all add*: *of-nat-mult*)

end

context *comm-semiring-1*
begin

The divides relation

lemma *le-imp-power-dvd*:
assumes $m \leq n$ **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:
  a ^ n dvd b ⟹ m ≤ n ⟹ a ^ m dvd b
  by (rule dvd-trans [OF le-imp-power-dvd])

lemma dvd-power-same:
  x dvd y ⟹ x ^ n dvd y ^ n
  by (induct n) (auto simp add: mult-dvd-mono)

lemma dvd-power-le:
  x dvd y ⟹ m ≥ n ⟹ x ^ n dvd y ^ m
  by (rule power-le-dvd [OF dvd-power-same])

lemma dvd-power [simp]:
  assumes n > (0::nat) ∨ x = 1
  shows x dvd (x ^ n)
using assms proof
  assume 0 < n
  then have x ^ n = x ^ Suc (n - 1) by simp
  then show x dvd (x ^ n) by simp
next
  assume x = 1
  then show x dvd (x ^ n) by simp
qed

end

context ring-1
begin

lemma power-minus:
  (- a) ^ n = (- 1) ^ n * a ^ 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

end

context linordered-semidom
begin
```

lemma *zero-less-power* [*simp*]:
0 < a ⟹ 0 < a ^ n
by (*induct n*) (*simp-all add*: *mult-pos-pos*)

lemma *zero-le-power* [*simp*]:
0 ≤ a ⟹ 0 ≤ a ^ n
by (*induct n*) (*simp-all add*: *mult-nonneg-nonneg*)

lemma *one-le-power*[*simp*]:
1 ≤ a ⟹ 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 *power-gt1-lemma*:
assumes *gt1*: *1 < a*
shows *1 < a * a ^ n*
proof −
from *gt1* **have** *0 ≤ a*
by (*fact order-trans* [*OF zero-le-one less-imp-le*])
have *1 * 1 < a * 1* **using** *gt1* **by** *simp*
also have *... ≤ a * a ^ n* **using** *gt1*
by (*simp only*: *mult-mono ‹0 ≤ a› one-le-power order-less-imp-le*
zero-le-one order-refl)
finally show *?thesis* **by** *simp*
qed

lemma *power-gt1*:
1 < a ⟹ 1 < a ^ Suc n
by (*simp add*: *power-gt1-lemma*)

lemma *one-less-power* [*simp*]:
1 < a ⟹ 0 < n ⟹ 1 < a ^ n
by (*cases n*) (*simp-all add*: *power-gt1-lemma*)

lemma *power-le-imp-le-exp*:
assumes *gt1*: *1 < a*
shows *a ^ m ≤ a ^ n ⟹ m ≤ n*
proof (*induct m arbitrary*: *n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc m*)
show *?case*
proof (*cases n*)
case *0*

```
    with Suc.prems Suc.hyps have a * a ^ m ≤ 1 by simp
    with gt1 show ?thesis
      by (force simp only: power-gt1-lemma
          not-less [symmetric])
  next
    case (Suc n)
    with Suc.prems Suc.hyps show ?thesis
      by (force dest: mult-left-le-imp-le
          simp add: 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 ^ 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 ⟹ a ^ m < a ^ n ⟹ m < n
  by (simp add: order-less-le [of m n] less-le [of a^m a^n]
    power-le-imp-le-exp)

lemma power-mono:
  a ≤ b ⟹ 0 ≤ a ⟹ a ^ n ≤ b ^ n
  by (induct n)
    (auto intro: mult-mono order-trans [of 0 a b])

lemma power-strict-mono [rule-format]:
  a < b ⟹ 0 ≤ a ⟹ 0 < n ⟶ a ^ n < b ^ n
  by (induct n)
   (auto simp add: mult-strict-mono le-less-trans [of 0 a b])
```

Lemma for *power-strict-decreasing*

```
lemma power-Suc-less:
  0 < a ⟹ a < 1 ⟹ a * a ^ n < a ^ n
  by (induct n)
    (auto simp add: mult-strict-left-mono)

lemma power-strict-decreasing [rule-format]:
  n < N ⟹ 0 < a ⟹ a < 1 ⟶ a ^ N < a ^ n
proof (induct N)
  case 0 then show ?case by simp
next
  case (Suc N) then show ?case
  apply (auto simp add: power-Suc-less less-Suc-eq)
  apply (subgoal-tac a * a^N < 1 * a^n)
  apply simp
  apply (rule mult-strict-mono) apply auto
```

done
qed

Proof resembles that of *power-strict-decreasing*

lemma *power-decreasing* [*rule-format*]:
$n \leq N \Longrightarrow 0 < a \Longrightarrow a \leq 1 \longrightarrow a \hat{} N \leq a \hat{} n$
proof (*induct N*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc N*) **then show** *?case*
apply (*auto simp add*: *le-Suc-eq*)
apply (*subgoal-tac* $a * a \hat{} N \leq 1 * a \hat{} n$, *simp*)
apply (*rule mult-mono*) **apply** *auto*
done
qed

lemma *power-Suc-less-one*:
$0 < a \Longrightarrow a < 1 \Longrightarrow a \hat{} Suc\ n < 1$
using *power-strict-decreasing* [*of 0 Suc n a*] **by** *simp*

Proof again resembles that of *power-strict-decreasing*

lemma *power-increasing* [*rule-format*]:
$n \leq N \Longrightarrow 1 \leq a \Longrightarrow a \hat{} n \leq a \hat{} N$
proof (*induct N*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc N*) **then show** *?case*
apply (*auto simp add*: *le-Suc-eq*)
apply (*subgoal-tac* $1 * a \hat{} n \leq a * a \hat{} N$, *simp*)
apply (*rule mult-mono*) **apply** (*auto simp add*: *order-trans* [*OF zero-le-one*])
done
qed

Lemma for *power-strict-increasing*

lemma *power-less-power-Suc*:
$1 < a \Longrightarrow a \hat{} n < a * a \hat{} n$
by (*induct n*) (*auto simp add*: *mult-strict-left-mono less-trans* [*OF zero-less-one*])

lemma *power-strict-increasing* [*rule-format*]:
$n < N \Longrightarrow 1 < a \longrightarrow a \hat{} n < a \hat{} N$
proof (*induct N*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc N*) **then show** *?case*
apply (*auto simp add*: *power-less-power-Suc less-Suc-eq*)
apply (*subgoal-tac* $1 * a \hat{} n < a * a \hat{} N$, *simp*)
apply (*rule mult-strict-mono*) **apply** (*auto simp add*: *less-trans* [*OF zero-less-one*] *less-imp-le*)
done

qed

lemma *power-increasing-iff* [*simp*]:
1 < b ⟹ b ^ x ≤ b ^ y ⟷ x ≤ y
by (*blast intro*: *power-le-imp-le-exp power-increasing less-imp-le*)

lemma *power-strict-increasing-iff* [*simp*]:
1 < b ⟹ 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 ≤ 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*:
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
by (*blast intro*: *power-le-imp-le-base antisym eq-refl sym*)

lemma *power-eq-imp-eq-base*:
a ^ n = b ^ n ⟹ 0 ≤ a ⟹ 0 ≤ b ⟹ 0 < n ⟹ a = b
by (*cases n*) (*simp-all del*: *power-Suc*, *rule power-inject-base*)

end

context *linordered-idom*
begin

lemma *power-abs*:
abs (a ^ n) = abs a ^ n

by (*induct n*) (*auto simp add*: *abs-mult*)

lemma *abs-power-minus* [*simp*]:
abs ((−*a*) ^ *n*) = *abs* (*a* ^ *n*)
by (*simp add*: *power-abs*)

lemma *zero-less-power-abs-iff* [*simp*, *no-atp*]:
0 < *abs a* ^ *n* ⟷ *a* ≠ *0* ∨ *n* = *0*
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **show** *?case* **by** (*auto simp add*: *Suc zero-less-mult-iff*)
qed

lemma *zero-le-power-abs* [*simp*]:
0 ≤ *abs a* ^ *n*
by (*rule zero-le-power* [*OF abs-ge-zero*])

end

context *ring-1-no-zero-divisors*
begin

lemma *field-power-not-zero*:
a ≠ *0* ⟹ *a* ^ *n* ≠ *0*
by (*induct n*) *auto*

end

context *division-ring*
begin

FIXME reorient or rename to *nonzero-inverse-power*

lemma *nonzero-power-inverse*:
a ≠ *0* ⟹ *inverse* (*a* ^ *n*) = (*inverse a*) ^ *n*
by (*induct n*)
(*simp-all add*: *nonzero-inverse-mult-distrib power-commutes field-power-not-zero*)

end

context *field*
begin

lemma *nonzero-power-divide*:
b ≠ *0* ⟹ (*a* / *b*) ^ *n* = *a* ^ *n* / *b* ^ *n*
by (*simp add*: *divide-inverse power-mult-distrib nonzero-power-inverse*)

end

lemma *power-0-Suc* [*simp*]:
(*0*::*'a*::{*power*, *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*::{*power*, *semiring-0*}))
by (*induct n*) *simp-all*

lemma *power-eq-0-iff* [*simp*]:
a ^ *n* = *0* ⟷
a = (*0*::*'a*::{*mult-zero*,*zero-neq-one*,*no-zero-divisors*,*power*}) ∧ *n* ≠ *0*
by (*induct n*)
(*auto simp add*: *no-zero-divisors elim*: *contrapos-pp*)

lemma (**in** *field*) *power-diff*:
assumes *nz*: *a* ≠ *0*
shows *n* ≤ *m* ⟹ *a* ^ (*m* − *n*) = *a* ^ *m* / *a* ^ *n*
by (*induct m n rule*: *diff-induct*) (*simp-all add*: *nz field-power-not-zero*)

Perhaps these should be simprules.

lemma *power-inverse*:
fixes *a* :: *'a*::*division-ring-inverse-zero*
shows *inverse* (*a* ^ *n*) = *inverse a* ^ *n*
apply (*cases a* = *0*)
apply (*simp add*: *power-0-left*)
apply (*simp add*: *nonzero-power-inverse*)
done

lemma *power-one-over*:
1 / (*a*::*'a*::{*field-inverse-zero*, *power*}) ^ *n* = (*1* / *a*) ^ *n*
by (*simp add*: *divide-inverse*) (*rule power-inverse*)

lemma *power-divide*:
(*a* / *b*) ^ *n* = (*a*::*'a*::*field-inverse-zero*) ^ *n* / *b* ^ *n*
apply (*cases b* = *0*)
apply (*simp add*: *power-0-left*)
apply (*rule nonzero-power-divide*)
apply *assumption*
done

18.2 Exponentiation for the Natural Numbers

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 ^ m = Suc 0 ⟷ m = 0 ∨ x = Suc 0
by (*induct 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-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

18.3 Code generator tweak

lemma *power-power-power* [*code*, *code-unfold*, *code-inline del*]:
*power = power.power (1::'a::{power}) (op *)*
unfolding *power-def power.power-def* **..**

declare *power.power.simps* [*code*]

code-modulename *SML*
Power Arith

code-modulename *OCaml*
Power Arith

code-modulename *Haskell*
Power Arith

end

19 Option: Datatype option

theory *Option*
imports *Datatype*
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 *inj-Some* [*simp*]: *inj-on Some A*
by *(rule inj-onI) simp*

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 *UNIV-option-conv*: *UNIV = insert None (range Some)*
by(*auto intro*: *classical*)

19.0.1 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**
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-const (**open**) *set map*

19.0.2 Code generator setup

definition *is-none* :: '*a option* ⇒ *bool* **where**
[*code-post*]: *is-none x* ⟷ *x* = *None*

lemma *is-none-code* [*code*]:
shows *is-none None* ⟷ *True*
and *is-none* (*Some x*) ⟷ *False*
unfolding *is-none-def* **by** *simp-all*

lemma *is-none-none*:
is-none x ⟷ *x* = *None*
by (*simp add*: *is-none-def*)

lemma [*code-unfold*]:
eq-class.eq x None ⟷ *is-none x*
by (*simp add*: *eq is-none-none*)

hide-const (**open**) *is-none*

code-type *option*
(*SML* - *option*)
(*OCaml* - *option*)
(*Haskell Maybe* -)
(*Scala* !*Option*[(-)])

code-const *None* **and** *Some*
(*SML NONE* **and** *SOME*)
(*OCaml None* **and** *Some* -)
(*Haskell Nothing* **and** *Just*)
(*Scala None* **and** !*Some*((-)))

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

code-reserved *Scala*
Option None Some

end

20 Finite-Set: Finite sets

theory *Finite-Set*
imports *Power Option*
begin

20.1 Predicate for finite sets

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** $A \neq UNIV$ **by** *blast*
thus *?thesis* **by** *blast*
qed

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
finite F ==>
$P\ \{\}$ ==> (!!$x\ F$. *finite* F ==> $x \notin F$ ==> $P\ F$ ==> $P\ (insert\ x\ F)$) ==>
$P\ F$
— Discharging $x \notin F$ entails extra work.
proof −
assume $P\ \{\}$ **and**
insert: !!$x\ F$. *finite* F ==> $x \notin F$ ==> $P\ F$ ==> $P\ (insert\ x\ F)$
assume *finite* F
thus $P\ F$
proof *induct*
show $P\ \{\}$ **by** *fact*
fix $x\ F$ **assume** F: *finite* F **and** P: $P\ F$
show $P\ (insert\ x\ F)$
proof *cases*
assume $x \in F$
hence $insert\ x\ F = F$ **by** (*rule insert-absorb*)
with P **show** *?thesis* **by** (*simp only*:)
next
assume $x \notin F$
from F *this* P **show** *?thesis* **by** (*rule insert*)
qed
qed
qed

lemma *finite-ne-induct*[*case-names singleton insert*, *consumes 2*]:
assumes *fin*: *finite* F **shows** $F \neq \{\} \Longrightarrow$
$[\![\ \bigwedge x.\ P\{x\};$
$\bigwedge x\ F.\ [\![$ *finite* F; $F \neq \{\}$; $x \notin F$; $P\ F\ ]\!] \Longrightarrow P\ (insert\ x\ F)\ ]\!]$
$\Longrightarrow P\ F$
using *fin*
proof *induct*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert* $x\ F$)
show *?case*
proof *cases*
assume $F = \{\}$
thus *?thesis* **using** ‹$P\ \{x\}$› **by** *simp*
next
assume $F \neq \{\}$
thus *?thesis* **using** *insert* **by** *blast*
qed
qed

lemma *finite-subset-induct* [*consumes 2*, *case-names empty insert*]:
assumes *finite F* **and** $F \subseteq A$
and *empty*: $P\ \{\}$
and *insert*: $!!a\ F.$ *finite* F ==> $a \in A$ ==> $a \notin F$ ==> $P\ F$ ==> P (*insert* $a\ F$)
shows $P\ F$
proof −
from ‹*finite F*› **and** ‹$F \subseteq A$›
show *?thesis*
proof *induct*
show $P\ \{\}$ **by** *fact*
next
fix $x\ F$
assume *finite F* **and** $x \notin F$ **and**
P: $F \subseteq A$ ==> $P\ F$ **and** i: *insert* $x\ F \subseteq A$
show P (*insert* $x\ F$)
proof (*rule insert*)
from i **show** $x \in A$ **by** *blast*
from i **have** $F \subseteq A$ **by** *blast*
with P **show** $P\ F$ **.**
show *finite F* **by** *fact*
show $x \notin F$ **by** *fact*
qed
qed
qed

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 ∃ (n::*nat*) f. $A = f$ ‘ {i. i<n} & *inj-on f* {i. i<n}
using *fin*
proof *induct*
case *empty*
show *?case*
proof show ∃f. $\{\} = f$ ‘ {i::*nat*. $i < 0$} & *inj-on f* {i. i<0} **by** *simp*

```
  qed
next
  case (insert a A)
  have notinA: a ∉ A by fact
  from insert.hyps obtain n f
    where A = f ` {i::nat. i < n} inj-on f {i. i < n} by blast
  hence insert a A = f(n:=a) ` {i. i < Suc n}
       inj-on (f(n:=a)) {i. i < Suc n} using notinA
    by (auto simp add: image-def Ball-def inj-on-def less-Suc-eq)
  thus ?case by blast
qed

lemma nat-seg-image-imp-finite:
  !!f A. A = f ` {i::nat. i<n} ⟹ finite A
proof (induct n)
  case 0 thus ?case by simp
next
  case (Suc n)
  let ?B = f ` {i. i < n}
  have finB: finite ?B by(rule Suc.hyps[OF refl])
  show ?case
  proof cases
    assume ∃ k<n. f n = f k
    hence A = ?B using Suc.prems by(auto simp:less-Suc-eq)
    thus ?thesis using finB by simp
  next
    assume ¬(∃ k<n. f n = f k)
    hence A = insert (f n) ?B using Suc.prems by(auto simp:less-Suc-eq)
    thus ?thesis using finB by simp
  qed
qed

lemma finite-conv-nat-seg-image:
  finite A = (∃ (n::nat) f. A = f ` {i::nat. i<n})
by(blast intro: nat-seg-image-imp-finite dest: finite-imp-nat-seg-image-inj-on)

lemma finite-imp-inj-to-nat-seg:
assumes finite A
shows EX f n::nat. f`A = {i. i<n} & inj-on f A
proof -
  from finite-imp-nat-seg-image-inj-on[OF ‹finite A›]
  obtain f and n::nat where bij: bij-betw f {i. i<n} A
    by (auto simp:bij-betw-def)
  let ?f = the-inv-into {i. i<n} f
  have inj-on ?f A & ?f ` A = {i. i<n}
    by (fold bij-betw-def) (rule bij-betw-the-inv-into[OF bij])
  thus ?thesis by blast
qed
```

lemma *finite-Collect-less-nat*[*iff*]: *finite*{*n*::*nat*. *n*<*k*}
by(*fastsimp simp*: *finite-conv-nat-seg-image*)

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 *rev-finite-subset*: *finite B* ==> *A* ⊆ *B* ==> *finite A*
by (*rule finite-subset*)

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-Inter*[*intro*]: *EX A*:*M*. *finite*(*A*) ⟹ *finite*(*Inter M*)
by (*blast intro*: *Inter-lower finite-subset*)

lemma *finite-INT*[*intro*]: *EX x*:*I*. *finite*(*A x*) ⟹ *finite*(*INT x*:*I*. *A x*)
by (*blast intro*: *INT-lower finite-subset*)

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-image-set [simp]:
 finite {x. P x} ⟹ finite { f x | x. P x }
 by (simp add: image-Collect [symmetric])

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-subset* [*OF inj-vimage-singleton*])
done

lemma *finite-vimageD*:
assumes *fin*: *finite* (*h* −‘ *F*) **and** *surj*: *surj h*
shows *finite F*
proof −
have *finite* (*h* ‘ (*h* −‘ *F*)) **using** *fin* **by** (*rule finite-imageI*)
also have *h* ‘ (*h* −‘ *F*) = *F* **using** *surj* **by** (*rule surj-image-vimage-eq*)
finally show *finite F* **.**
qed

lemma *finite-vimage-iff*: *bij h* ⟹ *finite* (*h* −‘ *F*) ⟷ *finite F*

unfolding *bij-def* **by** (*auto elim*: *finite-vimageD finite-vimageI*)

The finite UNION of finite sets

lemma *finite-UN-I*: *finite A* ==> (!!*a*. *a*:*A* ==> *finite* (*B a*)) ==> *finite* (*UN a*:*A*. *B a*)
by (*induct set*: *finite*) *simp-all*

Strengthen RHS to $(\forall x{\in}A.\ \mathit{finite}\ (B\ x)) \wedge \mathit{finite}\ \{x \in A.\ B\ x \neq \{\}\}$?
We'd need to prove $\mathit{finite}\ C \Longrightarrow \forall A\ B.\ \mathit{UNION}\ A\ B \subseteq C \longrightarrow \mathit{finite}\ \{x \in A.\ B\ x \neq \{\}\}$ by induction.

lemma *finite-UN* [*simp*]:
finite A ==> *finite* (*UNION A B*) = (*ALL x*:*A*. *finite* (*B x*))
by (*blast intro*: *finite-UN-I finite-subset*)

lemma *finite-Collect-bex*[*simp*]: *finite A* $\Longrightarrow$
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*} $\Longrightarrow$
finite{*x*. *EX y*. *P y* & *Q x y*} = (*ALL y*. *P y* $\longrightarrow$ *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*)

lemma *finite-PlusD*:
fixes *A* :: *'a set* **and** *B* :: *'b set*
assumes *fin*: *finite* (*A* <+> *B*)
shows *finite A finite B*
proof −
have *Inl* ' *A* ⊆ *A* <+> *B* **by** *auto*
hence *finite* (*Inl* ' *A* :: (*'a* + *'b*) *set*) **using** *fin* **by**(*rule finite-subset*)
thus *finite A* **by**(*rule finite-imageD*)(*auto intro*: *inj-onI*)
next
have *Inr* ' *B* ⊆ *A* <+> *B* **by** *auto*
hence *finite* (*Inr* ' *B* :: (*'a* + *'b*) *set*) **using** *fin* **by**(*rule finite-subset*)
thus *finite B* **by**(*rule finite-imageD*)(*auto intro*: *inj-onI*)
qed

lemma *finite-Plus-iff*[*simp*]: *finite* (*A* <+> *B*) ⟷ *finite A* ∧ *finite B*
by(*auto intro*: *finite-PlusD finite-Plus*)

lemma *finite-Plus-UNIV-iff*[*simp*]:
finite (*UNIV* :: (*'a* + *'b*) *set*) =

(*finite* (*UNIV* :: *'a set*) & *finite* (*UNIV* :: *'b set*))
by(*subst UNIV-Plus-UNIV*[*symmetric*])(*rule finite-Plus-iff*)

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*: *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*])

lemma *finite-subset-image*:
  assumes *finite B*
  shows *B* ⊆ *f* ' *A* ⟹ ∃ *C*⊆*A*. *finite C* ∧ *B* = *f* ' *C*
using *assms* **proof**(*induct*)
  case *empty* **thus** *?case* **by** *simp*
next
  case *insert* **thus** *?case*
    by (*clarsimp simp del*: *image-insert simp add*: *image-insert*[*symmetric*])
      blast
qed

20.2 Class *finite*

class *finite* =
  assumes *finite-UNIV*: *finite* (*UNIV* :: *'a set*)
begin

lemma *finite* [*simp*]: *finite* (*A* :: *'a set*)
  by (*rule subset-UNIV finite-UNIV finite-subset*)+

end

lemma *UNIV-unit* [*no-atp*]:
  UNIV = {()} **by** *auto*

instance *unit* :: *finite* **proof**
qed (*simp add*: *UNIV-unit*)

lemma *UNIV-bool* [*no-atp*]:

UNIV = {*False*, *True*} **by** *auto*

instance *bool* :: *finite* **proof**
qed (*simp add*: *UNIV-bool*)

instance ∗ :: (*finite*, *finite*) *finite* **proof**
qed (*simp only*: *UNIV-Times-UNIV* [*symmetric*] *finite-cartesian-product finite*)

lemma *finite-option-UNIV* [*simp*]:
finite (*UNIV* :: *'a option set*) = *finite* (*UNIV* :: *'a set*)
by (*auto simp add*: *UNIV-option-conv elim*: *finite-imageD intro*: *inj-Some*)

instance *option* :: (*finite*) *finite* **proof**
qed (*simp add*: *UNIV-option-conv*)

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* **proof**
qed (*simp only*: *UNIV-Plus-UNIV* [*symmetric*] *finite-Plus finite*)

20.3 A basic 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*

20.3.1 From *fold-graph* to *fold*

context *fun-left-comm*
begin

lemma *fold-graph-insertE-aux*:
fold-graph $f\ z\ A\ y \Longrightarrow a \in A \Longrightarrow \exists y'.\ y = f\ a\ y' \wedge$ *fold-graph* $f\ z\ (A - \{a\})\ y'$
proof (*induct set*: *fold-graph*)
case (*insertI* $x\ A\ y$) **show** *?case*
proof (*cases* $x = a$)
assume $x = a$ **with** *insertI* **show** *?case* **by** *auto*
next
assume $x \neq a$
then obtain y' **where** y: $y = f\ a\ y'$ **and** y': *fold-graph* $f\ z\ (A - \{a\})\ y'$
using *insertI* **by** *auto*
have *1*: $f\ x\ y = f\ a\ (f\ x\ y')$
unfolding y **by** (*rule fun-left-comm*)
have *2*: *fold-graph* $f\ z\ (insert\ x\ A - \{a\})\ (f\ x\ y')$
using y' **and** ‹$x \neq a$› **and** ‹$x \notin A$›
by (*simp add*: *insert-Diff-if fold-graph.insertI*)
from *1 2* **show** *?case* **by** *fast*
qed

qed *simp*

lemma *fold-graph-insertE*:
  assumes *fold-graph f z* (*insert x A*) *v* **and** $x \notin A$
  obtains *y* **where** $v = f\ x\ y$ **and** *fold-graph f z A y*
using *assms* **by** (*auto dest*: *fold-graph-insertE-aux* [*OF - insertI1*])

lemma *fold-graph-determ*:
  fold-graph f z A x $\Longrightarrow$ *fold-graph f z A y* $\Longrightarrow$ $y = x$
proof (*induct arbitrary*: *y set*: *fold-graph*)
  case (*insertI x A y v*)
  from ‹*fold-graph f z* (*insert x A*) *v*› **and** ‹$x \notin A$›
  obtain y' **where** $v = f\ x\ y'$ **and** *fold-graph f z A* y'
    by (*rule fold-graph-insertE*)
  from ‹*fold-graph f z A* y'› **have** $y' = y$ **by** (*rule insertI*)
  with ‹$v = f\ x\ y'$› **show** $v = f\ x\ y$ **by** *simp*
qed *fast*

lemma *fold-equality*:
  fold-graph f z A y $\Longrightarrow$ *fold f z A* $= y$
by (*unfold fold-def*) (*blast intro*: *fold-graph-determ*)

lemma *fold-graph-fold*: *finite A* $\Longrightarrow$ *fold-graph f z A* (*fold f z A*)
unfolding *fold-def*
apply (*rule theI'*)
apply (*rule ex-ex1I*)
apply (*erule finite-imp-fold-graph*)
apply (*erule* (*1*) *fold-graph-determ*)
done

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* [*simp*]:
  finite A ==> $x \notin A$ ==> *fold f z* (*insert x A*) $= f\ x$ (*fold f z A*)
apply (*rule fold-equality*)
apply (*erule fold-graph.insertI*)
apply (*erule fold-graph-fold*)
done

lemma *fold-fun-comm*:
  finite A $\Longrightarrow$ $f\ x$ (*fold f z A*) = *fold f* (*f x z*) *A*
proof (*induct rule*: *finite-induct*)
  case *empty* **then show** *?case* **by** *simp*
next
  case (*insert y A*) **then show** *?case*
    by (*simp add*: *fun-left-comm*[*of x*])

qed

lemma *fold-insert2*:
finite A $\Longrightarrow$ $x \notin A$ $\Longrightarrow$ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
by (*simp add*: *fold-fun-comm*)

lemma *fold-rec*:
assumes *finite A* **and** $x \in A$
shows *fold f z A* = *f x* (*fold f z* ($A - \{x\}$))
proof −
have *A*: *A* = *insert x* ($A - \{x\}$) **using** ‹$x \in A$› **by** *blast*
then have *fold f z A* = *fold f z* (*insert x* ($A - \{x\}$)) **by** *simp*
also have ... = *f x* (*fold f z* ($A - \{x\}$))
by (*rule fold-insert*) (*simp add*: ‹*finite A*›)+
finally show *?thesis* **.**
qed

lemma *fold-insert-remove*:
assumes *finite A*
shows *fold f z* (*insert x A*) = *f x* (*fold f z* ($A - \{x\}$))
proof −
from ‹*finite A*› **have** *finite* (*insert x A*) **by** *auto*
moreover have $x \in$ *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 $x \in A$
then obtain *B* **where** *A* = *insert x B* **and** $x \notin B$ **by** (*rule set-insert*)
then show *?thesis* **using** *assms* **by** (*simp add*:*fun-left-idem*)
next
assume $x \notin A$ **then show** *?thesis* **using** *assms* **by** *simp*
qed

declare *fold-insert*[*simp del*] *fold-insert-idem*[*simp*]

lemma *fold-insert-idem2*:
  finite A $\Longrightarrow$ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
by(*simp add*:*fold-fun-comm*)

end

20.3.2 Expressing set operations via *fold*

lemma (**in** *fun-left-comm*) *fun-left-comm-apply*:
  fun-left-comm (λx. *f* (*g x*))
proof
qed (*simp-all add*: *fun-left-comm*)

lemma (**in** *fun-left-comm-idem*) *fun-left-comm-idem-apply*:
  fun-left-comm-idem (λx. *f* (*g x*))
  by (*rule fun-left-comm-idem.intro*, *rule fun-left-comm-apply*, *unfold-locales*)
    (*simp-all add*: *fun-left-idem*)

lemma *fun-left-comm-idem-insert*:
  fun-left-comm-idem insert
proof
qed *auto*

lemma *fun-left-comm-idem-remove*:
  fun-left-comm-idem (λx *A*. *A* − {*x*})
proof
qed *auto*

lemma (**in** *semilattice-inf*) *fun-left-comm-idem-inf*:
  fun-left-comm-idem inf
proof
qed (*auto simp add*: *inf-left-commute*)

lemma (**in** *semilattice-sup*) *fun-left-comm-idem-sup*:
  fun-left-comm-idem sup
proof
qed (*auto simp add*: *sup-left-commute*)

lemma *union-fold-insert*:
  assumes *finite A*
  shows *A* ∪ *B* = *fold insert B A*
proof −
  interpret *fun-left-comm-idem insert* **by** (*fact fun-left-comm-idem-insert*)
  from ‹*finite A*› **show** *?thesis* **by** (*induct A arbitrary*: *B*) *simp-all*
qed

lemma *minus-fold-remove*:
assumes *finite A*
shows $B - A = fold\ (\lambda x\ A.\ A - \{x\})\ B\ A$
proof −
interpret *fun-left-comm-idem* $\lambda x\ A.\ A - \{x\}$ **by** (*fact fun-left-comm-idem-remove*)
from ‹*finite A*› **show** *?thesis* **by** (*induct A arbitrary*: *B*) *auto*
qed

context *complete-lattice*
begin

lemma *inf-Inf-fold-inf*:
assumes *finite A*
shows *inf B* (*Inf A*) = *fold inf B A*
proof −
interpret *fun-left-comm-idem inf* **by** (*fact fun-left-comm-idem-inf*)
from ‹*finite A*› **show** *?thesis* **by** (*induct A arbitrary*: *B*)
(*simp-all add*: *Inf-empty Inf-insert inf-commute fold-fun-comm*)
qed

lemma *sup-Sup-fold-sup*:
assumes *finite A*
shows *sup B* (*Sup A*) = *fold sup B A*
proof −
interpret *fun-left-comm-idem sup* **by** (*fact fun-left-comm-idem-sup*)
from ‹*finite A*› **show** *?thesis* **by** (*induct A arbitrary*: *B*)
(*simp-all add*: *Sup-empty Sup-insert sup-commute fold-fun-comm*)
qed

lemma *Inf-fold-inf*:
assumes *finite A*
shows *Inf A* = *fold inf top A*
using *assms inf-Inf-fold-inf* [*of A top*] **by** (*simp add*: *inf-absorb2*)

lemma *Sup-fold-sup*:
assumes *finite A*
shows *Sup A* = *fold sup bot A*
using *assms sup-Sup-fold-sup* [*of A bot*] **by** (*simp add*: *sup-absorb2*)

lemma *inf-INFI-fold-inf*:
assumes *finite A*
shows *inf B* (*INFI A f*) = *fold* ($\lambda A.$ *inf* (*f A*)) *B A* (**is** *?inf* = *?fold*)
proof (*rule sym*)
interpret *fun-left-comm-idem inf* **by** (*fact fun-left-comm-idem-inf*)
interpret *fun-left-comm-idem* $\lambda A.$ *inf* (*f A*) **by** (*fact fun-left-comm-idem-apply*)
from ‹*finite A*› **show** *?fold* = *?inf*
by (*induct A arbitrary*: *B*)
(*simp-all add*: *INFI-def Inf-empty Inf-insert inf-left-commute*)
qed

lemma *sup-SUPR-fold-sup*:
assumes *finite A*
shows *sup B* (*SUPR A f*) = *fold* (λ*A*. *sup* (*f A*)) *B A* (**is** *?sup* = *?fold*)
proof (*rule sym*)
interpret *fun-left-comm-idem sup* **by** (*fact fun-left-comm-idem-sup*)
interpret *fun-left-comm-idem* λ*A*. *sup* (*f A*) **by** (*fact fun-left-comm-idem-apply*)
from ‹*finite A*› **show** *?fold* = *?sup*
by (*induct A arbitrary*: *B*)
(*simp-all add*: *SUPR-def Sup-empty Sup-insert sup-left-commute*)
qed

lemma *INFI-fold-inf*:
assumes *finite A*
shows *INFI A f* = *fold* (λ*A*. *inf* (*f A*)) *top A*
using *assms inf-INFI-fold-inf* [*of A top*] **by** *simp*

lemma *SUPR-fold-sup*:
assumes *finite A*
shows *SUPR A f* = *fold* (λ*A*. *sup* (*f A*)) *bot A*
using *assms sup-SUPR-fold-sup* [*of A bot*] **by** *simp*

end

20.4 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* ∉ *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*)
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* **by** *induct auto*

lemma *fold-image-cong*:
finite A $\Longrightarrow$
(!!*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-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*

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*)

lemma *fold-image-Un-one*:
assumes *fS*: *finite S* **and** *fT*: *finite T*
and *I0*: $\forall$ *x* ∈ *S*∩*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

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*: ∀ *y*∈*S'*. ∃!*x*. *x*∈ *S* ∧ *h*(*x*) = *y*
and *f12*: ∀ *x*∈*S*. *h x* ∈ *S'* ∧ *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*: ⋀*x*. *x* ∈ *S* ⟹ (*f2* ∘ *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*: ⋀*y*. *y* ∈ *T* ⟹ *k y* ∈ *S* ∧ *h* (*k y*) = *y*
and *hk*: ⋀*x*. *x* ∈ *S* ⟹ *h x* ∈ *T* ∧ *k* (*h x*) = *x* ∧ *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

20.5 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*

definition *fold1* :: ('*a* => '*a* => '*a*) => '*a set* => '*a* **where**
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 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* **show** *?case* **by** (*subst mult-commute* [*of z b*], *fast*)
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. $[\![a \in A;\ b \notin A]\!] \Longrightarrow$ *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 $a = x \vee a \in A$ **using** *insertI* **by** *simp*
thus *?case*
proof
assume $a = x$
with *insertI* **show** *?thesis*
by (*simp add*: *id-def* [*symmetric*], *blast intro*: *fold-graph-insert-swap*)
next
assume *ainA*: $a \in A$
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*: *fold-graph-determ*)
apply (*subgoal-tac fold-graph times a A x*)
apply (*best intro*: *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*: ($A \neq \{\}$) = ($\exists\, x\ B.\ A = insert\ x\ B\ \&\ x \notin B$)
apply *safe*
apply *simp*
apply (*drule-tac x*=*x* **in** *spec*)
apply (*drule-tac x*=*A*−{x} **in** *spec*, *auto*)
done

lemma *fold1-insert*:
  assumes *nonempty*: $A \neq \{\}$ **and** A: *finite* A $x \notin A$
  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 (*rule mult-left-commute*)
apply (*rule mult-left-idem*)
done

lemma *fold1-insert-idem* [*simp*]:
  assumes *nonempty*: $A \neq \{\}$ **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*
    next
      assume $A' \neq \{\}$
      with *prems* **show** *?thesis*
        by (*simp add*: *fold1-insert mult-assoc* [*symmetric*])
    qed
  next
    assume $a \neq x$
    with *prems* **show** *?thesis*
      by (*simp add*: *insert-commute fold1-eq-fold*)
  qed
qed

lemma *hom-fold1-commute*:
assumes *hom*: !!$x\ y.\ h\ (x * y) = h\ x * h\ y$
and N: *finite* N $N \neq \{\}$ **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

lemma *fold1-eq-fold-idem*:
assumes *finite A*
shows *fold1 times* (*insert a A*) = *fold times a A*
proof (*cases* $a \in A$)
case *False*
with *assms* **show** *?thesis* **by** (*simp add*: *fold1-eq-fold*)
next
interpret *fun-left-comm-idem times* **by** (*fact fun-left-comm-idem*)
case *True* **then obtain** *b B*
where A: A = *insert a B* **and** $a \notin B$ **by** (*rule set-insert*)
with *assms* **have** *finite B* **by** *auto*
then have *fold times a* (*insert a B*) = *fold times* ($a * a$) B
using ‹$a \notin B$› **by** (*rule fold-insert2*)
then show *?thesis*
using ‹$a \notin B$› ‹*finite B*› **by** (*simp add*: *fold1-eq-fold A*)
qed

end

Now the recursion rules for definitions:

lemma *fold1-singleton-def*: g = *fold1* $f \Longrightarrow g\ \{a\} = a$
by *simp*

lemma (**in** *ab-semigroup-mult*) *fold1-insert-def*:
⟦ g = *fold1 times*; *finite A*; $x \notin A$; $A \neq \{\}$ ⟧ $\Longrightarrow g$ (*insert x A*) = $x * g\ A$
by (*simp add*:*fold1-insert*)

lemma (**in** *ab-semigroup-idem-mult*) *fold1-insert-idem-def*:
⟦ g = *fold1 times*; *finite A*; $A \neq \{\}$ ⟧ $\Longrightarrow g$ (*insert x A*) = $x * g\ A$
by *simp*

20.5.1 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.

20.5.2 Lemmas about *fold1*

context *ab-semigroup-mult*
begin

lemma *fold1-Un*:
assumes *A*: *finite A A* $\neq \{\}$
shows *finite B* $\Longrightarrow$ *B* $\neq \{\} \Longrightarrow$ *A Int B* $= \{\} \Longrightarrow$
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* $\neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x * y \in \{x,y\}$
shows *fold1 times A* $\in$ *A*
using *A*
proof (*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **using** *elem* **by** (*force simp add*:*fold1-insert*)
qed

end

lemma (**in** *ab-semigroup-idem-mult*) *fold1-Un2*:
assumes *A*: *finite A A* $\neq \{\}$
shows *finite B* $\Longrightarrow$ *B* $\neq \{\} \Longrightarrow$
fold1 times (*A Un B*) = *fold1 times A* $*$ *fold1 times B*
using *A*
proof(*induct rule*:*finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case *insert* **thus** *?case* **by** (*simp add*: *mult-assoc*)
qed

20.6 Locales as mini-packages for fold operations

20.6.1 The natural case

locale *folding* =
fixes *f* :: $'a \Rightarrow 'b \Rightarrow 'b$
fixes *F* :: $'a$ *set* $\Rightarrow 'b \Rightarrow 'b$
assumes *commute-comp*: *f y* $\circ$ *f x* = *f x* $\circ$ *f y*

assumes *eq-fold*: *finite A* $\Longrightarrow$ *F A s* = *fold f s A*
begin

lemma *empty* [*simp*]:
F {} = *id*
by (*simp add*: *eq-fold expand-fun-eq*)

lemma *insert* [*simp*]:
assumes *finite A* **and** $x \notin A$
shows *F* (*insert x A*) = *F A* ∘ *f x*
proof −
interpret *fun-left-comm f* **proof**
qed (*insert commute-comp*, *simp add*: *expand-fun-eq*)
from *fold-insert2 assms*
have $\bigwedge s$. *fold f s* (*insert x A*) = *fold f* (*f x s*) *A* **.**
with ‹*finite A*› **show** *?thesis* **by** (*simp add*: *eq-fold expand-fun-eq*)
qed

lemma *remove*:
assumes *finite A* **and** $x \in A$
shows *F A* = *F* (*A* − {*x*}) ∘ *f x*
proof −
from ‹$x \in A$› **obtain** *B* **where** *A*: *A* = *insert x B* **and** $x \notin B$
by (*auto dest*: *mk-disjoint-insert*)
moreover from ‹*finite A*› *this* **have** *finite B* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *insert-remove*:
assumes *finite A*
shows *F* (*insert x A*) = *F* (*A* − {*x*}) ∘ *f x*
using *assms* **by** (*cases* $x \in A$) (*simp-all add*: *remove insert-absorb*)

lemma *commute-left-comp*:
f y ∘ (*f x* ∘ *g*) = *f x* ∘ (*f y* ∘ *g*)
by (*simp add*: *o-assoc commute-comp*)

lemma *commute-comp′*:
assumes *finite A*
shows *f x* ∘ *F A* = *F A* ∘ *f x*
using *assms* **by** (*induct A*)
(*simp*, *simp del*: *o-apply add*: *o-assoc*, *simp del*: *o-apply add*: *o-assoc* [*symmetric*] *commute-comp*)

lemma *commute-left-comp′*:
assumes *finite A*
shows *f x* ∘ (*F A* ∘ *g*) = *F A* ∘ (*f x* ∘ *g*)
using *assms* **by** (*simp add*: *o-assoc commute-comp′*)

lemma *commute-comp''*:
assumes *finite A* **and** *finite B*
shows $F\ B \circ F\ A = F\ A \circ F\ B$
using *assms* **by** (*induct A*)
(*simp-all add*: *o-assoc*, *simp add*: *o-assoc* [*symmetric*] *commute-comp'*)

lemma *commute-left-comp''*:
assumes *finite A* **and** *finite B*
shows $F\ B \circ (F\ A \circ g) = F\ A \circ (F\ B \circ g)$
using *assms* **by** (*simp add*: *o-assoc commute-comp''*)

lemmas *commute-comps* = *o-assoc* [*symmetric*] *commute-comp commute-left-comp commute-comp' commute-left-comp' commute-comp'' commute-left-comp''*

lemma *union-inter*:
assumes *finite A* **and** *finite B*
shows $F\ (A \cup B) \circ F\ (A \cap B) = F\ A \circ F\ B$
using *assms* **by** (*induct A*)
(*simp-all del*: *o-apply add*: *insert-absorb Int-insert-left commute-comps*,
simp add: *o-assoc*)

lemma *union*:
assumes *finite A* **and** *finite B*
and $A \cap B = \{\}$
shows $F\ (A \cup B) = F\ A \circ F\ B$
proof −
from *union-inter* ‹*finite A*› ‹*finite B*› **have** $F\ (A \cup B) \circ F\ (A \cap B) = F\ A \circ F\ B$ **.**
with ‹$A \cap B = \{\}$› **show** *?thesis* **by** *simp*
qed

end

20.6.2 The natural case with idempotency

locale *folding-idem* = *folding* +
assumes *idem-comp*: $f\ x \circ f\ x = f\ x$
begin

lemma *idem-left-comp*:
$f\ x \circ (f\ x \circ g) = f\ x \circ g$
by (*simp add*: *o-assoc idem-comp*)

lemma *in-comp-idem*:
assumes *finite A* **and** $x \in A$
shows $F\ A \circ f\ x = F\ A$
using *assms* **by** (*induct A*)
(*auto simp add*: *commute-comps idem-comp*, *simp add*: *commute-left-comp'* [*symmetric*] *commute-comp'*)

lemma *subset-comp-idem*:
assumes *finite A* **and** $B \subseteq A$
shows $F\ A \circ F\ B = F\ A$
proof −
from *assms* **have** *finite B* **by** (*blast dest*: *finite-subset*)
then show *?thesis* **using** ‹$B \subseteq A$› **by** (*induct B*)
(*simp-all add*: *o-assoc in-comp-idem* ‹*finite A*›)
qed

declare *insert* [*simp del*]

lemma *insert-idem* [*simp*]:
assumes *finite A*
shows $F\ (\mathit{insert}\ x\ A) = F\ A \circ f\ x$
using *assms* **by** (*cases* $x \in A$) (*simp-all add*: *insert in-comp-idem insert-absorb*)

lemma *union-idem*:
assumes *finite A* **and** *finite B*
shows $F\ (A \cup B) = F\ A \circ F\ B$
proof −
from *assms* **have** *finite* $(A \cup B)$ **and** $A \cap B \subseteq A \cup B$ **by** *auto*
then have $F\ (A \cup B) \circ F\ (A \cap B) = F\ (A \cup B)$ **by** (*rule subset-comp-idem*)
with *assms* **show** *?thesis* **by** (*simp add*: *union-inter*)
qed

end

20.6.3 The image case with fixed function

no-notation *times* (**infixl** $*$ *70*)
no-notation *Groups.one* (*1*)

locale *folding-image-simple* = *comm-monoid* +
fixes $g :: ('b \Rightarrow 'a)$
fixes $F :: 'b\ set \Rightarrow 'a$
assumes *eq-fold-g*: *finite* $A \Longrightarrow F\ A = \mathit{fold\text{-}image}\ f\ g\ 1\ A$
begin

lemma *empty* [*simp*]:
$F\ \{\} = 1$
by (*simp add*: *eq-fold-g*)

lemma *insert* [*simp*]:
assumes *finite A* **and** $x \notin A$
shows $F\ (\mathit{insert}\ x\ A) = g\ x * F\ A$
proof −
interpret *fun-left-comm* $\%x\ y.\ (g\ x) * y$ **proof**
qed (*simp add*: *ac-simps*)

with *assms* **have** *fold-image* (*op* ∗) *g 1* (*insert x A*) = *g x* ∗ *fold-image* (*op* ∗) *g 1 A*
by (*simp add*: *fold-image-def*)
with ‹*finite A*› **show** *?thesis* **by** (*simp add*: *eq-fold-g*)
qed

lemma *remove*:
assumes *finite A* **and** *x* ∈ *A*
shows *F A* = *g x* ∗ *F* (*A* − {*x*})
proof −
from ‹*x* ∈ *A*› **obtain** *B* **where** *A*: *A* = *insert x B* **and** *x* ∉ *B*
by (*auto dest*: *mk-disjoint-insert*)
moreover from ‹*finite A*› *this* **have** *finite B* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *insert-remove*:
assumes *finite A*
shows *F* (*insert x A*) = *g x* ∗ *F* (*A* − {*x*})
using *assms* **by** (*cases x* ∈ *A*) (*simp-all add*: *remove insert-absorb*)

lemma *neutral*:
assumes *finite A* **and** ∀ *x*∈*A*. *g x* = *1*
shows *F A* = *1*
using *assms* **by** (*induct A*) *simp-all*

lemma *union-inter*:
assumes *finite A* **and** *finite B*
shows *F* (*A* ∪ *B*) ∗ *F* (*A* ∩ *B*) = *F A* ∗ *F B*
using *assms* **proof** (*induct A*)
case *empty* **then show** *?case* **by** *simp*
next
case (*insert x A*) **then show** *?case*
by (*auto simp add*: *insert-absorb Int-insert-left commute* [*of* - *g x*] *assoc left-commute*)
qed

corollary *union-inter-neutral*:
assumes *finite A* **and** *finite B*
and *I0*: ∀ *x* ∈ *A*∩*B*. *g x* = *1*
shows *F* (*A* ∪ *B*) = *F A* ∗ *F B*
using *assms* **by** (*simp add*: *union-inter* [*symmetric*] *neutral*)

corollary *union-disjoint*:
assumes *finite A* **and** *finite B*
assumes *A* ∩ *B* = {}
shows *F* (*A* ∪ *B*) = *F A* ∗ *F B*
using *assms* **by** (*simp add*: *union-inter-neutral*)

end

20.6.4 The image case with flexible function

locale *folding-image* = *comm-monoid* +
  fixes $F :: ('b \Rightarrow 'a) \Rightarrow 'b\ set \Rightarrow 'a$
  assumes *eq-fold*: $\bigwedge g.\ finite\ A \Longrightarrow F\ g\ A = fold\text{-}image\ f\ g\ 1\ A$

sublocale *folding-image* < *folding-image-simple op* $*$ *1 g F g* **proof**
qed (*fact eq-fold*)

context *folding-image*
begin

lemma *reindex*:
  assumes *finite A* **and** *inj-on h A*
  shows $F\ g\ (h\ `\ A) = F\ (g \circ h)\ A$
  using *assms* **by** (*induct A*) *auto*

lemma *cong*:
  assumes *finite A* **and** $\bigwedge x.\ x \in A \Longrightarrow g\ x = h\ x$
  shows $F\ g\ A = F\ h\ A$
proof −
  from *assms* **have** *ALL C. C <= A −−> (ALL x:C. g x = h x) −−> F g C = F h C*
  apply − **apply** (*erule finite-induct*) **apply** *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
  with *assms* **show** *?thesis* **by** *simp*
qed

lemma *UNION-disjoint*:
  assumes *finite I* **and** $\forall\, i{\in}I.$ *finite* $(A\ i)$
  and $\forall\, i{\in}I.\ \forall\, j{\in}I.\ i \neq j \longrightarrow A\ i \cap A\ j = \{\}$
  shows $F\ g\ (UNION\ I\ A) = F\ (F\ g \circ A)\ I$
apply (*insert assms*)
apply (*induct set*: *finite*, *simp*, *atomize*)
apply (*subgoal-tac* $\forall\, i{\in}Fa.\ x \neq i$)
 prefer *2* **apply** *blast*
apply (*subgoal-tac A x Int UNION Fa A* $= \{\}$)
 prefer *2* **apply** *blast*

apply (*simp add*: *union-disjoint*)
done

lemma *distrib*:
assumes *finite A*
shows $F\ (\lambda x.\ g\ x * h\ x)\ A = F\ g\ A * F\ h\ A$
using *assms* **by** (*rule finite-induct*) (*simp-all add*: *assoc commute left-commute*)

lemma *related*:
assumes *Re*: $R\ 1\ 1$
and *Rop*: $\forall x1\ y1\ x2\ y2.\ R\ x1\ x2 \wedge R\ y1\ y2 \longrightarrow R\ (x1 * y1)\ (x2 * y2)$
and *fS*: *finite S* **and** *Rfg*: $\forall x{\in}S.\ R\ (h\ x)\ (g\ x)$
shows $R\ (F\ h\ S)\ (F\ g\ S)$
using *fS* **by** (*rule finite-subset-induct*) (*insert assms*, *auto*)

lemma *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 $F\ f1\ S = F\ f2\ S'$
proof−
from *h f12* **have** *hS*: $h\ `\ S = S'$ **by** *blast*
{fix *x y* **assume** *H*: $x \in S\ y \in S\ h\ x = h\ y$
from *f12 h H* **have** $x = y$ **by** *auto* **}**
hence *hinj*: *inj-on h S* **unfolding** *inj-on-def Ex1-def* **by** *blast*
from *f12* **have** *th*: $\bigwedge x.\ x \in S \Longrightarrow (f2 \circ h)\ x = f1\ x$ **by** *auto*
from *hS* **have** $F\ f2\ S' = F\ f2\ (h\ `\ S)$ **by** *simp*
also have $\ldots = F\ (f2\ o\ h)\ S$ **using** *reindex* [*OF fS hinj*, *of f2*] **.**
also have $\ldots = F\ f1\ S$ **using** *th cong* [*OF fS*, *of f2 o h f1*]
by *blast*
finally show *?thesis* **..**
qed

lemma *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*: $\bigwedge x.\ x \in S \Longrightarrow h\ x \in T \wedge k\ (h\ x) = x \wedge g\ (h\ x) = j\ x$
shows $F\ j\ S = F\ g\ T$

apply (*rule eq-general* [*OF fS*, *of T h g j*])
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

20.6.5 The image case with fixed function and idempotency

locale *folding-image-simple-idem* = *folding-image-simple* +
assumes *idem*: $x * x = x$

sublocale *folding-image-simple-idem* < *semilattice* **proof**
qed (*fact idem*)

context *folding-image-simple-idem*
begin

lemma *in-idem*:
assumes *finite A* **and** $x \in A$
shows $g\ x * F\ A = F\ A$
using *assms* **by** (*induct A*) (*auto simp add*: *left-commute*)

lemma *subset-idem*:
assumes *finite A* **and** $B \subseteq A$
shows $F\ B * F\ A = F\ A$
proof −
from *assms* **have** *finite B* **by** (*blast dest*: *finite-subset*)
then show *?thesis* **using** ‹$B \subseteq A$› **by** (*induct B*)
(*auto simp add*: *assoc in-idem* ‹*finite A*›)
qed

declare *insert* [*simp del*]

lemma *insert-idem* [*simp*]:
assumes *finite A*
shows $F\ (\mathit{insert}\ x\ A) = g\ x * F\ A$
using *assms* **by** (*cases* $x \in A$) (*simp-all add*: *insert in-idem insert-absorb*)

lemma *union-idem*:
assumes *finite A* **and** *finite B*
shows $F\ (A \cup B) = F\ A * F\ B$
proof −
from *assms* **have** *finite* $(A \cup B)$ **and** $A \cap B \subseteq A \cup B$ **by** *auto*
then have $F\ (A \cap B) * F\ (A \cup B) = F\ (A \cup B)$ **by** (*rule subset-idem*)
with *assms* **show** *?thesis* **by** (*simp add*: *union-inter* [*of A B*, *symmetric*] *commute*)
qed

end

20.6.6 The image case with flexible function and idempotency

locale *folding-image-idem* = *folding-image* +
  assumes *idem*: $x * x = x$

sublocale *folding-image-idem* < *folding-image-simple-idem op* $*$ *1 g F g* **proof qed** (*fact idem*)

20.6.7 The neutral-less case

locale *folding-one* = *abel-semigroup* +
  fixes $F :: 'a\ set \Rightarrow 'a$
  assumes *eq-fold*: *finite* $A \Longrightarrow F\ A = fold1\ f\ A$
begin

lemma *singleton* [*simp*]:
  $F\ \{x\} = x$
  by (*simp add*: *eq-fold*)

lemma *eq-fold'*:
  assumes *finite* A **and** $x \notin A$
  shows $F\ (insert\ x\ A) = fold\ (op\ *)\ x\ A$
proof −
  interpret *ab-semigroup-mult op* $*$ **proof qed** (*simp-all add*: *ac-simps*)
  with *assms* **show** *?thesis* **by** (*simp add*: *eq-fold fold1-eq-fold*)
qed

lemma *insert* [*simp*]:
  assumes *finite* A **and** $x \notin A$ **and** $A \neq \{\}$
  shows $F\ (insert\ x\ A) = x * F\ A$
proof −
  from ‹$A \neq \{\}$› **obtain** b **where** $b \in A$ **by** *blast*
 then obtain B **where** $*$: $A = insert\ b\ B\ b \notin B$ **by** (*blast dest*: *mk-disjoint-insert*)
  with ‹*finite A*› **have** *finite B* **by** *simp*
  interpret *fold*: *folding op* $*$ $\lambda a\ b.\ fold\ (op\ *)\ b\ a$ **proof**
  qed (*simp-all add*: *expand-fun-eq ac-simps*)
  thm *fold.commute-comp'* [*of B b*, *simplified expand-fun-eq*, *simplified*]
  from ‹*finite B*› *fold.commute-comp'* [*of B x*]
    have *op* $*$ $x \circ (\lambda b.\ fold\ op\ *\ b\ B) = (\lambda b.\ fold\ op\ *\ b\ B) \circ op\ *\ x$ **by** *simp*
 then have *A*: $x * fold\ op\ *\ b\ B = fold\ op\ *\ (b * x)\ B$ **by** (*simp add*: *expand-fun-eq commute*)
  from ‹*finite B*› $*$ *fold.insert* [*of B b*]
    have $(\lambda x.\ fold\ op\ *\ x\ (insert\ b\ B)) = (\lambda x.\ fold\ op\ *\ x\ B) \circ op\ *\ b$ **by** *simp*
  then have *B*: *fold op* $*$ x $(insert\ b\ B) = fold\ op\ *\ (b * x)\ B$ **by** (*simp add*: *expand-fun-eq*)
  from *A B assms* $*$ **show** *?thesis* **by** (*simp add*: *eq-fold' del*: *fold.insert*)
qed

lemma *remove*:
  assumes *finite* A **and** $x \in A$

shows $F\ A = (\mathit{if}\ A - \{x\} = \{\}\ \mathit{then}\ x\ \mathit{else}\ x * F\ (A - \{x\}))$
proof −
from *assms* **obtain** B **where** $A = \mathit{insert}\ x\ B$ **and** $x \notin B$ **by** (*blast dest*: *mk-disjoint-insert*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *insert-remove*:
assumes *finite A*
shows $F\ (\mathit{insert}\ x\ A) = (\mathit{if}\ A - \{x\} = \{\}\ \mathit{then}\ x\ \mathit{else}\ x * F\ (A - \{x\}))$
using *assms* **by** (*cases* $x \in A$) (*simp-all add*: *insert-absorb remove*)

lemma *union-disjoint*:
assumes *finite A* $A \neq \{\}$ **and** *finite B* $B \neq \{\}$ **and** $A \cap B = \{\}$
shows $F\ (A \cup B) = F\ A * F\ B$
using *assms* **by** (*induct A rule*: *finite-ne-induct*) (*simp-all add*: *ac-simps*)

lemma *union-inter*:
assumes *finite A* **and** *finite B* **and** $A \cap B \neq \{\}$
shows $F\ (A \cup B) * F\ (A \cap B) = F\ A * F\ B$
proof −
from *assms* **have** $A \neq \{\}$ **and** $B \neq \{\}$ **by** *auto*
from ‹*finite A*› ‹$A \neq \{\}$› ‹$A \cap B \neq \{\}$› **show** *?thesis* **proof** (*induct A rule*: *finite-ne-induct*)
case (*singleton x*) **then show** *?case* **by** (*simp add*: *insert-absorb ac-simps*)
next
case (*insert x A*) **show** *?case* **proof** (*cases* $x \in B$)
case *True* **then have** $B \neq \{\}$ **by** *auto*
with *insert True* ‹*finite B*› **show** *?thesis* **by** (*cases* $A \cap B = \{\}$)
(*simp-all add*: *insert-absorb ac-simps union-disjoint*)
next
case *False* **with** *insert* **have** $F\ (A \cup B) * F\ (A \cap B) = F\ A * F\ B$ **by** *simp*
moreover from *False* ‹*finite B*› *insert* **have** *finite* $(A \cup B)$ $x \notin A \cup B$ $A \cup B \neq \{\}$
by *auto*
ultimately show *?thesis* **using** *False* ‹*finite A*› ‹$x \notin A$› ‹$A \neq \{\}$› **by** (*simp add*: *assoc*)
qed
qed
qed

lemma *closed*:
assumes *finite A* $A \neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x * y \in \{x, y\}$
shows $F\ A \in A$
using ‹*finite A*› ‹$A \neq \{\}$› **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **then show** *?case* **by** *simp*
next
case *insert* **with** *elem* **show** *?case* **by** *force*
qed

end

20.6.8 The neutral-less case with idempotency

locale *folding-one-idem* = *folding-one* +
  assumes *idem*: $x * x = x$

sublocale *folding-one-idem* < *semilattice* **proof**
qed (*fact idem*)

context *folding-one-idem*
begin

lemma *in-idem*:
  assumes *finite A* **and** $x \in A$
  shows $x * F\ A = F\ A$
proof −
  from *assms* **have** $A \neq \{\}$ **by** *auto*
  with ‹*finite A*› **show** *?thesis* **using** ‹$x \in A$› **by** (*induct A rule*: *finite-ne-induct*)
(*auto simp add*: *ac-simps*)
qed

lemma *subset-idem*:
  assumes *finite A* $B \neq \{\}$ **and** $B \subseteq A$
  shows $F\ B * F\ A = F\ A$
proof −
  from *assms* **have** *finite B* **by** (*blast dest*: *finite-subset*)
  then show *?thesis* **using** ‹$B \neq \{\}$› ‹$B \subseteq A$› **by** (*induct B rule*: *finite-ne-induct*)
    (*simp-all add*: *assoc in-idem* ‹*finite A*›)
qed

lemma *eq-fold-idem'*:
  assumes *finite A*
  shows $F\ (\mathit{insert}\ a\ A) = \mathit{fold}\ (\mathit{op} *)\ a\ A$
proof −
  interpret *ab-semigroup-idem-mult op* ∗ **proof qed** (*simp-all add*: *ac-simps*)
  with *assms* **show** *?thesis* **by** (*simp add*: *eq-fold fold1-eq-fold-idem*)
qed

lemma *insert-idem* [*simp*]:
  assumes *finite A* **and** $A \neq \{\}$
  shows $F\ (\mathit{insert}\ x\ A) = x * F\ A$
proof (*cases* $x \in A$)
  case *False* **from** ‹*finite A*› ‹$x \notin A$› ‹$A \neq \{\}$› **show** *?thesis* **by** (*rule insert*)
next
  case *True*
  from ‹*finite A*› ‹$A \neq \{\}$› **show** *?thesis* **by** (*simp add*: *in-idem insert-absorb True*)
qed

lemma *union-idem*:
assumes *finite A* $A \neq \{\}$ **and** *finite B* $B \neq \{\}$
shows $F\ (A \cup B) = F\ A * F\ B$
proof (*cases* $A \cap B = \{\}$)
case *True* **with** *assms* **show** *?thesis* **by** (*simp add*: *union-disjoint*)
next
case *False*
from *assms* **have** *finite* $(A \cup B)$ **and** $A \cap B \subseteq A \cup B$ **by** *auto*
with *False* **have** $F\ (A \cap B) * F\ (A \cup B) = F\ (A \cup B)$ **by** (*auto intro*: *subset-idem*)
with *assms False* **show** *?thesis* **by** (*simp add*: *union-inter* [*of A B*, *symmetric*] *commute*)
qed

lemma *hom-commute*:
assumes *hom*: $\bigwedge x\ y.\ h\ (x * y) = h\ x * h\ y$
and *N*: *finite N* $N \neq \{\}$ **shows** $h\ (F\ N) = F\ (h\ `\ N)$
using *N* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case (*insert n N*)
then have $h\ (F\ (\mathit{insert}\ n\ N)) = h\ (n * F\ N)$ **by** *simp*
also have $\ldots = h\ n * h\ (F\ N)$ **by** (*rule hom*)
also have $h\ (F\ N) = F\ (h\ `\ N)$ **by**(*rule insert*)
also have $h\ n * \ldots = F\ (\mathit{insert}\ (h\ n)\ (h\ `\ N))$
using *insert* **by**(*simp*)
also have *insert* $(h\ n)\ (h\ `\ N) = h\ `\ \mathit{insert}\ n\ N$ **by** *simp*
finally show *?case* **.**
qed

end

notation *times* (**infixl** $*$ *70*)
notation *Groups.one* (*1*)

20.7 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 *fold-image* things are easy:

definition *card* :: *'a set* ⇒ *nat* **where**
card A = (*if finite A then fold-image* (*op* +) (λ*x*. *1*) *0 A else 0*)

interpretation *card*!: *folding-image-simple op* + *0* λ*x*. *1 card* **proof**
qed (*simp add*: *card-def*)

lemma *card-infinite* [*simp*]:
¬ *finite A* ⟹ *card A* = *0*

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

lemma *card-empty*:
card {} = *0*
by (*fact card.empty*)

lemma *card-insert-disjoint*:
finite A ==> *x* ∉ *A* ==> *card* (*insert x A*) = *Suc* (*card A*)
by *simp*

lemma *card-insert-if*:
finite A ==> *card* (*insert x A*) = (*if x* ∈ *A then card A else Suc* (*card A*))
by *auto* (*simp add*: *card.insert-remove card.remove*)

lemma *card-ge-0-finite*:
card A > *0* ⟹ *finite A*
by (*rule ccontr*) *simp*

lemma *card-0-eq* [*simp*, *no-atp*]:
finite A ⟹ *card A* = *0* ⟷ *A* = {}
by (*auto dest*: *mk-disjoint-insert*)

lemma *finite-UNIV-card-ge-0*:
finite (*UNIV* :: *'a set*) ⟹ *card* (*UNIV* :: *'a set*) > *0*
by (*rule ccontr*) *simp*

lemma *card-eq-0-iff*:
card A = *0* ⟷ *A* = {} ∨ ¬ *finite A*
by *auto*

lemma *card-gt-0-iff*:
0 < *card A* ⟷ *A* ≠ {} ∧ *finite A*
by (*simp add*: *neq0-conv* [*symmetric*] *card-eq-0-iff*)

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*:
assumes *finite B* **and** *A* ⊆ *B*
shows *card A* ≤ *card B*
proof −
from *assms* **have** *finite A* **by** (*auto intro*: *finite-subset*)
then show *?thesis* **using** *assms* **proof** (*induct A arbitrary*: *B*)
case *empty* **then show** *?case* **by** *simp*
next
case (*insert x A*)
then have *x* ∈ *B* **by** *simp*
from *insert* **have** *A* ⊆ *B* − {*x*} **and** *finite* (*B* − {*x*}) **by** *auto*
with *insert.hyps* **have** *card A* ≤ *card* (*B* − {*x*}) **by** *auto*
with ‹*finite A*› ‹*x* ∉ *A*› ‹*finite B*› ‹*x* ∈ *B*› **show** *?case* **by** *simp* (*simp only*: *card.remove*)
qed
qed

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 (*fact card.union-inter* [*symmetric*])

lemma *card-Un-disjoint*: *finite A* ==> *finite B*

```
  ==> A Int B = {} ==> card (A Un B) = card A + card B
 by (fact card.union-disjoint)

lemma card-Diff-subset:
 assumes finite B and B ⊆ A
 shows card (A − B) = card A − card B
proof (cases finite A)
 case False with assms show ?thesis by simp
next
 case True with assms show ?thesis by (induct B arbitrary: A) simp-all
qed

lemma card-Diff-subset-Int:
 assumes AB: finite (A ∩ B) shows card (A − B) = card A − card (A ∩ B)
proof −
 have A − B = A − A ∩ B by auto
 thus ?thesis
  by (simp add: card-Diff-subset AB)
qed

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

lemma finite-psubset-induct[consumes 1, case-names psubset]:
 assumes fin: finite A
 and     major: ⋀A. finite A ⟹ (⋀B. B ⊂ A ⟹ P B) ⟹ P A
```

```
  shows P A
using fin
proof (induct A taking: card rule: measure-induct-rule)
  case (less A)
  have fin: finite A by fact
  have ih: ⋀B. ⟦card B < card A; finite B⟧ ⟹ P B by fact
  { fix B
    assume asm: B ⊂ A
    from asm have card B < card A using psubset-card-mono fin by blast
    moreover
    from asm have B ⊆ A by auto
    then have finite B using fin finite-subset by blast
    ultimately
    have P B using ih by simp
  }
  with fin show P A using major by blast
qed
```

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-Un-disjoint insert-partition
       finite-subset [of - ⋃ (insert x F)])
done

lemma card-eq-UNIV-imp-eq-UNIV:
  assumes fin: finite (UNIV :: 'a set)
  and card: card A = card (UNIV :: 'a set)
  shows A = (UNIV :: 'a set)
proof
  show A ⊆ UNIV by simp
  show UNIV ⊆ A
  proof
    fix x
    show x ∈ A
    proof (rule ccontr)
      assume x ∉ A
      then have A ⊂ UNIV by auto
      with fin have card A < card (UNIV :: 'a set) by (fact psubset-card-mono)
      with card show False by simp
    qed
  qed
qed
```

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* ≠ *0* **using** *assms* **by** *auto*
  ultimately obtain *b* **where** *b*: *b* ∈ *A* **by** *auto*
  show *?thesis*
  proof (*intro exI conjI*)
    show *A* = *insert b* (*A*−{*b*}) **using** *b* **by** *blast*
    show *b* ∉ *A* − {*b*} **by** *blast*
    show *card* (*A* − {*b*}) = *k* **and** *k* = *0* ⟶ *A* − {*b*} = {}
      using *assms b fin* **by**(*fastsimp dest*:*mk-disjoint-insert*)+
  qed
qed

lemma *card-Suc-eq*:
  (*card A* = *Suc k*) =
   (∃ *b B*. *A* = *insert b B* & *b* ∉ *B* & *card B* = *k* & (*k*=*0* ⟶ *B*={}))
apply(*rule iffI*)
 apply(*erule card-eq-SucD*)
apply(*auto*)
apply(*subst card-insert*)
 apply(*auto intro*:*ccontr*)
done

lemma *finite-fun-UNIVD2*:
  assumes *fin*: *finite* (*UNIV* :: (*'a* ⇒ *'b*) *set*)
  shows *finite* (*UNIV* :: *'b set*)
proof −
  from *fin* **have** *finite* (*range* (λ*f* :: *'a* ⇒ *'b*. *f arbitrary*))
    by(*rule finite-imageI*)
  moreover have *UNIV* = *range* (λ*f* :: *'a* ⇒ *'b*. *f arbitrary*)
    by(*rule UNIV-eq-I*) *auto*
  ultimately show *finite* (*UNIV* :: *'b set*) **by** *simp*
qed

lemma *card-UNIV-unit*: *card* (*UNIV* :: *unit set*) = *1*
  unfolding *UNIV-unit* **by** *simp*

20.7.1 Cardinality of image

lemma *card-image-le*: *finite A* ==> *card* (*f* ‘ *A*) <= *card A*
apply (*induct set*: *finite*)
 apply *simp*
apply (*simp add*: *le-SucI card-insert-if*)
done

lemma *card-image*:

assumes *inj-on f A*
shows *card* (*f* ‘ *A*) = *card A*
proof (*cases finite A*)
case *True* **then show** *?thesis* **using** *assms* **by** (*induct A*) *simp-all*
next
case *False* **then have** ¬ *finite* (*f* ‘ *A*) **using** *assms* **by** (*auto dest*: *finite-imageD*)
with *False* **show** *?thesis* **by** *simp*
qed

lemma *bij-betw-same-card*: *bij-betw f A B* ⟹ *card A* = *card B*
by(*auto simp*: *card-image bij-betw-def*)

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-antisym card-inj-on-le*)

20.7.2 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

lemma *card-Plus-conv-if*:
card (*A* <+> *B*) = (*if finite A* ∧ *finite B then card A* + *card B else 0*)
by (*auto simp add*: *card-Plus*)

20.7.3 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. KammÃ¼ller.

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*: *disjoint-eq-subset-Compl*)
done

20.7.4 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*[*iff*]: ~*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*[*no-atp*]:
¬ *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*
qed

end

21 Relation: Relations

theory *Relation*
imports *Datatype Finite-Set*
begin

21.1 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 :: [('*a* * '*b*) *set*, ('*b* * '*c*) *set*] => ('*a* * '*c*) *set*
(**infixr** *O* *75*) **where**
r O s == {(*x*,*z*). *EX y*. (*x*, *y*) : *r* & (*y*, *z*) : *s*}

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 == ⋃*x*∈*A*. {(*x*,*x*)}

definition
Domain :: (′*a* ∗ ′*b*) *set* => ′*a set* **where**
Domain r == {*x*. *EX y*. (*x*,*y*):*r*}

definition
Range :: (′*a* ∗ ′*b*) *set* => ′*b set* **where**
Range r == *Domain*(*r*^−*1*)

definition
Field :: (′*a* ∗ ′*a*) *set* => ′*a set* **where**
Field r == *Domain r* ∪ *Range r*

definition
refl-on :: [′*a set*, (′*a* ∗ ′*a*) *set*] => *bool* **where** — reflexivity over a set
refl-on A r == *r* ⊆ *A* × *A* & (*ALL x*: *A*. (*x*,*x*) : *r*)

abbreviation
refl :: (′*a* ∗ ′*a*) *set* => *bool* **where** — reflexivity over a type
refl == *refl-on UNIV*

definition
sym :: (′*a* ∗ ′*a*) *set* => *bool* **where** — symmetry predicate
sym r == *ALL x y*. (*x*,*y*): *r* −−> (*y*,*x*): *r*

definition
antisym :: (′*a* ∗ ′*a*) *set* => *bool* **where** — antisymmetry predicate
antisym r == *ALL x y*. (*x*,*y*):*r* −−> (*y*,*x*):*r* −−> *x*=*y*

definition
trans :: (′*a* ∗ ′*a*) *set* => *bool* **where** — transitivity predicate
trans r == (*ALL x y z*. (*x*,*y*):*r* −−> (*y*,*z*):*r* −−> (*x*,*z*):*r*)

definition
irrefl :: (′*a* ∗ ′*a*) *set* => *bool* **where**
irrefl r ≡ ∀*x*. (*x*,*x*) ∉ *r*

definition
total-on :: ′*a set* => (′*a* ∗ ′*a*) *set* => *bool* **where**

total-on A r ≡ ∀ *x*∈*A*.∀ *y*∈*A*. *x*≠*y* ⟶ (*x*,*y*)∈*r* ∨ (*y*,*x*)∈*r*

abbreviation *total* ≡ *total-on UNIV*

definition
single-valued :: (*'a* ∗ *'b*) *set* => *bool* **where**
single-valued r == *ALL x y*. (*x*,*y*):*r* −−> (*ALL z*. (*x*,*z*):*r* −−> *y*=*z*)

definition
inv-image :: (*'b* ∗ *'b*) *set* => (*'a* => *'b*) => (*'a* ∗ *'a*) *set* **where**
inv-image r f == {(*x*, *y*). (*f x*, *f y*) : *r*}

21.2 The identity relation

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

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

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

lemma *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*)

21.3 Diagonal: identity over a set

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

lemma *Id-on-eqI*: *a* = *b* ==> *a* : *A* ==> (*a*, *b*) : *Id-on A*
by (*simp add*: *Id-on-def*)

lemma *Id-onI* [*intro*!,*no-atp*]: *a* : *A* ==> (*a*, *a*) : *Id-on A*
by (*rule Id-on-eqI*) (*rule refl*)

lemma *Id-onE* [*elim*!]:
c : *Id-on A* ==> (!!*x*. *x* : *A* ==> *c* = (*x*, *x*) ==> *P*) ==> *P*
— The general elimination rule.

by (*unfold Id-on-def*) (*iprover elim*!: *UN-E singletonE*)

lemma *Id-on-iff*: $((x, y) : Id\text{-}on\ A) = (x = y\ \&\ x : A)$
by *blast*

lemma *Id-on-subset-Times*: $Id\text{-}on\ A \subseteq A \times A$
by *blast*

21.4 Composition of two relations

lemma *rel-compI* [*intro*]:
$(a, b) : r \Longrightarrow (b, c) : s \Longrightarrow (a, c) : r\ O\ s$
by (*unfold rel-comp-def*) *blast*

lemma *rel-compE* [*elim*!]: $xz : r\ O\ s \Longrightarrow$
$(!!x\ y\ z.\ xz = (x, z) \Longrightarrow (x, y) : r \Longrightarrow (y, z) : s \Longrightarrow P) \Longrightarrow P$
by (*unfold rel-comp-def*) (*iprover elim*!: *CollectE splitE exE conjE*)

lemma *rel-compEpair*:
$(a, c) : r\ O\ s \Longrightarrow (!!y.\ (a, y) : r \Longrightarrow (y, c) : s \Longrightarrow P) \Longrightarrow P$
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*:
$r \subseteq A \times B \Longrightarrow s \subseteq B \times C \Longrightarrow (r\ O\ s) \subseteq A \times C$
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*

lemma *rel-comp-UNION-distrib*: $s\ O\ UNION\ I\ r = UNION\ I\ (\%i.\ s\ O\ r\ i)$
by *auto*

lemma *rel-comp-UNION-distrib2*: $UNION\ I\ r\ O\ s = UNION\ I\ (\%i.\ r\ i\ O\ s)$
by *auto*

21.5 Reflexivity

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

lemma *refl-onD*: $refl\text{-}on\ A\ r \Longrightarrow a : A \Longrightarrow (a, a) : r$
by (*unfold refl-on-def*) *blast*

lemma *refl-onD1*: $refl\text{-}on\ A\ r \Longrightarrow (x, y) : r \Longrightarrow x : A$
by (*unfold refl-on-def*) *blast*

lemma *refl-onD2*: $refl\text{-}on\ A\ r \Longrightarrow (x, y) : r \Longrightarrow y : A$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-Int*: $refl\text{-}on\ A\ r \Longrightarrow refl\text{-}on\ B\ s \Longrightarrow refl\text{-}on\ (A \cap B)\ (r \cap s)$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-Un*: $refl\text{-}on\ A\ r \Longrightarrow refl\text{-}on\ B\ s \Longrightarrow refl\text{-}on\ (A \cup B)\ (r \cup s)$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-INTER*:
$ALL\ x{:}S.\ refl\text{-}on\ (A\ x)\ (r\ x) \Longrightarrow refl\text{-}on\ (INTER\ S\ A)\ (INTER\ S\ r)$
by (*unfold refl-on-def*) *fast*

lemma *refl-on-UNION*:
$ALL\ x{:}S.\ refl\text{-}on\ (A\ x)\ (r\ x) \Longrightarrow refl\text{-}on\ (UNION\ S\ A)\ (UNION\ S\ r)$
by (*unfold refl-on-def*) *blast*

lemma *refl-on-empty*[*simp*]: $refl\text{-}on\ \{\}\ \{\}$
by(*simp add*:*refl-on-def*)

lemma *refl-on-Id-on*: $refl\text{-}on\ A\ (Id\text{-}on\ A)$
by (*rule refl-onI* [*OF Id-on-subset-Times Id-onI*])

21.6 Antisymmetry

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

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

lemma *antisym-subset*: *r ⊆ s ==> antisym s ==> antisym r*
by (*unfold antisym-def*) *blast*

lemma *antisym-empty* [*simp*]: *antisym {}*
by (*unfold antisym-def*) *blast*

lemma *antisym-Id-on* [*simp*]: *antisym (Id-on A)*
by (*unfold antisym-def*) *blast*

21.7 Symmetry

lemma *symI*: *(!!a b. (a, b) : r ==> (b, a) : r) ==> sym r*
by (*unfold sym-def*) *iprover*

lemma *symD*: *sym r ==> (a, b) : r ==> (b, a) : r*
by (*unfold sym-def*, *blast*)

lemma *sym-Int*: *sym r ==> sym s ==> sym (r ∩ s)*
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-Un*: *sym r ==> sym s ==> sym (r ∪ s)*
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-INTER*: *ALL x:S. sym (r x) ==> sym (INTER S r)*
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-UNION*: *ALL x:S. sym (r x) ==> sym (UNION S r)*
by (*fast intro*: *symI dest*: *symD*)

lemma *sym-Id-on* [*simp*]: *sym (Id-on A)*
by (*rule symI*) *clarify*

21.8 Transitivity

lemma *transI*:
(!!x y z. (x, y) : r ==> (y, z) : r ==> (x, z) : r) ==> trans r
by (*unfold trans-def*) *iprover*

lemma *transD*: *trans r ==> (a, b) : r ==> (b, c) : r ==> (a, c) : r*
by (*unfold trans-def*) *iprover*

lemma *trans-Int*: *trans r ==> trans s ==> trans (r ∩ s)*
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-INTER*: *ALL x:S. trans (r x) ==> trans (INTER S r)*
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-Id-on* [*simp*]: *trans* (*Id-on A*)
by (*fast intro*: *transI elim*: *transD*)

lemma *trans-diff-Id*: *trans r* ⟹ *antisym r* ⟹ *trans* (*r*−*Id*)
unfolding *antisym-def trans-def* **by** *blast*

21.9 Irreflexivity

lemma *irrefl-diff-Id*[*simp*]: *irrefl*(*r*−*Id*)
by(*simp add*:*irrefl-def*)

21.10 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*)

21.11 Converse

lemma *converse-iff* [*iff*]: ((*a*,*b*): *r*^−*1*) = ((*b*,*a*) : *r*)
by (*simp add*: *converse-def*)

lemma *converseI*[*sym*]: (*a*, *b*) : *r* ==> (*b*, *a*) : *r*^−*1*
by (*simp add*: *converse-def*)

lemma *converseD*[*sym*]: (*a*,*b*) : *r*^−*1* ==> (*b*, *a*) : *r*
by (*simp add*: *converse-def*)

lemma *converseE* [*elim*!]:
yx : *r*^−*1* ==> (!!*x y*. *yx* = (*y*, *x*) ==> (*x*, *y*) : *r* ==> *P*) ==> *P*
— More general than *converseD*, as it "splits" the member of the relation.
by (*unfold converse-def*) (*iprover elim*!: *CollectE splitE bexE*)

lemma *converse-converse* [*simp*]: (*r*^−*1*)^−*1* = *r*
by (*unfold converse-def*) *blast*

lemma *converse-rel-comp*: (*r O s*)^−*1* = *s*^−*1 O r*^−*1*
by *blast*

lemma *converse-Int*: (*r* ∩ *s*)^−*1* = *r*^−*1* ∩ *s*^−*1*
by *blast*

lemma *converse-Un*: (*r* ∪ *s*)^−*1* = *r*^−*1* ∪ *s*^−*1*
by *blast*

lemma *converse-INTER*: (*INTER S r*)^−*1* = (*INT x*:*S*. (*r x*)^−*1*)
by *fast*

lemma *converse-UNION*: $(UNION\ S\ r)\hat{}-1 = (UN\ x{:}S.\ (r\ x)\hat{}-1)$
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* $(r \cup r\hat{}-1)$
by (*unfold sym-def*) *blast*

lemma *sym-Int-converse*: *sym* $(r \cap r\hat{}-1)$
by (*unfold sym-def*) *blast*

lemma *total-on-converse*[*simp*]: *total-on A* $(r\hat{}-1)$ = *total-on A r*
by (*auto simp*: *total-on-def*)

21.12 Domain

declare *Domain-def* [*no-atp*]

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

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

lemma *DomainE* [*elim*!]:
$a : Domain\ r \Longrightarrow (!!y.\ (a,\ y) : r \Longrightarrow P) \Longrightarrow P$
by (*iprover dest*!: *iffD1* [*OF Domain-iff*])

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

lemma *Domain-empty-iff*: $Domain\ r = \{\} \longleftrightarrow r = \{\}$
by *auto*

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*]: $Domain\ (Id\text{-}on\ A) = A$
by *blast*

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

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

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

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

lemma *Domain-converse*[*simp*]: $Domain(r\hat{}-1) = Range\ r$
by(*auto simp*:*Range-def*)

lemma *Domain-mono*: $r \subseteq s \Longrightarrow Domain\ r \subseteq Domain\ s$
by *blast*

lemma *fst-eq-Domain*: $fst\ `\ R = Domain\ R$
by (*auto intro*!:*image-eqI*)

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*

21.13 Range

lemma *Range-iff*: $(a : Range\ r) = (EX\ y.\ (y,\ a) : r)$
by (*simp add*: *Domain-def Range-def*)

lemma *RangeI* [*intro*]: $(a,\ b) : r \Longrightarrow b : Range\ r$
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-empty-iff*: *Range r* = {} ⟷ *r* = {}
by *auto*

lemma *Range-insert*: *Range* (*insert* (*a*, *b*) *r*) = *insert b* (*Range r*)
by *blast*

lemma *Range-Id* [*simp*]: *Range Id* = *UNIV*
by *blast*

lemma *Range-Id-on* [*simp*]: *Range* (*Id-on A*) = *A*
by *auto*

lemma *Range-Un-eq*: *Range*(*A* ∪ *B*) = *Range*(*A*) ∪ *Range*(*B*)
by *blast*

lemma *Range-Int-subset*: *Range*(*A* ∩ *B*) ⊆ *Range*(*A*) ∩ *Range*(*B*)
by *blast*

lemma *Range-Diff-subset*: *Range*(*A*) − *Range*(*B*) ⊆ *Range*(*A* − *B*)
by *blast*

lemma *Range-Union*: *Range* (*Union S*) = (⋃ *A*∈*S*. *Range A*)
by *blast*

lemma *Range-converse*[*simp*]: *Range*(*r*^−*1*) = *Domain r*
by *blast*

lemma *snd-eq-Range*: *snd* ' *R* = *Range R*
by (*auto intro*!:*image-eqI*)

21.14 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* $(\bigcup R) = \bigcup(Field\ `\ R)$
by(*auto simp*:*Field-def*)

lemma *Field-converse*[*simp*]: *Field*$(r\hat{}-1) = Field\ r$
by(*auto simp*:*Field-def*)

21.15 Image of a set under a relation

declare *Image-def* [*no-atp*]

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*,*no-atp*]: $(a,\ b) : r \Longrightarrow a : A \Longrightarrow b : r``A$
by (*unfold Image-def*) *blast*

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

lemma *rev-ImageI*: $a : A \Longrightarrow (a,\ b) : r \Longrightarrow b : r\ ``\ A$
— This version's more effective when we already have the required a
by *blast*

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

lemma *Image-Id* [*simp*]: $Id\ ``\ A = A$
by *blast*

lemma *Image-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*:
$single\text{-}valued\ (converse\ R) \Longrightarrow R\ ``\ (A \cap B) = R\ ``\ A \cap R\ ``\ B$
by (*simp add*: *single-valued-def*, *blast*)

lemma *Image-Un*: $R\ ``\ (A \cup B) = R\ ``\ A \cup R\ ``\ B$

by *blast*

lemma *Un-Image*: $(R \cup S)$ `` $A = R$ `` $A \cup S$ `` A
by *blast*

lemma *Image-subset*: $r \subseteq A \times B$ ==> $r``C \subseteq B$
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*

21.16 Single valued relations

lemma *single-valuedI*:
$ALL\ x\ y.\ (x,y){:}r \longrightarrow (ALL\ z.\ (x,z){:}r \longrightarrow y{=}z)$ ==> *single-valued* r
by (*unfold single-valued-def*)

lemma *single-valuedD*:
single-valued r ==> $(x, y) : r$ ==> $(x, z) : r$ ==> $y = z$
by (*simp add*: *single-valued-def*)

lemma *single-valued-rel-comp*:
single-valued r ==> *single-valued* s ==> *single-valued* $(r\ O\ s)$
by (*unfold single-valued-def*) *blast*

lemma *single-valued-subset*:
$r \subseteq s$ ==> *single-valued* s ==> *single-valued* r
by (*unfold single-valued-def*) *blast*

lemma *single-valued-Id* [*simp*]: *single-valued Id*
by (*unfold single-valued-def*) *blast*

lemma *single-valued-Id-on* [*simp*]: *single-valued* (*Id-on A*)
by (*unfold single-valued-def*) *blast*

21.17 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*

21.18 Inverse image

lemma *sym-inv-image*: *sym r* ==> *sym* (*inv-image r f*)
by (*unfold sym-def inv-image-def*) *blast*

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

lemma *in-inv-image*[*simp*]: ((*x*,*y*) : *inv-image r f*) = ((*f x*, *f y*) : *r*)
by (*auto simp*:*inv-image-def*)

lemma *converse-inv-image*[*simp*]: (*inv-image R f*)^-1 = *inv-image* (*R*^-1) *f*
unfolding *inv-image-def converse-def* **by** *auto*

21.19 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

lemma *finite-Domain*: *finite r ==> finite* (*Domain r*)
by (*induct set*: *finite*) (*auto simp add*: *Domain-insert*)

lemma *finite-Range*: *finite r ==> finite* (*Range r*)
by (*induct set*: *finite*) (*auto simp add*: *Range-insert*)

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

21.20 Miscellaneous

Version of *lfp-induct* for binary relations

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

Version of *subsetI* for binary relations

lemma *subrelI*: $(\bigwedge x\ y.\ (x, y) \in r \Longrightarrow (x, y) \in s) \Longrightarrow r \subseteq s$
by *auto*

end

22 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 (⊥)

22.1 Predicates as (complete) lattices

Handy introduction and elimination rules for ≤ 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 *rev-predicate1D*:
$P\ x$ ==> P <= Q ==> $Q\ x$
by (*rule predicate1D*)

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

22.1.1 Equality

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*)

22.1.2 Order relation

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*

22.1.3 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*)

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*)

22.1.4 Binary union

lemma *sup1I1* [*elim?*]: *A x* $\Longrightarrow$ *sup A B x*
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup2I1* [*elim?*]: *A x y* $\Longrightarrow$ *sup A B x y*
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup1I2* [*elim?*]: *B x* $\Longrightarrow$ *sup A B x*
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup2I2* [*elim?*]: *B x y* $\Longrightarrow$ *sup A B x y*
by (*simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup1E* [*elim*!]: *sup A B x* ==> (*A x* ==> *P*) ==> (*B x* ==> *P*) ==> *P*
by (*simp add*: *sup-fun-eq sup-bool-eq*) *iprover*

lemma *sup2E* [*elim*!]: *sup A B x y* ==> (*A x y* ==> *P*) ==> (*B x y* ==> *P*) ==> *P*
by (*simp add*: *sup-fun-eq sup-bool-eq*) *iprover*

Classical introduction rule: no commitment to *A* vs *B*.

lemma *sup1CI* [*intro*!]: ($\sim$ *B x* ==> *A x*) ==> *sup A B x*

by (*auto simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup2CI* [*intro*!]: (~ *B x y* ==> *A x y*) ==> *sup A B x y*
by (*auto simp add*: *sup-fun-eq sup-bool-eq*)

lemma *sup-Un-eq*: *sup* (λ*x*. *x* ∈ *R*) (λ*x*. *x* ∈ *S*) = (λ*x*. *x* ∈ *R* ∪ *S*)
by (*simp add*: *sup-fun-eq sup-bool-eq mem-def*)

lemma *sup-Un-eq2* [*pred-set-conv*]: *sup* (λ*x y*. (*x*, *y*) ∈ *R*) (λ*x y*. (*x*, *y*) ∈ *S*) = (λ*x y*. (*x*, *y*) ∈ *R* ∪ *S*)
by (*simp add*: *sup-fun-eq sup-bool-eq mem-def*)

22.1.5 Binary intersection

lemma *inf1I* [*intro*!]: *A x* ==> *B x* ==> *inf A B x*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf2I* [*intro*!]: *A x y* ==> *B x y* ==> *inf A B x y*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf1E* [*elim*!]: *inf A B x* ==> (*A x* ==> *B x* ==> *P*) ==> *P*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf2E* [*elim*!]: *inf A B x y* ==> (*A x y* ==> *B x y* ==> *P*) ==> *P*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf1D1*: *inf A B x* ==> *A x*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf2D1*: *inf A B x y* ==> *A x y*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf1D2*: *inf A B x* ==> *B x*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf2D2*: *inf A B x y* ==> *B x y*
by (*simp add*: *inf-fun-eq inf-bool-eq*)

lemma *inf-Int-eq*: *inf* (λ*x*. *x* ∈ *R*) (λ*x*. *x* ∈ *S*) = (λ*x*. *x* ∈ *R* ∩ *S*)
by (*simp add*: *inf-fun-eq inf-bool-eq mem-def*)

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*: *inf-fun-eq inf-bool-eq mem-def*)

22.1.6 Unions of families

lemma *SUP1-iff*: (*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*: $(SUP\ x{:}A.\ B\ x)\ b\ c = (EX\ x{:}A.\ B\ x\ b\ c)$
by (*simp add*: *SUPR-def Sup-fun-def Sup-bool-def*) *blast*

lemma *SUP1-I* [*intro*]: $a : A \Longrightarrow B\ a\ b \Longrightarrow (SUP\ x{:}A.\ B\ x)\ b$
by (*auto simp add*: *SUP1-iff*)

lemma *SUP2-I* [*intro*]: $a : A \Longrightarrow B\ a\ b\ c \Longrightarrow (SUP\ x{:}A.\ B\ x)\ b\ c$
by (*auto simp add*: *SUP2-iff*)

lemma *SUP1-E* [*elim*!]: $(SUP\ x{:}A.\ B\ x)\ b \Longrightarrow (!!x.\ x : A \Longrightarrow B\ x\ b \Longrightarrow R) \Longrightarrow R$
by (*auto simp add*: *SUP1-iff*)

lemma *SUP2-E* [*elim*!]: $(SUP\ x{:}A.\ B\ x)\ b\ c \Longrightarrow (!!x.\ x : A \Longrightarrow B\ x\ b\ c \Longrightarrow R) \Longrightarrow R$
by (*auto simp add*: *SUP2-iff*)

lemma *SUP-UN-eq*: $(SUP\ i.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (UN\ i.\ r\ i))$
by (*simp add*: *SUP1-iff expand-fun-eq*)

lemma *SUP-UN-eq2*: $(SUP\ i.\ (\lambda x\ y.\ (x, y) \in r\ i)) = (\lambda x\ y.\ (x, y) \in (UN\ i.\ r\ i))$
by (*simp add*: *SUP2-iff expand-fun-eq*)

22.1.7 Intersections of families

lemma *INF1-iff*: $(INF\ x{:}A.\ B\ x)\ b = (ALL\ x{:}A.\ B\ x\ b)$
by (*simp add*: *INFI-def Inf-fun-def Inf-bool-def*) *blast*

lemma *INF2-iff*: $(INF\ x{:}A.\ B\ x)\ b\ c = (ALL\ x{:}A.\ B\ x\ b\ c)$
by (*simp add*: *INFI-def Inf-fun-def Inf-bool-def*) *blast*

lemma *INF1-I* [*intro*!]: $(!!x.\ x : A \Longrightarrow B\ x\ b) \Longrightarrow (INF\ x{:}A.\ B\ x)\ b$
by (*auto simp add*: *INF1-iff*)

lemma *INF2-I* [*intro*!]: $(!!x.\ x : A \Longrightarrow B\ x\ b\ c) \Longrightarrow (INF\ x{:}A.\ B\ x)\ b\ c$
by (*auto simp add*: *INF2-iff*)

lemma *INF1-D* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b \Longrightarrow a : A \Longrightarrow B\ a\ b$
by (*auto simp add*: *INF1-iff*)

lemma *INF2-D* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b\ c \Longrightarrow a : A \Longrightarrow B\ a\ b\ c$
by (*auto simp add*: *INF2-iff*)

lemma *INF1-E* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b \Longrightarrow (B\ a\ b \Longrightarrow R) \Longrightarrow (a\ {}^{\sim}{:}\ A \Longrightarrow R) \Longrightarrow R$
by (*auto simp add*: *INF1-iff*)

lemma *INF2-E* [*elim*]: $(INF\ x{:}A.\ B\ x)\ b\ c \Longrightarrow (B\ a\ b\ c \Longrightarrow R) \Longrightarrow (a\ {}^{\sim}{:}\ A \Longrightarrow R) \Longrightarrow R$

by (*auto simp add*: *INF2-iff*)

lemma *INF-INT-eq*: (*INF i*. (λ*x*. *x* ∈ *r i*)) = (λ*x*. *x* ∈ (*INT i*. *r i*))
by (*simp add*: *INF1-iff expand-fun-eq*)

lemma *INF-INT-eq2*: (*INF i*. (λ*x y*. (*x*, *y*) ∈ *r i*)) = (λ*x y*. (*x*, *y*) ∈ (*INT i*. *r i*))
by (*simp add*: *INF2-iff expand-fun-eq*)

22.2 Predicates as relations

22.2.1 Composition

inductive
pred-comp :: ['*a* => '*b* => *bool*, '*b* => '*c* => *bool*] => '*a* => '*c* => *bool*
(**infixr** *OO* *75*)
for *r* :: '*a* => '*b* => *bool* **and** *s* :: '*b* => '*c* => *bool*
where
pred-compI [*intro*]: *r a b* ==> *s b c* ==> (*r OO s*) *a c*

inductive-cases *pred-compE* [*elim*!]: (*r OO s*) *a c*

lemma *pred-comp-rel-comp-eq* [*pred-set-conv*]:
((λ*x y*. (*x*, *y*) ∈ *r*) *OO* (λ*x y*. (*x*, *y*) ∈ *s*)) = (λ*x y*. (*x*, *y*) ∈ *r O s*)
by (*auto simp add*: *expand-fun-eq elim*: *pred-compE*)

22.2.2 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*]:
(λ*x y*. (*x*, *y*) ∈ *r*)^--*1* = (λ*x y*. (*x*, *y*) ∈ *r*^-*1*)
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-conversep* [*simp*]: (*r*^--*1*)^--*1* = *r*

by (*iprover intro*: *order-antisym conversepI dest*: *conversepD*)

lemma *converse-pred-comp*: $(r\ OO\ s)\hat{}{-}{-}1 = s\hat{}{-}{-}1\ OO\ r\hat{}{-}{-}1$
by (*iprover intro*: *order-antisym conversepI pred-compI*
elim: *pred-compE dest*: *conversepD*)

lemma *converse-meet*: $(inf\ r\ s)\hat{}{-}{-}1 = inf\ r\hat{}{-}{-}1\ s\hat{}{-}{-}1$
by (*simp add*: *inf-fun-eq inf-bool-eq*)
(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *converse-join*: $(sup\ r\ s)\hat{}{-}{-}1 = sup\ r\hat{}{-}{-}1\ s\hat{}{-}{-}1$
by (*simp add*: *sup-fun-eq sup-bool-eq*)
(*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *conversep-noteq* [*simp*]: $(op\ \tilde{}{=})\hat{}{-}{-}1 = op\ \tilde{}{=}$
by (*auto simp add*: *expand-fun-eq*)

lemma *conversep-eq* [*simp*]: $(op\ =)\hat{}{-}{-}1 = op\ =$
by (*auto simp add*: *expand-fun-eq*)

22.2.3 Domain

inductive
DomainP :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$
for $r :: 'a \Rightarrow 'b \Rightarrow bool$
where
DomainPI [*intro*]: $r\ a\ b \Longrightarrow DomainP\ r\ a$

inductive-cases *DomainPE* [*elim*!]: *DomainP r a*

lemma *DomainP-Domain-eq* [*pred-set-conv*]: $DomainP\ (\lambda x\ y.\ (x, y) \in r) = (\lambda x.\ x \in Domain\ r)$
by (*blast intro*!: *Orderings.order-antisym predicate1I*)

22.2.4 Range

inductive
RangeP :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'b \Rightarrow bool$
for $r :: 'a \Rightarrow 'b \Rightarrow bool$
where
RangePI [*intro*]: $r\ a\ b \Longrightarrow RangeP\ r\ b$

inductive-cases *RangePE* [*elim*!]: *RangeP r b*

lemma *RangeP-Range-eq* [*pred-set-conv*]: $RangeP\ (\lambda x\ y.\ (x, y) \in r) = (\lambda x.\ x \in Range\ r)$
by (*blast intro*!: *Orderings.order-antisym predicate1I*)

22.2.5 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*)

22.2.6 Powerset

definition *Powp* :: ('*a* ⇒ *bool*) ⇒ '*a set* ⇒ *bool* **where**
Powp A == λ*B*. ∀ *x* ∈ *B*. *A x*

lemma *Powp-Pow-eq* [*pred-set-conv*]: *Powp* (λ*x*. *x* ∈ *A*) = (λ*x*. *x* ∈ *Pow A*)
by (*auto simp add*: *Powp-def expand-fun-eq*)

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred pred-subset-eq*]

22.2.7 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*}

22.3 Predicates as enumerations

22.3.1 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*, *boolean-algebra*}
begin

definition
P ≤ Q ⟷ eval P ≤ eval Q

definition
P < Q ⟷ eval P < eval Q

definition
⊥ = Pred ⊥

definition
⊤ = Pred ⊤

definition
P ⊓ Q = Pred (eval P ⊓ eval Q)

definition
P ⊔ Q = Pred (eval P ⊔ eval Q)

definition
[*code del*]: *⨅ A = Pred (INFI A eval)*

definition
[*code del*]: *⨆ A = Pred (SUPR A eval)*

definition
- P = Pred (- eval P)

definition
P - Q = Pred (eval P - eval Q)

instance proof
qed (*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 uminus-pred-def minus-pred-def fun-Compl-def bool-Compl-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 ⤜ Q) ⤜ R = P ⤜ (λx. Q x ⤜ R)

by (*auto simp add*: *bind-def expand-fun-eq*)

lemma *bind-single*:
$P \gg= single = P$
by (*simp add*: *bind-def single-def*)

lemma *single-bind*:
$single\ x \gg= P = P\ x$
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 SUP1-iff 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 add*: *sup-fun-eq sup-bool-eq*)

lemma *supI2*: *eval B x* $\Longrightarrow$ *eval* (*A* $\sqcup$ *B*) *x*
unfolding *sup-pred-def* **by** (*simp add*: *sup-fun-eq sup-bool-eq*)

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*

lemma *single-not-bot* [*simp*]:
single x $\neq$ $\bot$
by (*auto simp add*: *single-def bot-pred-def expand-fun-eq*)

lemma *not-bot*:
assumes *A* $\neq$ $\bot$
obtains *x* **where** *eval A x*
using *assms* **by** (*cases A*)
(*auto simp add*: *bot-pred-def*, *auto simp add*: *mem-def*)

22.3.2 Emptiness check and definite choice

definition *is-empty* :: *'a pred* $\Rightarrow$ *bool* **where**
is-empty A $\longleftrightarrow$ *A* = $\bot$

lemma *is-empty-bot*:
is-empty $\bot$
by (*simp add*: *is-empty-def*)

lemma *not-is-empty-single*:
$\neg$ *is-empty* (*single x*)
by (*auto simp add*: *is-empty-def single-def bot-pred-def expand-fun-eq*)

lemma *is-empty-sup*:
is-empty (*A* $\sqcup$ *B*) $\longleftrightarrow$ *is-empty A* $\wedge$ *is-empty B*
by (*auto simp add*: *is-empty-def*)

definition *singleton* :: (*unit* => *'a*) $\Rightarrow$ *'a pred* $\Rightarrow$ *'a* **where**
singleton dfault A = (*if* $\exists!x$. *eval A x then THE x*. *eval A x else dfault* ())

lemma *singleton-eqI*:
$\exists!x$. *eval A x* $\Longrightarrow$ *eval A x* $\Longrightarrow$ *singleton dfault A* = *x*
by (*auto simp add*: *singleton-def*)

lemma *eval-singletonI*:
$\exists!x$. *eval A x* $\Longrightarrow$ *eval A* (*singleton dfault A*)
proof −

assume *assm*: $\exists!x.\ eval\ A\ x$
then obtain x **where** *eval A x* **..**
moreover with *assm* **have** *singleton dfault A* $= x$ **by** (*rule singleton-eqI*)
ultimately show *?thesis* **by** *simp*
qed

lemma *single-singleton*:
$\exists!x.\ eval\ A\ x \Longrightarrow single\ (singleton\ dfault\ A) = A$
proof −
assume *assm*: $\exists!x.\ eval\ A\ x$
then have *eval A* (*singleton dfault A*)
by (*rule eval-singletonI*)
moreover from *assm* **have** $\bigwedge x.\ eval\ A\ x \Longrightarrow singleton\ dfault\ A = x$
by (*rule singleton-eqI*)
ultimately have *eval* (*single* (*singleton dfault A*)) = *eval A*
by (*simp* (*no-asm-use*) *add*: *single-def expand-fun-eq*) *blast*
then show *?thesis* **by** (*simp add*: *eval-inject*)
qed

lemma *singleton-undefinedI*:
$\neg\ (\exists!x.\ eval\ A\ x) \Longrightarrow singleton\ dfault\ A = dfault\ ()$
by (*simp add*: *singleton-def*)

lemma *singleton-bot*:
singleton dfault $\bot$ = *dfault* ()
by (*auto simp add*: *bot-pred-def intro*: *singleton-undefinedI*)

lemma *singleton-single*:
singleton dfault (*single x*) = *x*
by (*auto simp add*: *intro*: *singleton-eqI singleI elim*: *singleE*)

lemma *singleton-sup-single-single*:
singleton dfault (*single x* $\sqcup$ *single y*) = (*if x* = *y then x else dfault* ())
proof (*cases x* = *y*)
case *True* **then show** *?thesis* **by** (*simp add*: *singleton-single*)
next
case *False*
have *eval* (*single x* $\sqcup$ *single y*) *x*
and *eval* (*single x* $\sqcup$ *single y*) *y*
by (*auto intro*: *supI1 supI2 singleI*)
with *False* **have** $\neg\ (\exists!z.\ eval\ (single\ x \sqcup single\ y)\ z)$
by *blast*
then have *singleton dfault* (*single x* $\sqcup$ *single y*) = *dfault* ()
by (*rule singleton-undefinedI*)
with *False* **show** *?thesis* **by** *simp*
qed

lemma *singleton-sup-aux*:
singleton dfault (*A* $\sqcup$ *B*) = (*if A* = $\bot$ *then singleton dfault B*

```
    else if B = ⊥ then singleton dfault A
    else singleton dfault
      (single (singleton dfault A) ⊔ single (singleton dfault B)))
proof (cases (∃!x. eval A x) ∧ (∃!y. eval B y))
  case True then show ?thesis by (simp add: single-singleton)
next
  case False
  from False have A-or-B:
    singleton dfault A = dfault () ∨ singleton dfault B = dfault ()
    by (auto intro!: singleton-undefinedI)
  then have rhs: singleton dfault
    (single (singleton dfault A) ⊔ single (singleton dfault B)) = dfault ()
    by (auto simp add: singleton-sup-single-single singleton-single)
  from False have not-unique:
    ¬ (∃!x. eval A x) ∨ ¬ (∃!y. eval B y) by simp
  show ?thesis proof (cases A ≠ ⊥ ∧ B ≠ ⊥)
    case True
    then obtain a b where a: eval A a and b: eval B b
      by (blast elim: not-bot)
    with True not-unique have ¬ (∃!x. eval (A ⊔ B) x)
      by (auto simp add: sup-pred-def bot-pred-def)
    then have singleton dfault (A ⊔ B) = dfault () by (rule singleton-undefinedI)
    with True rhs show ?thesis by simp
  next
    case False then show ?thesis by auto
  qed
qed

lemma singleton-sup:
  singleton dfault (A ⊔ B) = (if A = ⊥ then singleton dfault B
    else if B = ⊥ then singleton dfault A
    else if singleton dfault A = singleton dfault B then singleton dfault A else dfault
())
using singleton-sup-aux [of dfault A B] by (simp only: singleton-sup-single-single)
```

22.3.3 Derived operations

```
definition if-pred :: bool ⇒ unit pred where
  if-pred-eq: if-pred b = (if b then single () else ⊥)

definition holds :: unit pred ⇒ bool where
  holds-eq: holds P = eval P ()

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*)
lemma *f* () = *False* ∨ *f* () = *True*
by *simp*

lemma *closure-of-bool-cases*:
assumes (*f* :: *unit* ⇒ *bool*) = (%*u*. *False*) ⟹ *P f*
assumes *f* = (%*u*. *True*) ⟹ *P f*
shows *P f*
proof −
have *f* = (%*u*. *False*) ∨ *f* = (%*u*. *True*)
apply (*cases f* ())
apply (*rule disjI2*)
apply (*rule ext*)
apply (*simp add*: *unit-eq*)
apply (*rule disjI1*)
apply (*rule ext*)
apply (*simp add*: *unit-eq*)
done
from *this prems* **show** *?thesis* **by** *blast*
qed

lemma *unit-pred-cases*:
assumes *P* ⊥
assumes *P* (*single* ())
shows *P Q*
using *assms*
unfolding *bot-pred-def Collect-def empty-def single-def*
apply (*cases Q*)
apply *simp*
apply (*rule-tac f*=*fun* **in** *closure-of-bool-cases*)
apply *auto*
apply (*subgoal-tac* (%*x*. () = *x*) = (%*x*. *True*))
apply *auto*

done

lemma *holds-if-pred*:
holds (*if-pred b*) = *b*
unfolding *if-pred-eq holds-eq*
by (*cases b*) (*auto intro*: *singleI elim*: *botE*)

lemma *if-pred-holds*:
if-pred (*holds P*) = *P*
unfolding *if-pred-eq holds-eq*
by (*rule unit-pred-cases*) (*auto intro*: *singleI elim*: *botE*)

lemma *is-empty-holds*:
is-empty P ⟷ ¬ *holds P*
unfolding *is-empty-def holds-eq*
by (*rule unit-pred-cases*) (*auto elim*: *botE intro*: *singleI*)

22.3.4 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* (λ*u*. *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* (λ*u*. *apply f* (*g* ()))
unfolding *Seq-def* **by** (*rule sym*, *rule apply-bind*)

lemma *bot-set-code* [*code*]:
⊥ = *Seq* (λ*u*. *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 ⊥])
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 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

inductive eq :: 'a ⇒ 'a ⇒ bool where eq x x
```

lemma *eq-is-eq*: *eq x y* ≡ (*x* = *y*)
by (*rule eq-reflection*) (*auto intro*: *eq.intros elim*: *eq.cases*)

definition *map* :: ('*a* ⇒ '*b*) ⇒ '*a pred* ⇒ '*b pred* **where**
map f P = *P* ⤜ (*single* ∘ *f*)

primrec *null* :: '*a seq* ⇒ *bool* **where**
null Empty ⟷ *True*
| *null* (*Insert x P*) ⟷ *False*
| *null* (*Join P xq*) ⟷ *is-empty P* ∧ *null xq*

lemma *null-is-empty*:
null xq ⟷ *is-empty* (*pred-of-seq xq*)
by (*induct xq*) (*simp-all add*: *is-empty-bot not-is-empty-single is-empty-sup*)

lemma *is-empty-code* [*code*]:
is-empty (*Seq f*) ⟷ *null* (*f* ())
by (*simp add*: *null-is-empty Seq-def*)

primrec *the-only* :: (*unit* ⇒ '*a*) ⇒ '*a seq* ⇒ '*a* **where**
[*code del*]: *the-only dfault Empty* = *dfault* ()
| *the-only dfault* (*Insert x P*) = (*if is-empty P then x else let y* = *singleton dfault P in if x* = *y then x else dfault* ())
| *the-only dfault* (*Join P xq*) = (*if is-empty P then the-only dfault xq else if null xq then singleton dfault P*
else let x = *singleton dfault P*; *y* = *the-only dfault xq in*
if x = *y then x else dfault* ())

lemma *the-only-singleton*:
the-only dfault xq = *singleton dfault* (*pred-of-seq xq*)
by (*induct xq*)
(*auto simp add*: *singleton-bot singleton-single is-empty-def*
null-is-empty Let-def singleton-sup)

lemma *singleton-code* [*code*]:
singleton dfault (*Seq f*) = (*case f* ()
of Empty ⇒ *dfault* ()
| *Insert x P* ⇒ *if is-empty P then x*
else let y = *singleton dfault P in*
if x = *y then x else dfault* ()
| *Join P xq* ⇒ *if is-empty P then the-only dfault xq*
else if null xq then singleton dfault P
else let x = *singleton dfault P*; *y* = *the-only dfault xq in*
if x = *y then x else dfault* ())
by (*cases f* ())
(*auto simp add*: *Seq-def the-only-singleton is-empty-def*
null-is-empty singleton-bot singleton-single singleton-sup Let-def)

definition *not-unique* :: '*a pred* => '*a*

where
  [*code del*]: *not-unique A* = (*THE x*. *eval A x*)

definition *the* :: *'a pred* => *'a*
where
  [*code del*]: *the A* = (*THE x*. *eval A x*)

lemma *the-eq*[*code*]: *the A* = *singleton* (λ*x*. *not-unique A*) *A*
by (*auto simp add*: *the-def singleton-def not-unique-def*)

code-abort *not-unique*

code-reflect *Predicate*
  datatypes *pred* = *Seq* **and** *seq* = *Empty* | *Insert* | *Join*
  functions *map*

ML ⟪
signature PREDICATE =
sig
  datatype 'a pred = *Seq of* (*unit* -> *'a seq*)
  and 'a seq = *Empty* | *Insert of 'a* ∗ *'a pred* | *Join of 'a pred* ∗ *'a seq*
  val yield: *'a pred* -> (*'a* ∗ *'a pred*) *option*
  val yieldn: *int* -> *'a pred* -> *'a list* ∗ *'a pred*
  val map: (*'a* -> *'b*) -> *'a pred* -> *'b pred*
end;

structure Predicate : *PREDICATE* =
struct

datatype pred = *datatype Predicate.pred*
datatype seq = *datatype Predicate.seq*

fun map f = *Predicate.map f*;

fun yield (*Seq f*) = *next* (*f* ())
and next Empty = *NONE*
  | *next* (*Insert* (*x*, *P*)) = *SOME* (*x*, *P*)
  | *next* (*Join* (*P*, *xq*)) = (*case yield P*
    of NONE => *next xq*
     | *SOME* (*x*, *Q*) => *SOME* (*x*, *Seq* (*fn* - => *Join* (*Q*, *xq*))));

fun anamorph f k x = (*if k* = *0 then* ([], *x*)
  else case f x
   of NONE => ([], *x*)
    | *SOME* (*v*, *y*) => *let*
      val (*vs*, *z*) = *anamorph f* (*k* − *1*) *y*
     in (*v* :: *vs*, *z*) *end*);

fun yieldn P = *anamorph yield P*;

end;
⟩⟩

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-type (**open**) *pred seq*
hide-const (**open**) *Pred eval single bind is-empty singleton if-pred not-pred holds Empty Insert Join Seq member pred-of-seq apply adjunct null the-only eq map not-unique the*

end

23 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*^+

declare *rtrancl-def* [*nitpick-def del*]

rtranclp-def [*nitpick-def del*]
trancl-def [*nitpick-def del*]
tranclp-def [*nitpick-def del*]

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*)

23.1 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*

23.2 Reflexive-transitive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: (*sup* (λ*x y*. (*x*, *y*) ∈ *r*) *op* =) = (λ*x y*. (*x*, *y*) ∈ *r* ∪ *Id*)
by (*auto 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* **using** *a*
by (*induct x*≡*a b*) (*rule cases*)+

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^* ∩ s) O r ⊆ 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*: $(inf \ r \ op \ \sim=)\hat{}** = r\hat{}**$
apply (*rule sym*)
apply (*rule rtranclp-subset*)
apply *blast+*
done

theorem *rtranclp-converseD*:
assumes *r*: $(r\hat{}--1)\hat{}** \ x \ y$
shows $r\hat{}** \ y \ x$
proof −
from *r* **show** *?thesis*

```
  by induct (iprover intro: rtranclp-trans dest!: conversepD)+
qed

lemmas rtrancl-converseD = rtranclp-converseD [to-set]

theorem rtranclp-converseI:
  assumes r^** 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, case-names base step]:
  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 [consumes 1, case-names base step]:
  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*)

lemma *rtrancl-Un-separatorE*:
(*a*,*b*) : (*P* ∪ *Q*)^* ⟹ ∀ *x y*. (*a*,*x*) : *P*^* ⟶ (*x*,*y*) : *Q* ⟶ *x*=*y* ⟹ (*a*,*b*) : *P*^*
apply (*induct rule*:*rtrancl.induct*)
apply *blast*
apply (*blast intro*:*rtrancl-trans*)
done

lemma *rtrancl-Un-separator-converseE*:
(*a*,*b*) : (*P* ∪ *Q*)^* ⟹ ∀ *x y*. (*x*,*b*) : *P*^* ⟶ (*y*,*x*) : *Q* ⟶ *y*=*x* ⟹ (*a*,*b*) : *P*^*
apply (*induct rule*:*converse-rtrancl-induct*)
apply *blast*
apply (*blast intro*:*rtrancl-trans*)
done

lemma *Image-closed-trancl*:
assumes *r* `` *X* ⊆ *X* **shows** *r*^\* `` *X* = *X*
proof −
from *assms* **have** ∗∗: {*y*. ∃ *x*∈*X*. (*x*, *y*) ∈ *r*} ⊆ *X* **by** *auto*
have ⋀*x y*. (*y*, *x*) ∈ *r*^* ⟹ *y* ∈ *X* ⟹ *x* ∈ *X*
proof −
fix *x y*
assume ∗: *y* ∈ *X*
assume (*y*, *x*) ∈ *r*^*
then show *x* ∈ *X*
proof *induct*
case *base* **show** *?case* **by** (*fact* ∗)
next
case *step* **with** ∗∗ **show** *?case* **by** *auto*
qed
qed
then show *?thesis* **by** *auto*
qed

23.3 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 *a*: *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* **using** *a*
by (*induct x*≡*a b*) (*iprover intro*: *cases*)+

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^+ ∩ s) O r ⊆ s|] ==> r^+ ⊆ s
  apply (rule subsetI)
  apply (rule-tac p = x in PairE)
  apply clarify
  apply (erule trancl-induct)
   apply auto
  done

lemma trancl-unfold: r^+ = r Un r^+ O r
  by (auto intro: trancl-into-trancl elim: tranclE)
```

Transitivity of r^+

```
lemma trans-trancl [simp]: trans (r^+)
proof (rule transI)
  fix x y z
  assume (x, y) ∈ r^+
  assume (y, z) ∈ r^+
  then show (x, z) ∈ r^+
  proof induct
    case (base u)
    from ‹(x, y) ∈ r^+› and ‹(y, u) ∈ r›
    show (x, u) ∈ r^+ ..
  next
    case (step u v)
    from ‹(x, u) ∈ r^+› and ‹(u, v) ∈ r›
    show (x, v) ∈ r^+ ..
  qed
qed

lemmas trancl-trans = trans-trancl [THEN transD, standard]

lemma tranclp-trans:
  assumes xy: r^++ x y
  and yz: r^++ y z
```

shows *r^++ x z* **using** *yz xy*
by *induct iprover+*

lemma *trancl-id* [*simp*]: *trans r ⟹ r^+ = r*
apply *auto*
apply (*erule trancl-induct*)
apply *assumption*
apply (*unfold trans-def*)
apply *blast*
done

lemma *rtranclp-tranclp-tranclp*:
assumes *r^** x y*
shows *!!z. r^++ y z ==> r^++ x z* **using** *assms*
by *induct* (*iprover intro*: *tranclp-trans*)+

lemmas *rtrancl-trancl-trancl = rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: *r a b ==> r^++ b c ==> r^++ a c*
by (*erule tranclp-trans* [*OF tranclp.r-into-trancl*])

lemmas *trancl-into-trancl2 = tranclp-into-tranclp2* [*to-set*]

lemma *trancl-insert*:
(*insert* (*y*, *x*) *r*)*^+ = r^+ ∪ {(a, b). (a, y) ∈ r^* ∧ (x, b) ∈ r^**}
— primitive recursion for *trancl* over finite relations
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*simp only*: *split-tupled-all*)
apply (*erule trancl-induct*, *blast*)
apply (*blast intro*: *rtrancl-into-trancl1 trancl-into-rtrancl 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*: $(r\hat{}--1)\hat{}++ = (r\hat{}++)\hat{}--1$
by (*fastsimp simp add*: *expand-fun-eq*
intro!: *tranclp-converseI dest*!: *tranclp-converseD*)

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

lemma *sym-trancl*: *sym r* ==> *sym* $(r\hat{}+)$
by (*simp only*: *sym-conv-converse-eq trancl-converse* [*symmetric*])

lemma *converse-tranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: $r\hat{}++$ *a b*
and *cases*: !!*y*. *r y b* ==> *P*(*y*)
!!*y z*.[| *r y z*; $r\hat{}++$ *z b*; *P*(*z*) |] ==> *P*(*y*)
shows *P a*
apply (*rule tranclp-induct* [*OF tranclp-converseI*, *OF conversepI*, *OF major*])
apply (*rule cases*)
apply (*erule conversepD*)
apply (*blast intro*: *assms dest*!: *tranclp-converseD*)
done

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

lemma *tranclpD*: $R\hat{}++$ *x y* ==> *EX z*. *R x z* ∧ $R\hat{}**$ *z y*
apply (*erule converse-tranclp-induct*)
apply *auto*
apply (*blast intro*: *rtranclp-trans*)
done

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *converse-tranclpE*:
assumes *major*: *tranclp r x z*
assumes *base*: *r x z* ==> *P*
assumes *step*: ⋀ *y*. [| *r x y*; *tranclp r y z* |] ==> *P*
shows *P*
proof −
from *tranclpD*[*OF major*]
obtain *y* **where** *r x y* **and** *rtranclp r y z* **by** *iprover*
from *this*(*2*) **show** *P*
proof (*cases rule*: *rtranclp.cases*)
case *rtrancl-refl*
with ‹*r x y*› *base* **show** *P* **by** *iprover*
next
case *rtrancl-into-rtrancl*
from *this* **have** *tranclp r y z*
by (*iprover intro*: *rtranclp-into-tranclp1*)

with ‹*r x y*› *step* **show** *P* **by** *iprover*
qed
qed

lemmas *converse-tranclE* = *converse-tranclpE* [*to-set*]

lemma *tranclD2*:
(*x*, *y*) ∈ R^+ ⟹ ∃ *z*. (*x*, *z*) ∈ R^* ∧ (*z*, *y*) ∈ *R*
by (*blast elim*: *tranclE intro*: *trancl-into-rtrancl*)

lemma *irrefl-tranclI*: *r*^−1 ∩ *r*^* = {} ==> (*x*, *x*) ∉ *r*^+
by (*blast elim*: *tranclE dest*: *trancl-into-rtrancl*)

lemma *irrefl-trancl-rD*: !!*X*. *ALL x*. (*x*, *x*) ∉ *r*^+ ==> (*x*, *y*) ∈ *r* ==> *x* ≠ *y*
by (*blast dest*: *r-into-trancl*)

lemma *trancl-subset-Sigma-aux*:
(*a*, *b*) ∈ *r*^* ==> *r* ⊆ *A* × *A* ==> *a* = *b* ∨ *a* ∈ *A*
by (*induct rule*: *rtrancl-induct*) *auto*

lemma *trancl-subset-Sigma*: *r* ⊆ *A* × *A* ==> *r*^+ ⊆ *A* × *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*^++)^== = *r*^**
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*^=)^+ = *r*^*
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*]: {}^+ = {}
by (*auto elim*: *trancl-induct*)

lemma *rtrancl-empty* [*simp*]: {}^* = *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*)

lemma *trancl-unfold-right*: $r\hat{}+ = r\hat{}* \ O\ r$
by (*auto dest*: *tranclD2 intro*: *rtrancl-into-trancl1*)

lemma *trancl-unfold-left*: $r\hat{}+ = r \ O\ r\hat{}*$
by (*auto dest*: *tranclD intro*: *rtrancl-into-trancl2*)

Simplifying nested closures

lemma *rtrancl-trancl-absorb*[*simp*]: $(R\hat{}*)\hat{}+ = R\hat{}*$
by (*simp add*: *trans-rtrancl*)

lemma *trancl-rtrancl-absorb*[*simp*]: $(R\hat{}+)\hat{}* = R\hat{}*$
by (*subst reflcl-trancl*[*symmetric*]) *simp*

lemma *rtrancl-reflcl-absorb*[*simp*]: $(R\hat{}*)\hat{}= = R\hat{}*$
by *auto*

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*:
$[\![\ single\text{-}valued\ r;\ (x,y) \in r\hat{}*;\ (x,z) \in r\hat{}*\ ]\!]$
$\Longrightarrow (y,z) \in r\hat{}* \vee (z,y) \in r\hat{}*$
apply (*erule rtrancl-induct*)
apply *simp*
apply (*erule disjE*)
apply (*blast elim*:*converse-rtranclE dest*:*single-valuedD*)
apply(*blast intro*:*rtrancl-trans*)
done

lemma *r-r-into-trancl*: $(a,\ b) \in R ==> (b,\ c) \in R ==> (a,\ c) \in R\hat{}+$
by (*fast intro*: *trancl-trans*)

lemma *trancl-into-trancl* [*rule-format*]:
$(a,\ b) \in r^{+} ==> (b,\ c) \in r --> (a,c) \in r^{+}$
apply (*erule trancl-induct*)
apply (*fast intro*: *r-r-into-trancl*)
apply (*fast intro*: *r-r-into-trancl trancl-trans*)
done

lemma *tranclp-rtranclp-tranclp*:
$r^{++}\ a\ b ==> r^{**}\ b\ c ==> r^{++}\ a\ c$
apply (*drule tranclpD*)
apply (*elim exE conjE*)
apply (*drule rtranclp-trans*, *assumption*)
apply (*drule rtranclp-into-tranclp2*, *assumption*, *assumption*)
done

lemmas *trancl-rtrancl-trancl = tranclp-rtranclp-tranclp* [*to-set*]

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl.trancl-into-trancl trancl-into-trancl2
rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

23.4 The power operation on relations

$R \hat{} \hat{} n = R\ O\ ...\ O\ R$, the n-fold composition of R

overloading
relpow == *compow* :: *nat* ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set*
begin

primrec *relpow* :: *nat* ⇒ ('*a* × '*a*) *set* ⇒ ('*a* × '*a*) *set* **where**
  relpow 0 R = Id
| *relpow* (*Suc n*) *R* = (*R* ^^ *n*) *O R*

end

lemma *rel-pow-1* [*simp*]:
fixes *R* :: ('*a* × '*a*) *set*
shows *R* ^^ *1* = *R*
by *simp*

lemma *rel-pow-0-I*:
(*x*, *x*) ∈ *R* ^^ *0*
by *simp*

lemma *rel-pow-Suc-I*:
(*x*, *y*) ∈ *R* ^^ *n* ⟹ (*y*, *z*) ∈ *R* ⟹ (*x*, *z*) ∈ *R* ^^ *Suc n*
by *auto*

lemma *rel-pow-Suc-I2*:
(*x*, *y*) ∈ *R* ⟹ (*y*, *z*) ∈ *R* ^^ *n* ⟹ (*x*, *z*) ∈ *R* ^^ *Suc n*
by (*induct n arbitrary*: *z*) (*simp*, *fastsimp*)

lemma *rel-pow-0-E*:
(*x*, *y*) ∈ *R* ^^ *0* ⟹ (*x* = *y* ⟹ *P*) ⟹ *P*
by *simp*

lemma *rel-pow-Suc-E*:
(*x*, *z*) ∈ *R* ^^ *Suc n* ⟹ (⋀*y*. (*x*, *y*) ∈ *R* ^^ *n* ⟹ (*y*, *z*) ∈ *R* ⟹ *P*) ⟹ *P*
by *auto*

lemma *rel-pow-E*:
(*x*, *z*) ∈ *R* ^^ *n* ⟹ (*n* = *0* ⟹ *x* = *z* ⟹ *P*)
⟹ (⋀*y m*. *n* = *Suc m* ⟹ (*x*, *y*) ∈ *R* ^^ *m* ⟹ (*y*, *z*) ∈ *R* ⟹ *P*)
⟹ *P*
by (*cases n*) *auto*

lemma *rel-pow-Suc-D2*:
(*x*, *z*) ∈ *R* ^^ *Suc n* ⟹ (∃ *y*. (*x*, *y*) ∈ *R* ∧ (*y*, *z*) ∈ *R* ^^ *n*)
apply (*induct n arbitrary*: *x z*)
apply (*blast intro*: *rel-pow-0-I elim*: *rel-pow-0-E rel-pow-Suc-E*)
apply (*blast intro*: *rel-pow-Suc-I elim*: *rel-pow-0-E rel-pow-Suc-E*)
done

lemma *rel-pow-Suc-E2*:
(*x*, *z*) ∈ *R* ^^ *Suc n* ⟹ (⋀*y*. (*x*, *y*) ∈ *R* ⟹ (*y*, *z*) ∈ *R* ^^ *n* ⟹ *P*) ⟹ *P*
by (*blast dest*: *rel-pow-Suc-D2*)

lemma *rel-pow-Suc-D2'*:
∀ *x y z*. (*x*, *y*) ∈ *R* ^^ *n* ∧ (*y*, *z*) ∈ *R* ⟶ (∃ *w*. (*x*, *w*) ∈ *R* ∧ (*w*, *z*) ∈ *R* ^^ *n*)
by (*induct n*) (*simp-all*, *blast*)

lemma *rel-pow-E2*:
(*x*, *z*) ∈ *R* ^^ *n* ⟹ (*n* = *0* ⟹ *x* = *z* ⟹ *P*)
⟹ (⋀*y m*. *n* = *Suc m* ⟹ (*x*, *y*) ∈ *R* ⟹ (*y*, *z*) ∈ *R* ^^ *m* ⟹ *P*)
⟹ *P*
apply (*cases n*, *simp*)
apply (*cut-tac n*=*nat* **and** *R*=*R* **in** *rel-pow-Suc-D2'*, *simp*, *blast*)
done

lemma *rel-pow-add*: *R* ^^ (*m*+*n*) = *R*^^*m* *O* *R*^^*n*
by(*induct n*) *auto*

lemma *rel-pow-commute*: *R O R* ^^ *n* = *R* ^^ *n O R*
by (*induct n*) (*simp*, *simp add*: *O-assoc* [*symmetric*])

lemma *rtrancl-imp-UN-rel-pow*:
assumes *p* ∈ *R*^*
shows *p* ∈ (⋃ *n*. *R* ^^ *n*)
proof (*cases p*)
case (*Pair x y*)
with *assms* **have** (*x*, *y*) ∈ *R*^* **by** *simp*
then have (*x*, *y*) ∈ (⋃ *n*. *R* ^^ *n*) **proof** *induct*
case *base* **show** *?case* **by** (*blast intro*: *rel-pow-0-I*)
next

```
    case step then show ?case by (blast intro: rel-pow-Suc-I)
  qed
  with Pair show ?thesis by simp
qed

lemma rel-pow-imp-rtrancl:
  assumes p ∈ R ^^ n
  shows p ∈ R^*
proof (cases p)
  case (Pair x y)
  with assms have (x, y) ∈ R ^^ n by simp
  then have (x, y) ∈ R^* proof (induct n arbitrary: x y)
    case 0 then show ?case by simp
  next
    case Suc then show ?case
      by (blast elim: rel-pow-Suc-E intro: rtrancl-into-rtrancl)
  qed
  with Pair show ?thesis by simp
qed

lemma rtrancl-is-UN-rel-pow:
  R^* = (⋃n. R ^^ n)
  by (blast intro: rtrancl-imp-UN-rel-pow rel-pow-imp-rtrancl)

lemma rtrancl-power:
  p ∈ R^* ⟷ (∃n. p ∈ R ^^ n)
  by (simp add: rtrancl-is-UN-rel-pow)

lemma trancl-power:
  p ∈ R^+ ⟷ (∃n > 0. p ∈ R ^^ n)
  apply (cases p)
  apply simp
  apply (rule iffI)
   apply (drule tranclD2)
   apply (clarsimp simp: rtrancl-is-UN-rel-pow)
   apply (rule-tac x=Suc n in exI)
   apply (clarsimp simp: rel-comp-def)
   apply fastsimp
  apply clarsimp
  apply (case-tac n, simp)
  apply clarsimp
  apply (drule rel-pow-imp-rtrancl)
  apply (drule rtrancl-into-trancl1) apply auto
  done

lemma rtrancl-imp-rel-pow:
  p ∈ R^* ⟹ ∃n. p ∈ R ^^ n
  by (auto dest: rtrancl-imp-UN-rel-pow)
```

lemma *single-valued-rel-pow*:
fixes $R :: ('a * 'a)$ *set*
shows *single-valued* $R \Longrightarrow$ *single-valued* $(R \hat{\ }\hat{\ } n)$
apply (*induct n arbitrary*: *R*)
apply *simp-all*
apply (*rule single-valuedI*)
apply (*fast dest*: *single-valuedD elim*: *rel-pow-Suc-E*)
done

23.5 Setup of transitivity reasoner

ML ⟪

```
structure Trancl-Tac = Trancl-Tac
(
  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;
);

structure Tranclp-Tac = Trancl-Tac
(
  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;
);
⟫
```

declaration ⟪ *fn - =>*
Simplifier.map-ss (fn ss => ss
addSolver (mk-solver' Trancl (Trancl-Tac.trancl-tac o Simplifier.the-context))
addSolver (mk-solver' Rtrancl (Trancl-Tac.rtrancl-tac o Simplifier.the-context))
addSolver (mk-solver' Tranclp (Tranclp-Tac.trancl-tac o Simplifier.the-context))
addSolver (mk-solver' Rtranclp (Tranclp-Tac.rtrancl-tac o Simplifier.the-context)))
⟫

Optional methods.

method-setup *trancl* =
⟪ *Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.trancl-tac)* ⟫
⟪ *simple transitivity reasoner* ⟫
method-setup *rtrancl* =
⟪ *Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.rtrancl-tac)* ⟫
⟪ *simple transitivity reasoner* ⟫
method-setup *tranclp* =
⟪ *Scan.succeed (SIMPLE-METHOD' o Tranclp-Tac.trancl-tac)* ⟫
⟪ *simple transitivity reasoner (predicate version)* ⟫
method-setup *rtranclp* =
⟪ *Scan.succeed (SIMPLE-METHOD' o Tranclp-Tac.rtrancl-tac)* ⟫
⟪ *simple transitivity reasoner (predicate version)* ⟫

end

24 Wellfounded: Well-founded Recursion

theory *Wellfounded*
imports *Transitive-Closure*
uses (*Tools/Function/size.ML*)
begin

24.1 Basic Definitions

definition *wf* :: ('*a* * '*a*) *set* => *bool* **where**
wf(r) == (!*P*. (!*x*. (!*y*. (*y*,*x*):*r* --> *P(y)*) --> *P(x)*) --> (!*x*. *P(x)*)))

definition *wfP* :: ('*a* => '*a* => *bool*) => *bool* **where**

wfP r == wf {(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*

lemma *wf-asym*:
assumes *wf r* (*a*, *x*) ∈ *r*
obtains (*x*, *a*) ∉ *r*
by (*drule wf-not-sym*[*OF assms*])

lemma *wf-not-refl* [*simp*]: *wf r* ==> (*a*, *a*) ~: *r*
by (*blast elim*: *wf-asym*)

lemma *wf-irrefl*: **assumes** *wf r* **obtains** (*a*, *a*) ∉ *r*
by (*drule wf-not-refl*[*OF assms*])

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*: *class.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*)

24.2 Basic Results

Point-free characterization of well-foundedness

lemma *wfE-pf*:
assumes *wf*: *wf R*
assumes *a*: *A* ⊆ *R* `` *A*
shows *A* = {}
proof −
{ fix *x*
from *wf* **have** *x* ∉ *A*
proof *induct*
fix *x* **assume** ⋀*y*. (*y*, *x*) ∈ *R* ⟹ *y* ∉ *A*
then have *x* ∉ *R* `` *A* **by** *blast*
with *a* **show** *x* ∉ *A* **by** *blast*
qed
} thus *?thesis* **by** *auto*
qed

lemma *wfI-pf*:
assumes *a*: ⋀*A*. *A* ⊆ *R* `` *A* ⟹ *A* = {}
shows *wf R*
proof (*rule wfUNIVI*)
fix *P* :: '*a* ⇒ *bool* **and** *x*
let *?A* = {*x*. ¬ *P x*}
assume ∀*x*. (∀*y*. (*y*, *x*) ∈ *R* ⟶ *P y*) ⟶ *P x*
then have *?A* ⊆ *R* `` *?A* **by** *blast*
with *a* **show** *P x* **by** *blast*
qed

Minimal-element characterization of well-foundedness

lemma *wfE-min*:
assumes *wf*: *wf R* **and** *Q*: *x* ∈ *Q*
obtains *z* **where** *z* ∈ *Q* ⋀*y*. (*y*, *z*) ∈ *R* ⟹ *y* ∉ *Q*
using *Q wfE-pf*[*OF wf*, *of Q*] **by** *blast*

lemma *wfI-min*:
assumes *a*: ⋀*x Q*. *x* ∈ *Q* ⟹ ∃ *z*∈*Q*. ∀*y*. (*y*, *z*) ∈ *R* ⟶ *y* ∉ *Q*
shows *wf R*
proof (*rule wfI-pf*)
fix *A* **assume** *b*: *A* ⊆ *R* `` *A*

```
  { fix x assume x ∈ A
    from a[OF this] b have False by blast
  }
  thus A = {} by blast
qed

lemma wf-eq-minimal: wf r = (∀ Q x. x∈Q --> (∃ z∈Q. ∀ y. (y,z)∈r -->
y∉Q))
apply auto
apply (erule wfE-min, assumption, blast)
apply (rule wfI-min, auto)
done

lemmas wfP-eq-minimal = wf-eq-minimal [to-pred]
```

Well-foundedness of transitive closure

```
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

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*)

lemma *wfP-empty* [*iff*]:
wfP (λ*x y*. *False*)
proof −
have *wfP bot* **by** (*fact wf-empty* [*to-pred bot-empty-eq2*])
then show *?thesis* **by** (*simp add*: *bot-fun-eq bot-bool-eq*)
qed

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

Exponentiation

lemma *wf-exp*:
assumes *wf* (*R* ^^ *n*)
shows *wf R*
proof (*rule wfI-pf*)
fix *A* **assume** *A* ⊆ *R* `` *A*
then have *A* ⊆ (*R* ^^ *n*) `` *A* **by** (*induct n*) *force*+
with ‹*wf* (*R* ^^ *n*)›
show *A* = {} **by** (*rule wfE-pf*)
qed

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

24.3 Well-Foundedness Results for Unions

lemma *wf-union-compatible*:
assumes *wf R wf S*
assumes *R O S* ⊆ *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 $y \in Q$
with ‹$y \notin$ *?Q'*›
obtain w **where** $(w, y) \in R$ **and** $w \in Q$ **by** *auto*
from ‹$(w, y) \in R$› ‹$(y, z) \in S$› **have** $(w, z) \in R\ O\ S$ **by** (*rule rel-compI*)
with ‹$R\ O\ S \subseteq R$› **have** $(w, z) \in R$ **..**
with ‹$z \in$ *?Q'*› **have** $w \notin Q$ **by** *blast*
with ‹$w \in Q$› **show** *False* **by** *contradiction*
qed
}
with ‹$z \in$ *?Q'*› **show** $\exists z{\in}Q.\ \forall y.\ (y, z) \in R \cup S \longrightarrow y \notin Q$ **by** *blast*
qed

Well-foundedness of indexed union with disjoint domains and ranges

lemma *wf-UN*: [| *ALL i:I. wf(r i)*;
ALL i:I. ALL j:I. r i ~= r j --> Domain(r i) Int Range(r j) = {}
|] ==> *wf(UN i:I. r i)*
apply (*simp only*: *wf-eq-minimal*, *clarify*)
apply (*rename-tac A a*, *case-tac EX i:I. EX a:A. EX b:A. (b,a) : r i*)
prefer *2*
apply *force*
apply *clarify*
apply (*drule bspec*, *assumption*)
apply (*erule-tac x={a. a:A & (EX b:A. (b,a) : r i) } in allE*)
apply (*blast elim!: allE*)
done

lemma *wfP-SUP*:
$\forall i.\ wfP\ (r\ i) \Longrightarrow \forall i\ j.\ r\ i \neq r\ j \longrightarrow inf\ (DomainP\ (r\ i))\ (RangeP\ (r\ j)) = bot$
$\Longrightarrow wfP\ (SUPR\ UNIV\ r)$
by (*rule wf-UN* [**where** *I=UNIV* **and** *r=λi. {(x, y). r i x y}*, *to-pred SUP-UN-eq2*])
(*simp-all add*: *Collect-def*)

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*:
$wf\ (R \cup S) = wf\ (R\ O\ R \cup S\ O\ R \cup S)$ (**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) ∈ S O R ⟹ 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) ∈ S O R 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*])

24.4 Acyclic relations

definition *acyclic* :: ('*a* * '*a*) *set* => *bool* **where**
acyclic r == !*x*. (*x*,*x*) ~: *r*^+

abbreviation *acyclicP* :: ('*a* => '*a* => *bool*) => *bool* **where**
acyclicP r == *acyclic* {(*x*, *y*). *r x y*}

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*)

24.5 *nat* is well-founded

lemma *less-nat-rel*: *op* < = (λ*m n*. *n* = *Suc m*) ^++
proof (*rule ext*, *rule ext*, *rule iffI*)
  fix *n m* :: *nat*
  assume *m* < *n*
  then show (λ*m n*. *n* = *Suc m*) ^++ *m n*
  proof (*induct n*)
    case *0* **then show** *?case* **by** *auto*
  next
    case (*Suc n*) **then show** *?case*
      by (*auto simp add*: *less-Suc-eq-le le-less intro*: *tranclp.trancl-into-trancl*)
  qed
next
  fix *n m* :: *nat*
  assume (λ*m n*. *n* = *Suc m*) ^++ *m n*
  then show *m* < *n*
    by (*induct n*)
      (*simp-all add*: *less-Suc-eq-le reflexive le-less*)
qed

definition
  pred-nat :: (*nat* ∗ *nat*) *set* **where**
  pred-nat = {(*m*, *n*). *n* = *Suc m*}

definition
  less-than :: (*nat* ∗ *nat*) *set* **where**
  less-than = *pred-nat*^+

lemma *less-eq*: (*m*, *n*) ∈ *pred-nat*^+ ⟷ *m* < *n*
  unfolding *less-nat-rel pred-nat-def trancl-def* **by** *simp*

lemma *pred-nat-trancl-eq-le*:
  (*m*, *n*) ∈ *pred-nat*^∗ ⟷ *m* ≤ *n*
  unfolding *less-eq rtrancl-eq-or-trancl* **by** *auto*

lemma *wf-pred-nat*: *wf pred-nat*
  apply (*unfold wf-def pred-nat-def*, *clarify*)
  apply (*induct-tac x*, *blast*+)

done

lemma *wf-less-than* [*iff*]: *wf less-than*
by (*simp add*: *less-than-def wf-pred-nat* [*THEN wf-trancl*])

lemma *trans-less-than* [*iff*]: *trans less-than*
by (*simp add*: *less-than-def*)

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*])

24.6 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*: $R1 \leq R2$
  shows $accp\ R2 \leq accp\ R1$
proof (*rule predicate1I*)
  fix x **assume** $accp\ R2\ x$
  then show $accp\ R1\ x$
  proof (*induct x*)
    fix x
    assume *ih*: $\bigwedge y.\ R2\ y\ x \Longrightarrow accp\ R1\ y$
    with *sub* **show** $accp\ R1\ x$
      by (*blast intro*: *accp.accI*)
  qed
qed

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:
  assumes *subset*: $D \leq accp\ R$
    and *dcl*: $\bigwedge x\ z.\ [\![D\ x;\ R\ z\ x]\!] \Longrightarrow D\ z$
    and $D\ x$
    and *istep*: $\bigwedge x.\ [\![D\ x;\ (\bigwedge z.\ R\ z\ x \Longrightarrow P\ z)]\!] \Longrightarrow P\ x$
  shows $P\ x$
proof −
  from *subset* **and** ‹$D\ x$›
  have $accp\ R\ x$ **..**
  then show $P\ x$ **using** ‹$D\ x$›
  proof (*induct x*)
    fix x
    assume $D\ x$
      and $\bigwedge y.\ R\ y\ x \Longrightarrow D\ y \Longrightarrow P\ y$
    with *dcl* **and** *istep* **show** $P\ x$ **by** *blast*
  qed
qed

Set versions of the above theorems

lemmas *acc-induct* = *accp-induct* [*to-set*]

lemmas *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]

lemmas *acc-downward* = *accp-downward* [*to-set*]

lemmas *not-acc-down* = *not-accp-down* [*to-set*]

lemmas *acc-downwards-aux* = *accp-downwards-aux* [*to-set*]

lemmas *acc-downwards* = *accp-downwards* [*to-set*]

lemmas *acc-wfI = accp-wfPI [to-set]*

lemmas *acc-wfD = accp-wfPD [to-set]*

lemmas *wf-acc-iff = wfP-accp-iff [to-set]*

lemmas *acc-subset = accp-subset [to-set pred-subset-eq]*

lemmas *acc-subset-induct = accp-subset-induct [to-set pred-subset-eq]*

24.7 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

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* $X \Longrightarrow$ *finite* $Y \Longrightarrow Y \neq \{\} \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow \exists\, y{\in}Y.\ (x,\ y) \in R) \Longrightarrow (X,\ Y) \in$ *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** $M \in$ *acc* (*max-ext r*) (**is** - $\in$ *?W*)
proof *cases*
assume *finite M*
thus *?thesis*
proof (*induct M*)
show $\{\} \in$ *?W*
by (*rule accI*) (*auto elim*: *max-ext.cases*)
next
fix $M\ a$ **assume** $M \in$ *?W finite M*
with *wf* **show** *insert a M* $\in$ *?W*
proof (*induct arbitrary*: M)
fix $M\ a$
assume $M \in$ *?W* **and** [*intro*]: *finite M*
assume *hyp*: $\bigwedge b\ M.\ (b,\ a) \in r \Longrightarrow M \in$ *?W* $\Longrightarrow$ *finite M* $\Longrightarrow$ *insert b M* $\in$ *?W*
{
fix $N\ M$ :: *'a set*
assume *finite N finite M*
then
have $[\![M \in$ *?W* ; $(\bigwedge y.\ y \in N \Longrightarrow (y,\ a) \in r)]\!] \Longrightarrow\ N \cup M \in$ *?W*
by (*induct N arbitrary*: M) (*auto simp*: *hyp*)
}
note *add-less* = *this*

show *insert a M* $\in$ *?W*
proof (*rule accI*)
fix N **assume** *Nless*: $(N,$ *insert a M*$) \in$ *max-ext r*
hence *asm1*: $\bigwedge x.\ x \in N \Longrightarrow (x,\ a) \in r \vee (\exists\, y \in M.\ (x,\ y) \in r)$
by (*auto elim*!: *max-ext.cases*)

let *?N1* = $\{\ n \in N.\ (n,\ a) \in r\ \}$
let *?N2* = $\{\ n \in N.\ (n,\ a) \notin r\ \}$
have *N*: *?N1* $\cup$ *?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* $\in$ *?W*
proof *cases*
assume [*simp*]: $M = \{\}$
have *Mw*: $\{\} \in$ *?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
```

24.8 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

24.9 size of a datatype value

use *Tools/Function/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* [*no-atp*]

lemma [*code*]:
size (*P* :: *'a Predicate.pred*) = *0* **by** (*cases P*) *simp*

lemma [*code*]:
pred-size f P = *0* **by** (*cases P*) *simp*

end

25 FunDef: Function Definitions and Termination Proofs

theory *FunDef*
imports *Wellfounded*
uses
Tools/prop-logic.ML
Tools/sat-solver.ML
(*Tools/Function/function-lib.ML*)
(*Tools/Function/function-common.ML*)
(*Tools/Function/context-tree.ML*)
(*Tools/Function/function-core.ML*)
(*Tools/Function/sum-tree.ML*)
(*Tools/Function/mutual.ML*)
(*Tools/Function/pattern-split.ML*)
(*Tools/Function/function.ML*)
(*Tools/Function/relation.ML*)
(*Tools/Function/measure-functions.ML*)
(*Tools/Function/lexicographic-order.ML*)
(*Tools/Function/pat-completeness.ML*)
(*Tools/Function/fun.ML*)
(*Tools/Function/induction-schema.ML*)

(*Tools/Function/termination.ML*)
(*Tools/Function/scnp-solve.ML*)
(*Tools/Function/scnp-reconstruct.ML*)
begin

25.1 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-default d P*)
by (*simp add*: *theI' THE-default-def*)

lemma *THE-default1-equality*:
$[\![\exists!x.\ P\ x;\ P\ a]\!] \Longrightarrow$ *THE-default d P* = *a*
by (*simp add*: *the1-equality THE-default-def*)

lemma *THE-default-none*:
$\neg(\exists!x.\ P\ x) \Longrightarrow$ *THE-default d P* = *d*
by (*simp add*:*THE-default-def*)

lemma *fundef-ex1-existence*:
assumes *f-def*: *f* == $(\lambda x::'a.$ *THE-default* (*d x*) $(\lambda y.$ *G x y*))
assumes *ex1*: $\exists!y.$ *G x y*
shows *G x* (*f x*)
apply (*simp only*: *f-def*)
apply (*rule THE-defaultI'*)
apply (*rule ex1*)
done

lemma *fundef-ex1-uniqueness*:
assumes *f-def*: *f* == $(\lambda x::'a.$ *THE-default* (*d x*) $(\lambda y.$ *G x y*))
assumes *ex1*: $\exists!y.$ *G x y*
assumes *elm*: *G x* (*h x*)
shows *h x* = *f x*
apply (*simp only*: *f-def*)
apply (*rule THE-default1-equality* [*symmetric*])
apply (*rule ex1*)
apply (*rule elm*)
done

lemma *fundef-ex1-iff*:
assumes *f-def*: *f* == $(\lambda x::'a.$ *THE-default* (*d x*) $(\lambda y.$ *G x y*))
assumes *ex1*: $\exists!y.$ *G x y*
shows (*G x y*) = (*f x* = *y*)
apply (*auto simp*:*ex1 f-def THE-default1-equality*)
apply (*rule THE-defaultI'*)

```
  apply (rule ex1)
  done

lemma fundef-default-value:
  assumes f-def: f == (λx::'a. THE-default (d x) (λy. G x y))
  assumes graph: ⋀x y. G x y ⟹ 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/function-lib.ML
use Tools/Function/function-common.ML
use Tools/Function/context-tree.ML
use Tools/Function/function-core.ML
use Tools/Function/sum-tree.ML
use Tools/Function/mutual.ML
use Tools/Function/pattern-split.ML
use Tools/Function/relation.ML
use Tools/Function/function.ML
use Tools/Function/pat-completeness.ML
use Tools/Function/fun.ML
use Tools/Function/induction-schema.ML

setup ⟪
  Function.setup
  #> Pat-Completeness.setup
  #> Function-Fun.setup
  #> Induction-Schema.setup
⟫
```

25.2 Measure Functions

inductive *is-measure* :: $('a \Rightarrow nat) \Rightarrow bool$
where *is-measure-trivial*: *is-measure f*

use *Tools/Function/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/lexicographic-order.ML*
setup *Lexicographic-Order.setup*

25.3 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* **.**

25.4 Simp rules for termination proofs

lemma *termination-basic-simps*[*termination-simp*]:
$x < (y::nat) \Longrightarrow x < y + z$
$x < z \Longrightarrow x < y + z$
$x \leq y \Longrightarrow x \leq y + (z::nat)$
$x \leq z \Longrightarrow x \leq y + (z::nat)$
$x < y \Longrightarrow x \leq (y::nat)$
by *arith+*

declare *le-imp-less-Suc*[*termination-simp*]

lemma *prod-size-simp*[*termination-simp*]:
prod-size f g p = *f* (*fst p*) + *g* (*snd p*) + *Suc 0*
by (*induct p*) *auto*

25.5 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*])

25.6 Reduction Pairs

definition
reduction-pair P = (*wf* (*fst P*) ∧ *fst P O snd P* ⊆ *fst P*)

lemma *reduction-pairI*[*intro*]: *wf R* ⟹ *R O S* ⊆ *R* ⟹ *reduction-pair* (*R*, *S*)
unfolding *reduction-pair-def* **by** *auto*

lemma *reduction-pair-lemma*:
assumes *rp*: *reduction-pair P*
assumes *R* ⊆ *fst P*
assumes *S* ⊆ *snd P*
assumes *wf S*
shows *wf* (*R* ∪ *S*)
proof −
from *rp* ‹*S* ⊆ *snd P*› **have** *wf* (*fst P*) *fst P O S* ⊆ *fst P*
unfolding *reduction-pair-def* **by** *auto*
with ‹*wf S*› **have** *wf* (*fst P* ∪ *S*)
by (*auto intro*: *wf-union-compatible*)
moreover from ‹*R* ⊆ *fst P*› **have** *R* ∪ *S* ⊆ *fst P* ∪ *S* **by** *auto*
ultimately show *?thesis* **by** (*rule wf-subset*)
qed

definition
rp-inv-image = (λ(*R*,*S*) *f*. (*inv-image R f*, *inv-image S f*))

lemma *rp-inv-image-rp*:
reduction-pair P ⟹ *reduction-pair* (*rp-inv-image P f*)
unfolding *reduction-pair-def rp-inv-image-def split-def*
by *force*

25.7 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* ∪ {({}, {})}
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*$]\!] \Longrightarrow (insert\ x\ X, Y) \in$ *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*$]\!] \Longrightarrow (insert\ x\ XS, YS) \in$ *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 *R O S* ⊆ *R*
shows *max-ext R O* (*max-ext S* ∪ {({},{})}) ⊆ *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 *R O S* ⊆ *R*
shows *min-ext R O* (*min-ext S* ∪ {({},{})}) ⊆ *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*)

25.8 Tool setup

use *Tools/Function/termination.ML*
use *Tools/Function/scnp-solve.ML*
use *Tools/Function/scnp-reconstruct.ML*

setup ⟪ *ScnpReconstruct.setup* ⟫

ML-val — setup inactive
⟪
Context.theory-map (*Function-Common.set-termination-prover*
(*ScnpReconstruct.decomp-scnp-tac* [*ScnpSolve.MAX*, *ScnpSolve.MIN*, *ScnpSolve.MS*]))
⟫

end

26 Extraction: Program extraction for HOL

theory *Extraction*
imports *Option*
uses *Tools/rewrite-hol-proof.ML*
begin

26.1 Setup

setup ⟪
  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 @ [*ProofRewriteRules.expand-of-class thy*]) *o*
    ProofRewriteRules.elim-vars (*curry Const* @{*const-name default*}))
⟫

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-atomize induct-atomize′ induct-rulify induct-rulify′
  induct-rulify-fallback induct-trueI
  True-implies-equals TrueE

lemmas [*extraction-expand-def*] =
  induct-forall-def induct-implies-def induct-equal-def induct-conj-def
  induct-true-def induct-false-def

datatype *sumbool* = *Left* | *Right*

26.2 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*))

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \longrightarrow Q) \equiv Type\ (TYPE('P \Rightarrow 'Q))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$typeof\ (\forall x.\ P\ x) \equiv Type\ (TYPE(Null))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$typeof\ (\forall x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \Rightarrow 'P))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$typeof\ (\exists x{::}'a.\ P\ x) \equiv Type\ (TYPE('a))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$typeof\ (\exists x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

26.3 Realizability

realizability
$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x::'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall x.\ P\ x)) \equiv (\forall x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg P) \equiv (\forall x::'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P::bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(\mathit{realizes}\ t\ (P = Q)) \equiv (\mathit{realizes}\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

26.4 Computational content of basic inference rules

theorem *disjE-realizer*:
assumes *r*: *case x of Inl p ⇒ P p | Inr q ⇒ Q q*
and *r1*: $\bigwedge p.\ P\ p \Longrightarrow R\ (f\ p)$ **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*:
$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (\mathit{fst}\ (p,\ q)) \wedge Q\ (\mathit{snd}\ (p,\ q))$
by *simp*

theorem *exI-realizer*:
$P\ y\ x \Longrightarrow P\ (\mathit{snd}\ (x,\ y))\ (\mathit{fst}\ (x,\ y))$ **by** *simp*

theorem *exE-realizer*: $P\ (\mathit{snd}\ p)\ (\mathit{fst}\ p) \Longrightarrow$

$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q\ (f\ x\ y)) \Longrightarrow Q\ (let\ (x,\ y) = p\ in\ f\ x\ y)$
by (*cases p*) (*simp add*: *Let-def*)

theorem *exE-realizer'*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q$ **by** (*cases p*) *simp*

realizers
impI (*P*, *Q*): $\lambda pq.\ pq$
Λ (*c*: -) (*d*: -) *P Q pq* (*h*: -). *allI* · - · *c* · (Λ *x*. *impI* · - · - · - · (*h* · *x*))

impI (*P*): *Null*
Λ (*c*: -) *P Q* (*h*: -). *allI* · - · *c* · (Λ *x*. *impI* · - · - · - · (*h* · *x*))

impI (*Q*): $\lambda q.\ q\ \Lambda$ (*c*: -) *P Q q*. *impI* · - · -

impI: *Null impI*

mp (*P*, *Q*): $\lambda pq.\ pq$
Λ (*c*: -) (*d*: -) *P Q pq* (*h*: -) *p*. *mp* · - · - · (*spec* · - · *p* · *c* · *h*)

mp (*P*): *Null*
Λ (*c*: -) *P Q* (*h*: -) *p*. *mp* · - · - · (*spec* · - · *p* · *c* · *h*)

mp (*Q*): $\lambda q.\ q\ \Lambda$ (*c*: -) *P Q q*. *mp* · - · -

mp: *Null mp*

allI (*P*): $\lambda p.\ p\ \Lambda$ (*c*: -) *P* (*d*: -) *p*. *allI* · - · *d*

allI: *Null allI*

spec (*P*): $\lambda x\ p.\ p\ x\ \Lambda$ (*c*: -) *P x* (*d*: -) *p*. *spec* · - · *x* · *d*

spec: *Null spec*

exI (*P*): $\lambda x\ p.\ (x,\ p)\ \Lambda$ (*c*: -) *P x* (*d*: -) *p*. *exI-realizer* · *P* · *p* · *x* · *c* · *d*

exI: $\lambda x.\ x\ \Lambda\ P\ x$ (*c*: -) (*h*: -). *h*

exE (*P*, *Q*): $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
Λ (*c*: -) (*d*: -) *P Q* (*e*: -) *p* (*h*: -) *pq*. *exE-realizer* · *P* · *p* · *Q* · *pq* · *c* · *e* · *d* · *h*

exE (*P*): *Null*
Λ (*c*: -) *P Q* (*d*: -) *p*. *exE-realizer'* · - · - · - · - · *c* · *d*

exE (*Q*): $\lambda x\ pq.\ pq\ x$
Λ (*c*: -) *P Q* (*d*: -) *x* (*h1*: -) *pq* (*h2*: -). *h2* · *x* · *h1*

exE: *Null*

Λ *P Q* (*c*: -) *x* (*h1*: -) (*h2*: -). *h2* · *x* • *h1*

conjI (*P*, *Q*): *Pair*
Λ (*c*: -) (*d*: -) *P Q p* (*h*: -) *q*. *conjI-realizer* · *P* · *p* · *Q* · *q* • *c* • *d* • *h*

conjI (*P*): λ*p*. *p*
Λ (*c*: -) *P Q p*. *conjI* · - · -

conjI (*Q*): λ*q*. *q*
Λ (*c*: -) *P Q* (*h*: -) *q*. *conjI* · - · - • *h*

conjI: *Null conjI*

conjunct1 (*P*, *Q*): *fst*
Λ (*c*: -) (*d*: -) *P Q pq*. *conjunct1* · - · -

conjunct1 (*P*): λ*p*. *p*
Λ (*c*: -) *P Q p*. *conjunct1* · - · -

conjunct1 (*Q*): *Null*
Λ (*c*: -) *P Q q*. *conjunct1* · - · -

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*
Λ (*c*: -) (*d*: -) *P Q pq*. *conjunct2* · - · -

conjunct2 (*P*): *Null*
Λ (*c*: -) *P Q p*. *conjunct2* · - · -

conjunct2 (*Q*): λ*p*. *p*
Λ (*c*: -) *P Q p*. *conjunct2* · - · -

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*
Λ (*c*: -) (*d*: -) *P Q p*. *iffD2* · - · - • (*sum.cases-1* · *P* · - · *p* • *arity-type-bool* • *c* • *d*)

disjI1 (*P*): *Some*
Λ (*c*: -) *P Q p*. *iffD2* · - · - • (*option.cases-2* · - · *P* · *p* • *arity-type-bool* • *c*)

disjI1 (*Q*): *None*
Λ (*c*: -) *P Q*. *iffD2* · - · - • (*option.cases-1* · - · - • *arity-type-bool* • *c*)

disjI1: *Left*
Λ *P Q*. *iffD2* · - · - • (*sumbool.cases-1* · - · - • *arity-type-bool*)

disjI2 (*P*, *Q*): *Inr*

Λ (d: -) (c: -) Q P q. *iffD2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*sum.cases-2* $\cdot$ - $\cdot$ - $\cdot$ Q $\cdot$ q $\cdot$ *arity-type-bool* $\cdot$ c $\cdot$ d)

disjI2 (P): *None*
Λ (c: -) Q P. *iffD2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*option.cases-1* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *arity-type-bool* $\cdot$ c)

disjI2 (Q): *Some*
Λ (c: -) Q P q. *iffD2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*option.cases-2* $\cdot$ - $\cdot$ - $\cdot$ Q $\cdot$ q $\cdot$ *arity-type-bool* $\cdot$ c)

disjI2: *Right*
Λ Q P. *iffD2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ (*sumbool.cases-2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *arity-type-bool*)

disjE (P, Q, R): λpq pr qr.
(*case pq of Inl p* $\Rightarrow$ *pr p* | *Inr q* $\Rightarrow$ *qr q*)
Λ (c: -) (d: -) (e: -) P Q R pq (*h1*: -) pr (*h2*: -) qr.
disjE-realizer $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ pq $\cdot$ R $\cdot$ pr $\cdot$ qr $\cdot$ c $\cdot$ d $\cdot$ e $\cdot$ *h1* $\cdot$ *h2*

disjE (Q, R): λpq pr qr.
(*case pq of None* $\Rightarrow$ *pr* | *Some q* $\Rightarrow$ *qr q*)
Λ (c: -) (d: -) P Q R pq (*h1*: -) pr (*h2*: -) qr.
disjE-realizer2 $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ pq $\cdot$ R $\cdot$ pr $\cdot$ qr $\cdot$ c $\cdot$ d $\cdot$ *h1* $\cdot$ *h2*

disjE (P, R): λpq pr qr.
(*case pq of None* $\Rightarrow$ *qr* | *Some p* $\Rightarrow$ *pr p*)
Λ (c: -) (d: -) P Q R pq (*h1*: -) pr (*h2*: -) qr (*h3*: -).
disjE-realizer2 $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ pq $\cdot$ R $\cdot$ qr $\cdot$ pr $\cdot$ c $\cdot$ d $\cdot$ *h1* $\cdot$ *h3* $\cdot$ *h2*

disjE (R): λpq pr qr.
(*case pq of Left* $\Rightarrow$ *pr* | *Right* $\Rightarrow$ *qr*)
Λ (c: -) P Q R pq (*h1*: -) pr (*h2*: -) qr.
disjE-realizer3 $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ pq $\cdot$ R $\cdot$ pr $\cdot$ qr $\cdot$ c $\cdot$ *h1* $\cdot$ *h2*

disjE (P, Q): *Null*
Λ (c: -) (d: -) P Q R pq. *disjE-realizer* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ pq $\cdot$ (λx. R) $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ c $\cdot$ d $\cdot$ *arity-type-bool*

disjE (Q): *Null*
Λ (c: -) P Q R pq. *disjE-realizer2* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ pq $\cdot$ (λx. R) $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ c $\cdot$ *arity-type-bool*

disjE (P): *Null*
Λ (c: -) P Q R pq (*h1*: -) (*h2*: -) (*h3*: -).
disjE-realizer2 $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ pq $\cdot$ (λx. R) $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ c $\cdot$ *arity-type-bool* $\cdot$ *h1* $\cdot$ *h3* $\cdot$ *h2*

disjE: *Null*
Λ P Q R pq. *disjE-realizer3* $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ pq $\cdot$ (λx. R) $\cdot$ - $\cdot$ - $\cdot$ - $\cdot$ *arity-type-bool*

FalseE (P): *default*
Λ (c: -) P. *FalseE* $\cdot$ -

FalseE: *Null FalseE*

notI (*P*): *Null*
  Λ (*c*: -) *P* (*h*: -). *allI* · - · *c* · (Λ *x*. *notI* · - · (*h* · *x*))

notI: *Null notI*

notE (*P*, *R*): λ*p*. *default*
  Λ (*c*: -) (*d*: -) *P R* (*h*: -) *p*. *notE* · - · - · (*spec* · - · *p* · *c* · *h*)

notE (*P*): *Null*
  Λ (*c*: -) *P R* (*h*: -) *p*. *notE* · - · - · (*spec* · - · *p* · *c* · *h*)

notE (*R*): *default*
  Λ (*c*: -) *P R*. *notE* · - · -

notE: *Null notE*

subst (*P*): λ*s t ps*. *ps*
  Λ (*c*: -) *s t P* (*d*: -) (*h*: -) *ps*. *subst* · *s* · *t* · *P ps* · *d* · *h*

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
  Λ (*d*: -) (*c*: -) *Q P pq* (*h*: -) *p*.
    mp · - · - · (*spec* · - · *p* · *d* · (*conjunct1* · - · - · *h*))

iffD1 (*P*): λ*p*. *p*
  Λ (*c*: -) *Q P p* (*h*: -). *mp* · - · - · (*conjunct1* · - · - · *h*)

iffD1 (*Q*): *Null*
  Λ (*c*: -) *Q P q1* (*h*: -) *q2*.
    mp · - · - · (*spec* · - · *q2* · *c* · (*conjunct1* · - · - · *h*))

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
  Λ (*c*: -) (*d*: -) *P Q pq* (*h*: -) *q*.
    mp · - · - · (*spec* · - · *q* · *d* · (*conjunct2* · - · - · *h*))

iffD2 (*P*): λ*p*. *p*
  Λ (*c*: -) *P Q p* (*h*: -). *mp* · - · - · (*conjunct2* · - · - · *h*)

iffD2 (*Q*): *Null*
  Λ (*c*: -) *P Q q1* (*h*: -) *q2*.
    mp · - · - · (*spec* · - · *q2* · *c* · (*conjunct2* · - · - · *h*))

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
  Λ (*c*: -) (*d*: -) *P Q pq* (*h1* : -) *qp* (*h2* : -). *conjI-realizer* ·
    (λ*pq*. ∀ *x*. *P x* ⟶ *Q* (*pq x*)) · *pq* ·
    (λ*qp*. ∀ *x*. *Q x* ⟶ *P* (*qp x*)) · *qp* ·
    (*arity-type-fun* · *c* · *d*) ·
    (*arity-type-fun* · *d* · *c*) ·
    (*allI* · - · *c* · (Λ *x*. *impI* · - · - · (*h1* · *x*))) ·
    (*allI* · - · *d* · (Λ *x*. *impI* · - · - · (*h2* · *x*)))

iffI (*P*): λ*p*. *p*
  Λ (*c*: -) *P Q* (*h1* : -) *p* (*h2* : -). *conjI* · - · - ·
    (*allI* · - · *c* · (Λ *x*. *impI* · - · - · (*h1* · *x*))) ·
    (*impI* · - · - · *h2*)

iffI (*Q*): λ*q*. *q*
  Λ (*c*: -) *P Q q* (*h1* : -) (*h2* : -). *conjI* · - · - ·
    (*impI* · - · - · *h1*) ·
    (*allI* · - · *c* · (Λ *x*. *impI* · - · - · (*h2* · *x*)))

iffI: *Null iffI*

end

27 Plain: Plain HOL

theory *Plain*
imports *Datatype Record FunDef Extraction*
begin

Plain bootstrap of fundamental HOL tools and packages; does not include *Hilbert-Choice*.

ML ⟪ *path-add* ~~/*src*/*HOL*/*Library* ⟫

end

References